Mehdi Dastani
Koen V. Hindriks
John-Jules Charles Meyer
Editors

Specification and Verification of Multi-agent Systems

Foreword by
Wiebe van der Hoek

Springer

Specification and Verification of Multi-agent Systems

Mehdi Dastani · Koen V. Hindriks ·
John-Jules Charles Meyer
Editors

Specification and Verification of Multi-agent Systems

Foreword by
Wiebe van der Hoek

Springer

Editors
Mehdi Dastani
Utrecht University
Dept. Information & Computing Sciences
Padualaan 14
3584 CH Utrecht
The Netherlands
mehdi@cs.uu.nl

Koen V. Hindriks
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands
k.v.hindriks@tudelft.nl

John-Jules Ch. Meyer
Utrecht University
Dept. Information & Computer Sciences
Padualaan 14
3584 CH Utrecht
The Netherlands
jj@cs.uu.nl

ISBN 978-1-4419-6983-5 e-ISBN 978-1-4419-6984-2
DOI 10.1007/978-1-4419-6984-2
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010930883

Foreword

In the last decade, multi-agent systems have both become *widely applied* and also *increasingly complex*. The applications include the use of agents as automous decision makers in often safety-critical, dangerous, or high impact scenarios (traffic control, autonomous satellites, computational markets). The complexity arises from the fact that not only do we expect the agent to make decisions in situations that are not anticipated at forehand, the agent also interacts with other complex agents, with humans, organisations, and it lives in a dynamic environment (activators of an agent can fail, communication is prone to error, human response may be ambiguous, rules of an organisation may leave behaviour open or over-constrained, and environments may change 'spontaneously' or as a result of other agents acting upon it).

Taking these two facts together call for a rigorous *Specification and Verification of Multi-Agent Systems*. Since intelligent agents are computational systems, it is no wonder that this activity builds upon and extends concepts and ideas of specifying and verifying computer-based systems in general. For one, an axiom is that the tools are mainly *logical*, or in any case *formal*. But although traditional techniques of specification and verification go a long way when reasoning about the correctness of a single agent, there are additional questions already to be asked at this level: do we 'only' require the agent to *behave* well under a pre-defined set of inputs, or do we want to allow for (partially) undefined scenarios? And do we care about the 'correctness' of an agent's beliefs, desires and intentions during a computation? How do we want to guarantee that the agent 'knows what he is doing', has 'reasonable desires' and 'only drops an intention if it(s goal) is fulfilled, or cannot be reasonably be fulfilled any longer'? And a predecessor of this 'how to guarantee' question is equally important: what do we exactly mean by those requirements?

In a multi-agent system, specification and verification becomes only harder and, indeed, more interesting. Once we have an understanding of how to make the agents behave correctly individually, is this property then also compositional, in the sense that it applies to the system as a whole? What are the requirements we need to impose on the interaction among the agents, the ability of the human users, the organisation the agents represent, or the environment as a whole, in order to make

the multi-agent system behave as required? If we have means to specify this, then how do we check this?

This book, with contributions from many world-leading authors in the field, gives many answers to the questions above, thereby indeed often generalising ideas that have been around in the specification and verification community. For instance, the book addresses theorem proving ('*all* implementations, or models, of the specification verify') as a means to verify properties of cognitive agents, refinement (making sure the runs of a system are in a desired class) as a compositional means to derive correct implementations from a given specification, and model checking ('does this *particular* implementation verify') as a technique to verify certain multi-agent system behaviour. Model checking is applied to agent communication, to practical agent programming languages, and to languages in which agent goals play a major role: a complexity study of model checking for temporal and strategic logics complements these applications. There is also a chapter proposing a hybrid approach which guarantees on the one hand that some desirable properties in the specification language are met by any implementation in the agent programming language, and on top of that, a debugging framework that checks for temporal and cognitive assertions.

The book also has chapters that focus on specification languages, for instance a cognitive agent specification language with declarative and procedural components, and a temporal trace language to express dynamic properties of multi-agent systems. Another chapter advocates to use one (term rewriting) language and its tools for prototyping, verifying and testing agent programs.

If one takes the autonomy of agents to the extreme, one should leave it to the agents to act in a correct way. Indeed, this book has a chapter where norms are used as a way to specify correct, or desired behaviour, and to make sure that the agents comply with the norm, a game-theoretic framework is proposed. Although all chapters mentioned above comply with the maxim of formality, there is also a chapter that challenges this axiom, and claims that formal verification for assurance of agent systems is, on its own, not enough.

Mehdi Dastani, Koen Hindriks and John-Jules Meyer are well-chosen editors of this book. The work of their *Intelligent Systems* group in Utrecht (now also continued in the Man Machine Interaction group in Delft by Koen) encompasses all aspects this book addresses: their long history of involvement in verifaction of programs, their early work originating in modal logics for specification of agents, their hands-on experience with implementing agents in 3APL and its successors, and their involvement, from its early days, in formal approaches to normative systems, makes them better suited to compose this volume than anybody else.

January 2010, Liverpool, UK *Wiebe van der Hoek*

Contents

List of Contributors

Natasha Alechina
University of Nottingham, School of Computer Science, UK
e-mail: nza@cs.nott.ac.uk

Lăcrămioara Aştefănoaei
CWI (Centrum Wiskunde en Informatica), The Netherlands
e-mail: astefano@cwi.nl

Jamal Bentahar
Concordia University, Concordia Institute for Information Systems Engineering, Canada
e-mail: bentahar@ciise.concordia.ca

Frank S. de Boer
CWI (Centrum Wiskunde en Informatica), The Netherlands
e-mail: F.S.de.Boer@cwi.nl

Rafael H. Bordini
Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
e-mail: r.bordini@inf.ufrgs.br

Nils Bulling
Dept. of Informatics, Niedersächsische Technische Hochschule, Standort Clausthal, Germany
e-mail: bulling@in.tu-clausthal.de

Mehdi Dastani
Universiteit Utrecht, Department of Information and Computing Sciences, The Netherlands
e-mail: mehdi@cs.uu.nl

Louise A. Dennis
Department of Computer Science, University of Liverpool, U.K.
e-mail: l.a.dennis@liverpool.ac.uk

Jürgen Dix
Dept. of Informatics, Niedersächsische Technische Hochschule, Standort Clausthal, Germany
e-mail: dix@in.tu-clausthal.de

Berndt Farwer
School of Engineering and Computing Sciences, Durham University, U.K.
e-mail: berndt.farwer@durham.ac.uk

Michael Fisher
Department of Computer Science, University of Liverpool, U.K.
e-mail: mfisher@liverpool.ac.uk

Dov Gabbay
Computer Science King's College London, U.K. and ICR University of Luxembourg, Luxembourg
e-mail: dov.gabbay@kcl.ac.uk

Davide Grossi
ILLC University of Amsterdam, The Netherlands
e-mail: d.grossi@uva.nl

Koen V. Hindriks
Delft University of Technology, Melkweg 4, Delft, The Netherlands
e-mail: k.v.hindriks@tudelft.nl

Wojciech Jamroga
Computer Science and Communications, University of Luxembourg, Luxembourg and Dept. of Informatics, Niedersächsische Technische Hochschule, Standort Clausthal, Germany
e-mail: wojtek.jamroga@uni.lu

Fahad Khan,
University of Nottingham, School of Computer Science, U.K.
e-mail: afk@cs.nott.ac.uk

Yves Lespérance
Department of Computer Science and Engineering, York University, Canada
e-mail: lesperan@cse.yorku.ca

Hector J. Levesque
Department of Computer Science, University of Toronto, Toronto, Canada
e-mail: hector@ai.toronto.edu

Brian Logan
University of Nottingham, School of Computer Science, U.K.
e-mail: bsl@cs.nott.ac.uk

John-Jules Ch. Meyer
Universiteit Utrecht, Department of Information and Computing Sciences, The Netherlands
e-mail: jj@cs.uu.nl

M. Birna van Riemsdijk
Delft University of Technology, The Netherlands
e-mail: m.b.vanriemsdijk@tudelft.nl

Steven Shapiro
Department of Computer Science, University of Toronto, Toronto, Canada
e-mail: steven@cs.toronto.edu

Alexei Sharpanskykh
Vrije Universiteit Amsterdam, Department of Artificial Intelligence, Amsterdam, The Netherlands
e-mail: sharp@cs.vu.nl

Leendert van der Torre
ICR University of Luxembourg, Luxembourg
e-mail: leendert@vandertorre.com

Jan Treur
Vrije Universiteit Amsterdam, Department of Artificial Intelligence, Amsterdam, The Netherlands
e-mail: treur@cs.vu.nl

Wei Wan
Concordia University, Concordia Institute for Information Systems Engineering, Canada
e-mail: bentahar@ciise.concordia.ca

Michael Winikoff
University of Otago, Dunedin, New Zealand
e-mail: michael.winikoff@otago.ac.nz

Chapter 1

Using Theorem Proving to Verify Properties of Agent Programs

N. Alechina, M. Dastani, F. Khan, B. Logan, and J.-J. Ch. Meyer

Abstract We present a sound and complete logic for automatic verification of SimpleAPL programs. SimpleAPL is a fragment of agent programming languages such as 3APL and 2APL designed for the implementation of cognitive agents with beliefs, goals and plans. Our logic is a variant of PDL, and allows the specification of safety and liveness properties of agent programs. We prove a correspondence between the operational semantics of SimpleAPL and the models of the logic for two example program execution strategies. We show how to translate agent programs written in SimpleAPL into expressions of the logic, and give an example in which we show how to verify correctness properties for a simple agent program using theorem-proving.

N. Alechina, F. Khan, B. Logan
University of Nottingham, School of Computer Science, U.K. e-mail: {nza,afk,bsl}@cs.nott.ac.uk

M. Dastani, J.-J. Ch. Meyer
Universiteit Utrecht, Department of Information and Computing Sciences, The Netherlands e-mail: {mehdi,jj}@cs.uu.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_1,

1.1 Introduction

The specification and verification of agent architectures and programs is a key problem in agent research and development. Formal verification provides a degree of certainty regarding system behaviour which is difficult or impossible to obtain using conventional testing methodologies, particularly when applied to autonomous systems operating in open environments. For example, the use of appropriate specification and verification techniques can allow agent researchers to check that agent architectures and programming languages conform to general principles of rational agency, or agent developers to check that a particular agent program will achieve the agent's goals in a given range of environments. Ideally, such techniques should allow specification of key aspects of the agent's architecture such as its execution cycle (e.g., to explore commitment under different program execution strategies), and should admit a fully automated verification procedure. However, while there has been considerable work on the formal verification of software systems and on logics of agency, it has proved difficult to bring this work to bear on verification of agent programs. On the one hand, it can be difficult to specify and verify relevant properties of agent programs using conventional formal verification techniques, and on the other, standard epistemic logics of agency (e.g., [169]) fail to take into account the computational limitations of agent implementations.

Since an agent program is a special kind of program, logics intended for the specification of conventional programs can be used for specifying agent programming languages. In this approach we have some set of propositional variables or predicates to encode the agent's state, and, for example, dynamic or temporal operators for describing how the state changes as the computation evolves. However, for agents based on the Belief-Desire-Intention model of agency, such an approach fails to capture important structure in the agent's state which can be usefully exploited in verification. For example, we could encode the fact that the agent has the belief that p as the proposition u_1, and the fact that the agent has the goal that p as the proposition u_2. However such an encoding obscures the key logical relationship between the two facts, making it difficult to express general properties such as 'an agent cannot have as a goal a proposition which it currently believes'. It therefore seems natural for a logical language intended for reasoning about agent programs to include primitives for beliefs and goals the agent, e.g., where Bp means that the agent believes that p, and Gp means that the agent has a goal that p.

The next natural question is, what should the semantics of these operators be? For example, should the belief operator satisfy the KD45 properties? In our view, it is critical that the properties of the agent's beliefs and goals should be grounded in the computation of the agent (in the sense of [234]). If the agent implements a full classical reasoner (perhaps in a restricted logic), then we can formalise its beliefs as closed under classical inference. However if the agent's implementation simply matches belief literals against a database of believed propositions without any additional logical reasoning, we should not model its beliefs as closed under classical consequence.

In this paper, we present an approach to specification and verification which is tailored to the requirements of BDI-based agent programming languages [64]. Our approach is grounded in the computation of the agent and admits an automated verification procedure based on theorem proving. The use of theorem proving rather than model checking is motivated by the current state of the art regarding available verification frameworks and tools for PDL. In particular, to the best of our knowledge there is no model checking framework for PDL, while theorem proving techniques for this logic are readily available [246, 386]. We develop our approach in the context of SimpleAPL, a simplified version of the logic-based agent programming languages 3APL [64, 128] and 2APL [122, 127]. We present a sound and complete variant of PDL [173] for SimpleAPL which allows the specification of safety and liveness properties of SimpleAPL programs. Our approach allows us to capture the agent's execution strategy in the logic, and we prove a correspondence between the operational semantics of SimpleAPL and the models of the logic for two example execution strategies. Finally, we show how to translate agent programs written in SimpleAPL into expressions of the logic, and give an example in which we verify correctness properties of a simple agent program using the PDL theorem prover MSPASS [246]. While we focus on APL-like languages and consider only single agent programs, our approach can be generalised to other BDI-based agent programming languages and the verification of multi-agent systems.

1.2 An Agent Programming Language

In this section we present the syntax and semantics of SimpleAPL, a simplified version of logic based agent-oriented programming languages 3APL [64, 128] and 2APL [122, 127]. SimpleAPL contains the core features of 3APL and 2APL and allows the implementation of agents with beliefs, goals, actions, plans, and planning rules. The main features of 3APL/2APL not present in SimpleAPL are a first order language for beliefs and goals[1], a richer set of actions (e.g., abstract plans, communication actions) and a richer set of rule types (e.g., rules for revising plans and goals and for processing events).

1.2.1 SimpleAPL

Beliefs and Goals The *beliefs* of an agent represent its information about its environment, while its *goals* represent situations the agent wants to realize (not necessary all at once). The agent's beliefs are represented by a set of positive literals

[1] In 3APL and 2APL, an agent's beliefs are implemented as a set of first-order Horn clauses and an agent's goals are implemented as a set of conjunctions of ground atoms.

and its goals by a set of arbitrary literals. The initial beliefs and goals of an agent are specified by its program. For example, a simple vacuum cleaner agent might initially believe that it is in room 1 and its battery is charged:

```
Beliefs: room1, battery
```

and may initially want to achieve a situation in which both room 1 and room 2 are clean:

```
Goals: clean1, clean2
```

The beliefs and goals of an agent are related to each other: if an agent believes p, then it will not pursue p as a goal, and if an agent does not believe that p, it will not have $-p$ as a goal.

Basic Actions Basic actions specify the capabilities an agent can use to achieve its goals. There are three types of basic action: those that update the agent's beliefs and those which test its beliefs and goals. A *belief test action* tests whether a boolean belief expression is entailed by the agent's beliefs, i.e., it tests whether the agent has a certain belief. A *goal test action* tests whether a boolean goal expression is entailed by the agent's goals, i.e., it tests whether the agent has a certain goal. *Belief update actions* change the beliefs of the agent. A belief update action is specified in terms of its pre- and postconditions (which are sets of literals), and can be executed if one of its pre-conditions is entailed by the agent's current beliefs. Executing the action updates the agent's beliefs to make the corresponding postcondition entailed by the agent's belief base. While the belief base of the agent contains only positive literals, belief and goal expressions appearing in belief and goal test actions can be complex, and the pre- and postconditions of belief update actions may contain negative literals. We will define the notion of 'entailed' formally below. Informally, a pre-condition of an action is entailed by the agent's belief base if all positive literals in the precondition are contained in the agent's belief base, and for every negative literal $-p$ in the precondition, p is not in the belief base (i.e., we use entailment under the closed world assumption). After executing a belief update action, all positive literals in the corresponding postcondition are added to the belief base, and for every negative literal $-p$ in the postcondition, p is removed from the agent's belief base. For example, the following belief update specifications:

```
BeliefUpdates:
  {room1}           moveR  {-room1, room2}
  {room1, battery}  suck   {clean1, -battery}
  {room2, battery}  suck   {clean2, -battery}
  {room2}           moveL  {-room2, room1}
  {-battery}        charge {battery}
```

can be read as "if the agent is in room 1 and moves right, it ends up in room 2", and "if the agent is in room 1 and its battery is charged, it can perform a 'suck' action, after which room 1 is clean and its battery is discharged". Note that performing a 'suck' action in a different state, e.g., in the state where the agent is in room 2,

has a different result. Belief update actions are assumed to be deterministic, i.e., the pre-conditions of an action are assumed to be mutually exclusive.

Updating the agent's beliefs may result in achievement of one or more of the agent's goals. Goals which are achieved by the postcondition of an action are dropped. For example, if the agent has a goal to clean room 1, executing a 'suck' action in room 1 will cause it to drop the goal. For simplicity, we assume that the agent's beliefs about its environment are always correct and its actions in the environment are always successful, so the agent's beliefs describe the state of the real world. This assumption can be relaxed in a straightforward way by including the state of the environment in the models.

Plans In order to achieve its goals, an agent adopts *plans*. A plan consists of basic actions composed by sequence, conditional choice and conditional iteration operators. The sequence operator `;` takes two plans as arguments and indicates that the first plan should be performed before the second plan. The conditional choice and conditional iteration operators allow branching and looping and generate plans of the form `if` ϕ `then` $\{\pi_1\}$ `else` $\{\pi_2\}$ and `while` ϕ `do` $\{\pi\}$ respectively. The condition ϕ is evaluated with respect to the agent's current beliefs. For example, the plan:

```
if room1 then {suck} else {moveL; suck}
```

causes the agent to clean room 1 if it's currently in room 1, otherwise it first moves to room 1 and then cleans it.

Planning Goal Rules *Planning goal rules* are used by the agent to select a plan based on its current goals and beliefs. A planning goal rule consists of three parts: an (optional) goal query, a belief query, and a plan. The goal query specifies which goal(s) the plan achieves, and the belief query characterises the situation(s) in which it could be a good idea to execute the plan. Firing a planning goal rule causes the agent to adopt the specified plan. For example, the planning goal rule:

```
clean2 <- battery |
  if room2 then {suck} else {moveR; suck}
```

states that "if the agent's goal is to clean room 2 and its battery is charged, then the specified plan may be used to clean the room". Note that an agent can generate a plan based only on its current beliefs as the goal query is optional. This allows the implementation of *reactive* agents (agents without any goals). For example, the reactive rule:

```
<- -battery |
  if room2 then {charge} else {moveR; charge}
```

states "if the battery is low, the specified plan may be used to charge it". For simplicity, we assume that agents do not have initial plans, i.e., plans can only be generated during the agent's execution by planning goal rules.

1.2.2 SimpleAPL syntax

The syntax of SimpleAPL is given below in EBNF notation. We assume a set of belief update actions and a set of propositions, and use ⟨*aliteral*⟩ to denote the name of a belief update action and ⟨*literal*⟩ (⟨*pliteral*⟩) to denote belief and goal literals (positive literals).

⟨*APL_Prog*⟩ ::= `"BeliefUpdates:"` ⟨*updatespecs*⟩
| `"Beliefs:"` ⟨*pliterals*⟩
| `"Goals":` ⟨*literals*⟩
| `"PG rules:"` ⟨*pgrules*⟩
⟨*updatespecs*⟩ ::= [⟨*updatespec*⟩ (`","` ⟨*updatespec*⟩)*]
⟨*updatespec*⟩ ::= `"{"` ⟨*literals*⟩ `"}"` ⟨*aliteral*⟩ `"{"`⟨*literals*⟩`"}"`
⟨*pliterals*⟩ ::= [⟨*pliteral*⟩ (`","` ⟨*pliteral*⟩)*]
⟨*literals*⟩ ::= [⟨*literal*⟩ (`","` ⟨*literal*⟩)*]
⟨*plan*⟩ ::= ⟨*baction*⟩ | ⟨*seqplan*⟩ | ⟨*ifplan*⟩ | ⟨*whileplan*⟩
⟨*baction*⟩ ::= ⟨*aliteral*⟩ | ⟨*testbelief*⟩ | ⟨*testgoal*⟩
⟨*testbelief*⟩ ::= ⟨*bquery*⟩ `"?"`
⟨*testgoal*⟩ ::= ⟨*gquery*⟩ `"!"`
⟨*bquery*⟩ ::= ⟨*literal*⟩ | ⟨*bquery*⟩ `"and"` ⟨*bquery*⟩ | ⟨*bquery*⟩ `"or"` ⟨*bquery*⟩
⟨*gquery*⟩ ::= ⟨*literal*⟩ | ⟨*gquery*⟩ `"or"` ⟨*gquery*⟩
⟨*seqplan*⟩ ::= ⟨*plan*⟩ `";"` ⟨*plan*⟩
⟨*ifplan*⟩ ::= `"if"` ⟨*bquery*⟩ `"then {"` ⟨*plan*⟩ `"}"` [`"else {"` ⟨*plan*⟩ `"}"`]
⟨*whileplan*⟩ ::= `"while"` ⟨*bquery*⟩ `"do {"` ⟨*plan*⟩ `"}"`
⟨*pgrules*⟩ ::= [⟨*pgrule*⟩ (`","` ⟨*pgrule*⟩)*]
⟨*pgrule*⟩ ::= [⟨*gquery*⟩] `"<-"` ⟨*bquery*⟩ `"|"` ⟨*plan*⟩

1.3 Operational Semantics

We define the operational semantics of SimpleAPL in terms of a transition system. A transition system is a graph where the nodes are configurations of an agent program and the edges (transitions) are given by a set of transition rules. The configuration of a SimpleAPL agent program consists of the beliefs, goals and plan(s) of the agent. Each transition corresponds to a single computation step. Which transitions are possible in a configuration depends on the agent's execution strategy. Many execution strategies are possible and we do not have space here to describe them all in detail. Below we give two versions of the operational semantics, one for an agent which executes a single plan to completion before choosing another plan (non-interleaved execution), and another for an execution strategy which interleaves the execution of multiple plans with the adoption of new plans (interleaved execution).

These strategies were chosen as representative of deliberation strategies found in the literature and in current implementations of BDI-based agent programming languages. However neither of these strategies (or any other single strategy) is clearly

best for all agent task environments. For example, the non-interleaved strategy is appropriate in situations where a sequence of actions must be executed 'atomically' in order to ensure the success of a plan. However it means that the agent is unable to respond to new goals until the plan for the current goal has been executed. Conversely, the interleaved strategy allows an agent to pursue multiple goals at the same time, e.g., allowing an agent to respond to an urgent, short-duration task while engaged in a long-term task. However it can increase the risk that actions in different plans will interfere with each other. It is therefore important that the agent developer has the freedom to choose the strategy which is most appropriate to a particular problem.

Agent Configuration An agent configuration is a 3-tuple $\langle \sigma, \gamma, \Pi \rangle$ where σ is a set of positive literals representing the agent's beliefs, γ is a set of literals representing the agent's goals, and Π is a set of plan entries representing the agent's currently executing plans.[2] In the initial configuration the agent's initial beliefs and goals are those specified by its program, and Π is empty. Executing the agent's program modifies its initial configuration in accordance with the transition rules presented below. We first present the transition rules for the non-interleaved execution strategy and then those for interleaved execution.

For the formulation of the operational semantics we need to formalize some basic assumptions. In particular, we use the notion of belief entailment based on the closed-world assumption. This notion of entailment, which we denote by $\models_{cwa}$, is defined as follows:

$$
\begin{array}{ll}
\sigma \models_{cwa} p & \Leftrightarrow p \in \sigma \\
\sigma \models_{cwa} -p & \Leftrightarrow p \notin \sigma \\
\sigma \models_{cwa} \phi \text{ and } \psi & \Leftrightarrow \sigma \models_{cwa} \phi \text{ and } \sigma \models_{cwa} \psi \\
\sigma \models_{cwa} \phi \text{ or } \psi & \Leftrightarrow \sigma \models_{cwa} \phi \text{ or } \sigma \models_{cwa} \psi \\
\sigma \models_{cwa} \{\phi_1, \ldots, \phi_n\} & \Leftrightarrow \forall 1 \leq i \leq n \ \ \sigma \models_{cwa} \phi_i
\end{array}
$$

The notion of goal entailment, denoted by $\models_g$, corresponds to a formula being classically entailed by one of the goals in the goal base γ, and is defined as follows:

$$
\begin{array}{ll}
\gamma \models_g p & \Leftrightarrow p \in \gamma \\
\gamma \models_g -p & \Leftrightarrow -p \in \gamma \\
\gamma \models_g \phi \text{ or } \psi & \Leftrightarrow \gamma \models_g \phi \text{ or } \gamma \models_g \psi
\end{array}
$$

Note that "$\gamma \models_g \phi$ and $\psi \ \Leftrightarrow \ \gamma \models_g \phi$ and $\gamma \models_g \psi$" does not hold since ϕ and ψ may be entailed by two different goals γ_1 and γ_2 from γ, but there may be no $\gamma_i \in \gamma$ which entails both ψ and ϕ. In fact, in SimpleAPL there are no non-trivial conjunctive goal queries (that is, not of the form p and p) which may be entailed by the goal base, since the goal base consists of literals.

We assume that each belief update action α has a set of preconditions $\texttt{prec}_1(\alpha)$, ..., $\texttt{prec}_k(\alpha)$. Each $\texttt{prec}_i(\alpha)$ is a finite set of belief literals, and any two pre-

[2] As an agent's planning goal rules do not change during the execution of the agent's program, we do not include them in the agent configuration.

conditions for an action α, $\texttt{prec}_i(\alpha)$ and $\texttt{prec}_j(\alpha)$ $(i \neq j)$, are mutually exclusive (i.e., for any belief base σ, if $\sigma \models_{cwa} \texttt{prec}_i(\alpha)$ and $\sigma \models_{cwa} \texttt{prec}_j(\alpha)$ then $i = j$). For each $\texttt{prec}_i(\alpha)$ there is a unique corresponding postcondition $\texttt{post}_i(\alpha)$, which is also a finite set of literals. A belief update action α can be executed if the current set of agent's beliefs σ entails some precondition $\texttt{prec}_j(\alpha)$ of α with respect to $\models_{cwa}$. This holds when all positive literals p in $\texttt{prec}_j(\alpha)$ are in σ and $\sigma \cap \{p : -p \in \texttt{prec}_j(\alpha)\} = \emptyset$. The effect of updating a set of beliefs σ with α is given by $T_j(\alpha, \sigma) = \sigma \cup (\{p : p \in \texttt{post}_j(\alpha)\} \setminus \{p : -p \in \texttt{post}_j(\alpha)\})$, (i.e., executing the belief update action α adds the positive literals in its postcondition to the agent's beliefs and removes any existing beliefs if their negations are in the postcondition).

1.3.1 Non-interleaved execution

By non-interleaved execution we mean the following execution strategy: when in a configuration with no plan, choose a planning goal rule non-deterministically, apply it, execute the resulting plan; repeat.

Belief Update Actions A belief update action α can be executed if one of its preconditions is entailed by the agent's beliefs, i,e., $\sigma \models_{cwa} \phi$. Executing the action adds the literals in the corresponding postcondition to the agent's beliefs and removes any existing beliefs which are inconsistent with the postcondition, and causes the agent to drop any goals it believes to be achieved as a result of the update.

$$(1) \quad \frac{\sigma \models_{cwa} \texttt{prec}_j(\alpha) \qquad T_j(\alpha, \sigma) = \sigma'}{\langle \sigma, \gamma, \{\alpha; \pi\} \rangle \longrightarrow \langle \sigma', \gamma', \{\pi\} \rangle}$$

where $\gamma' = \gamma \setminus (\{p : p \in \sigma'\} \cup \{-p : p \notin \sigma'\})$ and T_j is the function that determines the effect of a belief update action on a belief base as defined above. Note that in this and in rules (2)-(7) below, π may be empty, in which case $\alpha;$ is identical to α.[3]

Belief and Goal Test Actions A belief test action $\beta?$ can be executed if β is entailed by the agent's beliefs.

$$(2) \quad \frac{\sigma \models_{cwa} \beta}{\langle \sigma, \gamma, \{\beta?; \pi\} \rangle \longrightarrow \langle \sigma, \gamma, \{\pi\} \rangle}$$

The execution of a belief test action $\beta?$ in a configuration where β is not entailed by the agent's beliefs causes execution of the plan to block. In the case of non-interleaved execution, this causes the whole agent to block.

A goal test action $\kappa!$ can be executed if κ is entailed by the agent's goals.

$$(3) \quad \frac{\gamma \models_{g} \kappa}{\langle \sigma, \gamma, \{\kappa!; \pi\} \rangle \longrightarrow \langle \sigma, \gamma, \{\pi\} \rangle}$$

[3] This avoids introducing an additional transition rule for the sequence operator ;

Similar to the belief test action, the execution of a goal test action $\kappa!$ in a configuration where κ is not entailed by the agent's goals, blocks.

Composite Plans The following transition rules specify the effect of executing the conditional choice and conditional iteration operators, respectively.

$$(4)\quad \frac{\sigma \models_{cwa} \phi}{\langle \sigma, \gamma, \{(\texttt{if}\ \phi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2); \pi\}\rangle \longrightarrow \langle \sigma, \gamma, \{\pi_1; \pi\}\rangle}$$

$$(5)\quad \frac{\sigma \not\models_{cwa} \phi}{\langle \sigma, \gamma, \{(\texttt{if}\ \phi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2); \pi\}\rangle \longrightarrow \langle \sigma, \gamma, \{\pi_2; \pi\}\rangle}$$

$$(6)\quad \frac{\sigma \models_{cwa} \phi}{\langle \sigma, \gamma, \{(\texttt{while}\ \phi\ \texttt{do}\ \pi_1); \pi\}\rangle \longrightarrow \langle \sigma, \gamma, \{\pi_1; (\texttt{while}\ \phi\ \texttt{do}\ \pi_1); \pi\}\rangle}$$

$$(7)\quad \frac{\sigma \not\models_{cwa} \phi}{\langle \sigma, \gamma, \{(\texttt{while}\ \phi\ \texttt{do}\ \pi_1); \pi\}\rangle \longrightarrow \langle \sigma, \gamma, \{\pi\}\rangle}$$

Planning Goal Rules A planning goal rule $\kappa \leftarrow \beta|\pi$ can be applied if κ is entailed by the agent's goals and β is entailed by the agent's beliefs. Applying the rule adds π to the agent's plans.

$$(8)\quad \frac{\gamma \models_g \kappa \quad \sigma \models_{cwa} \beta}{\langle \sigma, \gamma, \{\}\rangle \longrightarrow \langle \sigma, \gamma, \{\pi\}\rangle}$$

1.3.2 Interleaved execution

By interleaved execution we mean the following execution strategy: either apply a planning goal rule, or execute the first step in any of the current plans; repeat. Interleaved execution strategies are characteristic of many 'event-driven' agent programming languages such as AgentSpeak(L) [357] and its derivatives [75], where the agent may adopt a new intention at each processing cycle and can pursue multiple intentions in parallel. To simplify the presentation of the operational semantics of the interleaved strategy, we associate a unique name with each planning goal rule $r_i = \kappa_i \leftarrow \beta_i|\pi_i$, and add to each plan entry in the plan base the name of the planning goal rule whose application generated the plan entry, (i.e., an entry $r_i : \pi$ in the plan base indicates that the plan π was generated by applying the planning goal rule $r_i : \kappa_i \leftarrow \beta_i \mid \pi_i$). Note that, in a particular configuration, the actual plan π in the plan base may be different from the π_i generated by applying the planning goal rule r_i if some prefix of π_i has already been executed.

The transitions for an interleaved execution strategy are:

Belief updates Similar to transition rule (1), the following rule specifies the execution of belief update action in a configuration where the plan base can contain more than one plan entry.

$$(1^i)\quad \frac{r_i : \alpha;\pi \in \Pi \quad \sigma \models_{cwa} \mathtt{prec}_j(\alpha) \qquad T_j(\alpha,\sigma) = \sigma'}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma',\gamma',(\Pi \setminus \{r_i : \alpha;\pi\}) \cup \{r_i : \pi\}\rangle}$$

where $\gamma' = \gamma \setminus \sigma'$. We stipulate that $\Pi \cup \{r_i : \ \} = \Pi$. As before, π may be empty.

Belief and goal tests Again, similar to transition rules (2) and (3), the following rules specify the execution of belief and goal test actions in a configuration where the plan base can contain more than one plan entry.

$$(2^i)\quad \frac{r_i : \beta?;\pi \in \Pi \quad \sigma \models_{cwa} \beta}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma,\gamma,(\Pi \setminus \{r_i : \beta?;\pi\}) \cup \{r_i : \pi\}\rangle}$$

$$(3^i)\quad \frac{r_i : \kappa!;\pi \in \Pi \quad \gamma \models_{g} \kappa}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma,\gamma,(\Pi \setminus \{r_i : \kappa!;\pi\}) \cup \{r_i : \pi\}\rangle}$$

Composite plans The following transition rules specify the effect of executing the conditional choice and conditional iteration operators, respectively.

$$(4^i)\quad \frac{r_i : (\mathtt{if}\ \phi\ \mathtt{then}\ \pi_1\ \mathtt{else}\ \pi_2);\pi \in \Pi \quad \sigma \models_{cwa} \phi}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma,\gamma,\Pi' \cup \{r_i : \pi_1;\pi\}\rangle}$$

$$(5^i)\quad \frac{r_i : (\mathtt{if}\ \phi\ \mathtt{then}\ \pi_1\ \mathtt{else}\ \pi_2);\pi \in \Pi \quad \sigma \not\models_{cwa} \phi}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma,\gamma,\Pi' \cup \{r_i : \pi_2;\pi\}\rangle}$$

where $\Pi' = \Pi \setminus \{r_i : (\mathtt{if}\ \phi\ \mathtt{then}\ \pi_1\ \mathtt{else}\ \pi_2);\pi\}$.

$$(6^i)\quad \frac{r_i : (\mathtt{while}\ \phi\ \mathtt{do}\ \pi_1);\pi \in \Pi \quad \sigma \models_{cwa} \phi}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma,\gamma,\Pi' \cup \{r_i : (\pi_1;\mathtt{while}\ \phi\ \mathtt{do}\ \pi_1);\pi\}\rangle}$$

$$(7^i)\quad \frac{r_i : (\mathtt{while}\ \phi\ \mathtt{do}\ \pi_1);\pi \in \Pi \quad \sigma \not\models_{cwa} \phi}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma,\gamma,\Pi' \cup \{r_i : \pi\}\rangle}$$

where $\Pi' = \Pi \setminus \{r_i : (\mathtt{while}\ \phi\ \mathtt{do}\ \pi_1);\pi\}$.

Planning goal rules A planning goal rule $r_i = \kappa_i \leftarrow \beta_i | \pi_i$ can be applied if κ_i is entailed by the agent's goals and β_i is entailed by the agent's beliefs, and provided that the plan base does not already contain a (possibly partially executed) plan generated by applying r_i. Applying the rule r_i adds π_i to the agent's plans.

$$(8^i)\quad \frac{\gamma \models_{g} \kappa_i \quad \sigma \models_{cwa} \beta_i \quad r_i : \pi \notin \Pi}{\langle \sigma,\gamma,\Pi\rangle \longrightarrow \langle \sigma,\gamma,\Pi \cup \{r_i : \pi_i\}\rangle}$$

The transition system TS for the agent's program is generated by the initial configuration c_0 if it consists of c_0 and all configurations which can be reached by applying the above mentioned transition rules. Recall that the initial configuration always has an empty plan base.

1.4 Logic

In this section, we introduce a logic which allows us to specify properties of SimpleAPL agent programs.

We begin by defining transition systems which capture the *capabilities* of agents as specified by their belief update actions. These transition systems are more general than both versions of the operational semantics presented above, in that they do not describe a particular agent program or execution strategy, but all possible basic transitions between the possible belief and goal states of an agent. We then show how to interpret a variant of Propositional Dynamic Logic (PDL) with belief and goal operators in this semantics, and give a sound and complete axiom system for the logic. In section 1.4.4 we show how the beliefs, goals and plans of an agent can be translated into our logic.

1.4.1 Preliminary

The models of the logic are defined relative to an agent program with a set of planning goal rules Λ and a set of pre- and postconditions for all belief update actions $\mathbf{C}$. Let P denote the set of propositional variables occurring in Λ. A state s corresponds to a pair $\langle \sigma, \gamma \rangle$, where:

- $\sigma \subseteq P$ is a set of beliefs, and
- γ is a set of goals $\{(\text{–})u_1, \ldots, (\text{–})u_n : u_i \in P\}$; no goal in γ should be entailed (with respect to $\models_{cwa}$) by σ.

We model states as points, and beliefs and goals are assigned to a state by two assignments, V_b and V_g. Let the set of belief update actions be $\mathtt{Ac} = \{\alpha_1, \ldots, \alpha_m\}$. Executing an action α_i in different configurations may give different results so that for each $\alpha_i \in \mathtt{Ac}$ we have an associated set of pre- and postcondition pairs $\{(\mathtt{prec}_1, \mathtt{post}_1), \ldots, (\mathtt{prec}_k, \mathtt{post}_k)\}$ denoted by $C(\alpha_i)$. We assume that $C(\alpha_i)$ is finite, that different preconditions are mutually exclusive, and that each precondition has exactly one associated postcondition. If $\sigma \models_{cwa} \mathtt{prec}_j(\alpha)$ and $T_j(\alpha, \sigma) = \sigma'$, then in the models of our logic there will be a transition R_α from a state $s = (\sigma, \gamma)$ to a state $s' = (\sigma', \gamma')$ where $\gamma' = \gamma \setminus (\{p : p \in \sigma'\} \cup \{\text{–}p : p \notin \sigma'\})$.

1.4.2 Language

Assume that we can make PDL program expressions ρ out of belief update actions $\alpha_i \in \mathtt{Ac}$ by using sequential composition ;, test on formulas ?, union $\cup$, and finite

iteration $*$. The formulas on which we can test are any formulas of the language L defined below, although to express SimpleAPL plans we only need tests on beliefs and goals. Let Υ be the set of program expressions constructed in this way.

The language L for talking about the agent's beliefs, goals and plans is the language of PDL extended with a belief operator B and a goal operator G. A formula of L is defined as follows: if $p \in P$, then Bp and $G(\text{–})p$ are formulas; if ρ is a program expression and ϕ a formula, then $\langle\rho\rangle\phi$ and $[\rho]\phi$ are formulas; and L is closed under the usual boolean connectives. In the following, we will refer to the sublanguage of L which does not contain program modalities $\langle\rho\rangle$ and $[\rho]$ as L_0.

1.4.3 Semantics

A model for L is a structure $M = (S, \{R_\rho : \rho \in \Upsilon\}, V)$, where

- S is a set of states.
- $V = (V_b, V_g)$ is the evaluation function consisting of belief and goal valuation functions V_b and V_g; each state s can be identified with a pair (σ, γ), where $V_b(s) = \sigma$ and $V_g(s) = \gamma$.
- We define R_ρ for $\rho \in \Upsilon$ inductively by the following clauses:
  - R_α, for each belief update action $\alpha \in \mathtt{Ac}$, is a relation on S such that for any $s, s' \in S$ we have that $R_\alpha(s, s')$ iff for some $(\mathtt{prec}_j, \mathtt{post}_j) \in C(\alpha)$, $T_j(\alpha, V_b(s)) = V_b(s')$ and $V_g(s') = V_g(s) \setminus (\{p : p \in V_b(s')\} \cup \{\text{–}p : p \notin V_b(s')\}$. Note that this implies two things: first, an α transition can only originate in a state s which satisfies one of the preconditions for α; second, since pre-conditions are mutually exclusive, every such s satisfies exactly one pre-condition, and all α-successors of s satisfy the matching post-condition.
  - $R_{\rho_1;\rho_2} = R_{\rho_1} \circ R_{\rho_2} = \{(s_1, s_2) \ : \ s_1, s_2 \in S \ , \ \exists s_3 \in S\,(R_{\rho_1}(s_1, s_3) \wedge R_{\rho_2}(s_3, s_2))\}$
  - $R_{\phi?} = \{(s, s) \ : \ M, s \models \phi\}$, for each formula $\phi \in L$
  - $R_{\rho_1 \cup \rho_2} = R_{\rho_1} \ \cup \ R_{\rho_2}$
  - $R_{\rho^*} = (R_\rho)^*$, the reflexive transitive closure of R_ρ.

The relation $\models$ of a formula being true in a state of a model is defined inductively as follows:

- $M, s \models Bp$ iff $p \in V_b(s)$
- $M, s \models G(\text{–})p$ iff $(\text{–})p \in V_g(s)$
- $M, s \models \neg\phi$ iff $M, s \not\models \phi$
- $M, s \models \phi \wedge \psi$ iff $M, s \models \phi$ and $M, s \models \psi$
- $M, s \models \langle\rho\rangle\phi$ iff there exists a $s' \in S$ such that $R_\rho(s, s')$ and $M, s' \models \phi$.
- $M, s \models [\rho]\phi$ iff for all $s' \in S$ such that $R_\rho(s, s')$ we have that $M, s' \models \phi$.

Let the class of transition systems defined above be denoted $\mathbf{M_C}$ (note that $\mathbf{M}$ is parameterised by the set $\mathbf{C}$ of pre- and postconditions of belief update actions).

1.4.4 Axiomatisation

The beliefs, goals and plans of agent programs can be translated into PDL expressions as follows.

- Translation of belief formulas: let $p \in P$ and ϕ, ψ be belief query expressions (i.e., $\langle bquery \rangle$) of SimpleAPL
  - $f_b(p) = Bp$
  - $f_b(-p) = \neg Bp$
  - $f_b(\phi \texttt{ and } \psi) = f_b(\phi) \wedge f_b(\psi)$
  - $f_b(\phi \texttt{ or } \psi) = f_b(\phi) \vee f_b(\psi)$

  Observe that negative queries are translated using the closed world assumption: an agent is assumed to believe that p is false if it does not have p in its belief base.
- Translation of goal formulas:
  - $f_g(p) = Gp$
  - $f_g(-p) = G-p$
  - $f_g(\phi \texttt{ or } \psi) = f_g(\phi) \vee f_g(\psi)$
- Translation of plan expressions: let α_i be a belief update action, ϕ and ψ be belief and goal query expressions, and π, π_1, π_2 be plan expressions (i.e., $\langle plan \rangle$s) of SimpleAPL
  - $f_p(\alpha_i) = \alpha_i$
  - $f_p(\phi?) = f_b(\phi)?$
  - $f_p(\psi!) = f_g(\psi)?$
  - $f_p(\pi_1; \pi_2) = f_p(\pi_1); f_p(\pi_2)$
  - $f_p(\texttt{if } \phi \texttt{ then } \pi_1 \texttt{ else } \pi_2) = (f_b(\phi)?; f_p(\pi_1)) \cup (\neg f_b(\phi)?; f_p(\pi_2))$
  - $f_p(\texttt{while } \phi \texttt{ do } \pi) = (f_b(\phi)?; f_p(\pi))^*; \neg f_b(\phi)?$

Proposition 1.1. *For all states $s = (\sigma, \gamma)$ and for all belief formulae β and goal formulae κ we have that:*

1. $M, s \models f_b(\beta) \Leftrightarrow \sigma \models_{cwa} \beta$
2. $M, s \models f_g(\kappa) \Leftrightarrow \gamma \models_g \kappa$

Proof. We prove these two propositions by induction on the complexity of formulas β and κ, respectively.

1. $M, s \models f_b(\beta) \Leftrightarrow \sigma \models_{cwa} \beta$

- *Base case*: Let $\beta = p$ or $\beta = -p$. Then, we have:

  $M, s \models f_b(p) \Leftrightarrow M, s \models Bp \Leftrightarrow p \in V_b(s) \Leftrightarrow \sigma \models_{cwa} p$.

  $M, s \models f_b(-p) \Leftrightarrow M, s \models \neg Bp \Leftrightarrow p \notin V_b(s) \Leftrightarrow \sigma \models_{cwa} -p$.

- *Inductive case*: Let $\beta = \beta_1$ and β_2. Then, $M, s \models f_b(\beta_1 \text{ and } \beta_2) \Leftrightarrow M, s \models f_b(\beta_1)$ and $M, s \models f_b(\beta_2) \Leftrightarrow$ (by the inductive hypothesis) $\sigma \models_{cwa} \beta_1$ and $\sigma \models_{cwa} \beta_2 \Leftrightarrow$ (by the definition of $\models_{cwa}$) $\sigma \models_{cwa} \beta_1$ and β_2. The case for $\beta = \beta_1$ or β_2 similarly follows from the inductive hypothesis and $\sigma \models_{cwa} \beta_1$ or $\sigma \models_{cwa} \beta_2$ if and only if $\sigma \models_{cwa} \beta_1$ or β_2.

2. $M, s \models f_g(\kappa) \Leftrightarrow \gamma \models_g \kappa$

- *Base case*: $\kappa = (-)p$
  $M, s \models f_g((-)p) \Leftrightarrow M, s \models G(-)p \Leftrightarrow (-)p \in V_g(s) \Leftrightarrow \sigma \models_g p$.
- *Inductive case:* $\kappa = \kappa_1$ or κ_2
  $M, s \models f_g(\kappa_1 \text{ or } \kappa_2) \Leftrightarrow M, s \models f_g(\kappa_1) \vee f_g(\kappa_2) \Leftrightarrow M, s \models f_g(\kappa_1)$ or $M, s \models f_g(\kappa_2) \Leftrightarrow$ (by the inductive hypothesis) $\gamma \models_g \kappa_1$ or $\gamma \models_g \kappa_2 \Leftrightarrow$ (by the definition of $\models_g \gamma \models_g \kappa_1$ or κ_2.

Note that for every pre- and postcondition pair $(\mathtt{prec}_j, \mathtt{post}_j)$ we can describe states satisfying $\mathtt{prec}_j$ and states satisfying $\mathtt{post}_j$ by formulas of L. More formally, we define a formula $f_b(X)$ corresponding to a pre- or postcondition X as follows: $f_b(\{\phi_1, \ldots, \phi_n\}) = f_b(\phi_1) \wedge \ldots \wedge f_b(\phi_n)$. This allows us to axiomatise pre- and postconditions of belief update actions.

To axiomatise the set of models defined above relative to **C** we need:

CL classical propositional logic

PDL axioms of PDL (see, e.g., [210])

A1 beliefs are not goals (positive): $Bp \rightarrow \neg Gp$

A2 beliefs are not goals (negative): $G-p \rightarrow Bp$

A3 for every belief update action α_i and every pair of pre- and postconditions $(\mathtt{prec}_j, \mathtt{post}_j)$ in $C(\alpha_i)$ and formula Φ not containing any propositional variables occurring in $\mathtt{post}_j$:

$$f_b(\mathtt{prec}_j) \wedge \Phi \rightarrow [\alpha_i](f_b(\mathtt{post_j}) \wedge \Phi).$$

This is essentially a frame axiom for belief update actions.

A4 for every belief update action α_i where all possible preconditions in $C(\alpha_i)$ are $\mathtt{prec}_1, \ldots, \mathtt{prec}_k$:

$$\neg f_b(\mathtt{prec}_1) \wedge \ldots \wedge \neg f_b(\mathtt{prec}_k) \rightarrow \neg\langle\alpha_i\rangle\top$$

where $\top$ is a tautology. This axiom ensures that belief update actions cannot be performed in states that do not satisfy any of its preconditions.

A5 for every belief update action α_i and every precondition $\texttt{prec}_j$ in $C(\alpha_i)$, $f_b(\texttt{prec}_j) \rightarrow \langle \alpha_i \rangle \top$. This axiom ensures that belief update actions can be performed successfully when one of their preconditions holds.

Let us call the axiom system above $\mathbf{Ax_C}$ where, as before, $\mathbf{C}$ is the set of pre- and postconditions of basic actions.

Theorem 1.1. $\mathbf{Ax_C}$ *is sound and (weakly) complete for the class of regular models* $\mathbf{M_C}$.

Proof. Since our logic includes PDL, we cannot prove strong completeness (for every set of formulas Γ and formula ϕ, if $\Gamma \models \phi$ then $\Gamma \vdash \phi$) because PDL is not compact. Instead, we can prove weak completeness: every valid formula ϕ is derivable ($\models \phi \ \Rightarrow \vdash \phi$).

The proof of soundness is by straightforward induction on the length of a derivation. All axioms are clearly sound, and the inference rules are standard.

The proof of completeness is standard as far as the PDL part is concerned, see for example [47]. Take a consistent formula ϕ; we are going to build a finite satisfying model $M \in \mathbf{M_C}$ for ϕ.

We define the closure, $CL(\Sigma)$ of a set of formulas of our language based on the usual definition of the Fischer-Ladner closure under single negations of Σ. However we assume a special definition of subformula closure under which we do not permit the inclusion of propositional variables, e.g., if $Bp \in \Sigma$, then we do not allow p in the subformula closure of Σ, since we do not have bare propositional variables in our language. We also have an extra condition that if an action α occurs in ϕ, then $CL(\phi)$ contains $f_b(\psi)$ for all pre- and postconditions ψ for α.

The states of the satisfying model M will be all maximal consistent subsets of $CL(\phi)$. Let A, B be such maximal consistent sets, and α be a basic action. We define $R_\alpha(A, B)$ to hold if and only if the conjunction of formulas in A, $\wedge A$, is consistent with $\langle \alpha \rangle \wedge B$ (conjunction of formulas in B preceded by $\langle \alpha \rangle$).

The relations corresponding to complex programs ρ are defined inductively on top of the relations corresponding to basic actions using unions, compositions, and reflexive transitive closures, as is the case with regular models.

We define the assignment V in an obvious way:

- $p \in V_b(A)$ iff $Bp \in A$, where $Bp \in CL(\phi)$;
- $(\text{–})p \in V_g(A)$ iff $G(\text{–})p \in A$, where $G(\text{–})p \in CL(\phi)$.

The truth lemma follows easily on the basis of the PDL completeness proof given in [47]; so we have that for every $\psi \in CL(\phi)$,

$$\psi \in A \ \Leftrightarrow \ M, A \models \psi$$

Since our formula ϕ is consistent, it belongs to at least one maximal consistent set A, so it is satisfied in some state in M.

Clearly, beliefs and goals are disjoint because the states are consistent with respect to the axioms **A1** and **A2**.

All that remains to show is that this model M also satisfies the pre- and postconditions of the actions which occur in the formula ϕ: an action α is not applicable if none of its preconditions are satisfied, and if it is applied in a state s where one of its preconditions holds (recall that the preconditions are disjoint), then the corresponding postcondition holds in all states s' accessible from s by α.

First, consider an action α and state A such that A does not satisfy any of the preconditions of α. Then, by axiom **A4**, $\wedge A$ implies $[\alpha]\bot$, so there is no maximal consistent set B such that $\wedge A \wedge \langle\alpha\rangle \wedge B$ is consistent, so there is no α-transition from A. Now suppose that A satisfies one of the preconditions $\mathtt{prec}_j$ of α. Then $f_b(\mathtt{prec}_j) \in A$ (recall that $CL(\phi)$ contains $f_b(\mathtt{prec}_j)$, so we can use the truth lemma) and $\wedge A$ implies $[\alpha] f_b(\mathtt{post}_j)$ by **A3**. For any B such that $\wedge A \wedge \langle\alpha\rangle \wedge B$ is consistent, B has to contain $f_b(\mathtt{post}_j)$ since $f_b(\mathtt{post}_j)$ is in $CL(\phi)$ and $\wedge A \wedge \langle\alpha\rangle \neg f_b(\mathtt{post}_j)$ is not consistent, and such a successor B exists by **A5**. So every α-successor of A satisfies the postcondition. Similarly, we can show that for every literal q in A (in $CL(\phi)$), which does not occur in the postcondition $f_b(\mathtt{post}_j)$, it is not consistent to assume that its value has changed in a state accessible by α (e.g. $Bq \wedge \langle\alpha\rangle \neg Bq$ is inconsistent), because of **A3**; so all literals in the state A which do not occur in the postcondition $f_b(\mathtt{post}_j)$ do not change their value in the state accessible by α. Note that all other literals which do not occur in $CL(\phi)$ and by construction do not occur in the postconditions of any action occurring in $CL(\phi)$ are always absent in all states, so their value trivially does not change. □

1.5 Verification

In this section we show how to define exactly the set of paths in the transition system generated by the operational semantics which correspond to a PDL program expression. This allows us to verify properties of agent programs, such as 'all executions of a given program result in a state satisfying property ϕ'. More precisely, we would like to express that, given the initial beliefs and goals of the agent, the application of its planning goal rules and the execution of the resulting plans reach states in which the agent has certain beliefs and goals.

We distinguish two types of properties of agent programs: safety properties and liveness properties. Let $\phi \in L_0$ denote the initial beliefs and goals of an agent and $\psi \in L_0$ denote states in which certain beliefs and goals hold (i.e., ϕ, ψ are formulas of L_0 containing only Bp and $G(\neg)q$ atoms). The general form of safety and liveness properties is then: $\phi \rightarrow [\ \xi(\Lambda)\ ]\psi$ and $\phi \rightarrow \langle\xi(\Lambda)\rangle\psi$, respectively, where $\xi(\Lambda)$ describes the execution of the agent's program with a set of planning goal rules Λ.

1.5.1 Expressing the non-interleaved strategy

The application of a set of planning goal rules $\Lambda = \{r_i | r_i = \kappa_i \leftarrow \beta_i | \pi_i\}$ for an agent with a non-interleaved execution strategy is translated as follows:

$$\xi(\Lambda) = (\bigcup_{r_i \in \Lambda} (f_g(\kappa_i) \wedge f_b(\beta_i))?; f_p(\pi_i))^+$$

where $^+$ is the strict transitive closure operator: $\rho^+ = \rho;\rho^*$. This states that each planning goal rule is be applied zero or more times (but at least one planning goal rule will be applied).

Using this definition of $\xi(\Lambda)$, the general schema of safety and liveness properties for an agent with an interleaved execution strategy are then:

$\phi \rightarrow [\ (\bigcup_{r_i \in \Lambda} (f_g(\kappa_i) \wedge f_b(\beta_i))?; f_p(\pi_i))^+\ ]\psi$ for safety properties; and
$\phi \rightarrow \langle\ (\bigcup_{r_i \in \Lambda} (f_g(\kappa_i) \wedge f_b(\beta_i))?; f_p(\pi_i))^+\ \rangle\psi$ for liveness properties.

Below we show that the translation above is faithful, namely the PDL program expression which is the translation of the agent's program corresponds to the set of paths in the transition system generated by the operational semantics for that agent program. But first we need a few extra definitions.

A model generated by a state s_0 consists of all possible states which can be recursively reached from s_0 by following the basic relations. A state s and a configuration $c = \langle \sigma, \gamma, \Pi \rangle$ are matching if they have the same belief and goal bases, that is $V_b(s) = \sigma$ and $V_g(s) = \gamma$. We denote this as $s \sim c$. Let $\mathbf{C}$ be a set of pre- and postconditions of belief update actions and Λ a set of planning goal rules. Let TS be a transition system defined by the operational semantics for an agent using the non-interleaved execution strategy (all possible configurations $\langle \sigma, \gamma, \Pi \rangle$ and transitions between them, given Λ and $\mathbf{C}$) and M a model belonging to $\mathbf{M_C}$. TS and M are called matching if they are generated by c_0 and s_0, respectively, such that $s_0 \sim c_0$.

We now prove a theorem which will allow us to verify properties of reachability in TS by evaluating formulas $\langle \xi(\Lambda) \rangle \phi$ at s_0.

Theorem 1.2. *Assume that TS is a transition system defined by the operational semantics for an agent with a set of planning goal rules Λ with pre- and postconditions for basic actions $\mathbf{C}$ using a non-interleaved execution strategy, and M is a model in $\mathbf{M_C}$. Then if TS and M match, then a configuration c with an empty plan base is reachable from the initial configuration c_0 in TS iff a state s matching c is reachable from the initial state s_0 (matching c_0) along a path described by $\xi(\Lambda)$, i.e., $(s_0, s) \in R_{\xi(\Lambda)}$.*

Before proving the theorem, we need the following lemma:

Lemma 1.1. *For all $s = \langle \sigma, \gamma \rangle$, $s' = \langle \sigma', \gamma' \rangle$, and plans π and π', we have: $\langle \sigma, \gamma, \{\pi; \pi'\} \rangle \longrightarrow \langle \sigma', \gamma', \{\pi'\} \rangle$ in TS iff $R_{f_p(\pi)}(s, s')$ in M.*

Proof of Lemma 1.1. By induction on the length of π.

Basis of induction: We prove that the lemma holds for belief update actions, and belief and goal test actions. Clearly, with respect to the belief update actions α, $\langle \sigma, \gamma, \{\alpha; \pi'\}\rangle \longrightarrow \langle \sigma', \gamma', \{\pi'\}\rangle$ in TS iff $R_\alpha(s, s')$ in M, by the operational semantics rule (1) and the definition of R_α in terms of pre- and postconditions. For belief tests $\phi?$ and goal tests $\psi!$, the relations $R_{f_b(\phi)?}$ and $R_{f_g(\psi)?}$ hold for exactly the same pairs (s, s) for which belief and goal test transitions hold by rules (2) and (3). This follows from Proposition 1.

Inductive step: Assume the lemma holds for the sub-plans of π.

Let $\pi = \texttt{if}\ \phi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2$. Let us assume that $\sigma \models_{cwa} \phi$ and there is a transition from $\langle \sigma, \gamma, \{\pi; \pi'\}\rangle$ to $\langle \sigma, \gamma, \{\pi_1; \pi'\}\rangle$ by rule (4) in TS (the `else` case is similar). Then by Proposition 1, $M, s \models f_b(\phi)$, so $R_{f_b(\phi)?}(s, s)$. By the inductive hypothesis, there is a path from $\langle \sigma, \gamma, \{\pi_1; \pi'\}\rangle$ to $\langle \sigma', \gamma', \{\pi'\}\rangle$ iff $R_{f_p(\pi_1)}(s, s')$. Hence, $R_{f_b(\phi)?; f_p(\pi_1)}(s, s')$ and $R_{f_p(\pi)}(s, s')$. The other direction is similar. Assume that $R_{f_p(\pi)}(s, s')$ and $R_{f_b(\phi)?}(s, s)$ (the case of $R_{\neg f_b(\phi)?}(s, s)$ is identical). Then by Proposition 1, $\sigma \models_{cwa} \phi$ so by rule (4), there is a transition from $\langle \sigma, \gamma, \{\pi; \pi'\}\rangle$ to $\langle \sigma, \gamma, \{\pi_1; \pi'\}\rangle$ and from there by executing π_1 to $\langle \sigma', \gamma', \{\pi'\}\rangle$ (by the inductive hypothesis).

Let $\pi = \texttt{while}\ \phi\ \texttt{do}\ \pi_1$. Assume that there is a path in TS between $\langle \sigma, \gamma, \{\pi; \pi'\}\rangle$ and $\langle \sigma', \gamma', \{\pi'\}\rangle$. Note that from the rules (6) and (7) we can conclude that $\sigma' \not\models_{cwa} \phi$. By Proposition 1, $M, s' \models \neg f_b(\phi)$, so $R_{\neg f_b(\phi)}(s, s)$. Consider the path in TS; it is a sequence of configurations $\langle \sigma_1, \gamma_1, \{\pi; \pi'\}\rangle$, $\langle \sigma_2, \gamma_2, \{\pi_1; \pi; \pi'\}\rangle$, $\ldots$, $\langle \sigma_n, \gamma_n, \{\pi'\}\rangle$, where $(\sigma_1, \gamma_1) = (\sigma, \gamma)$, $(\sigma_n, \gamma_n) = (\sigma', \gamma')$ and one of the two cases holds. Either $n = 2$, so the path is of the form $\langle \sigma, \gamma, \{\pi; \pi'\}\rangle$, $\langle \sigma, \gamma, \{\pi'\}\rangle$. In this case (σ, γ) and (σ', γ') are the same (that is, $s = s'$), $\sigma \not\models_{cwa} \phi$ (rule 7) and $R_{\neg f_b(\phi)?}(s, s')$.

Or, $n > 2$, so there is a chain of configurations connected by paths corresponding to the executions of π_1. In this case, for each $i < n$ it holds that $\langle \sigma_i, \gamma_i, \{\pi_1; \pi'\}\rangle$ has a path to $\langle \sigma_{i+1}, \gamma_{i+1}, \{\pi'\}\rangle$. But then by the inductive hypothesis, $R_{f_p(\pi_1)}(s_i, s_{i+1})$ and $R_{f_p(\pi_1)^*}(s, s')$, hence $R_{f_p(\pi)}(s, s')$. The other direction is similar.

This completes the proof that all paths corresponding to an execution of a plan π in TS are described by $f_p(\pi)$ in M. □ (of Lemma)

Proof of Theorem 1.2. Observe that in the operational semantics for the non-interleaved execution strategy, any path between two configurations with an empty plan base consists of one or more cycles of executing one of the goal planning rules followed by the execution of the corresponding plan. We will prove the theorem by induction on the number of such cycles on the path between two configurations with empty plan bases, $\langle \sigma, \gamma, \{\}\rangle$ and $\langle \sigma', \gamma', \{\}\rangle$. We use the lemma above for the special case when π' is empty.

Basis of induction: the path involves one cycle. Suppose there is such a path in TS. This means that some planning goal rule $\phi \leftarrow \psi | \pi$ matched (so ϕ and ψ are true in $\langle \sigma, \gamma, \{\}\rangle$) and π was adopted, resulting in a configuration $\langle \sigma, \gamma, \{\pi\}\rangle$, and from that configuration there is a path to $\langle \sigma', \gamma', \{\}\rangle$. By the lemma, this means that

there is a corresponding path in M from (σ,γ) to (σ',γ') described by $f_p(\pi)$. Since ϕ and ψ are true in (σ,γ), there is a path from (σ,γ) to itself by $(f_g(\psi) \wedge f_b(\phi))?$, from which follows that there is a path in M from (σ,γ) to (σ',γ') described by $(f_g(\psi) \wedge f_b(\phi))?; f_p(\pi)$.

The other direction: assume that in M, there is a path from (σ,γ) to (σ',γ') described by $(f_g(\psi) \wedge f_b(\phi))?; f_p(\pi)$. We need to show that in TS, there is a path from $\langle\sigma,\gamma,\{\}\rangle$ to $\langle\sigma',\gamma',\{\}\rangle$. Since in M, there exists a transition from (σ,γ) to itself along $(f_g(\psi) \wedge f_b(\phi))?$, this means that ϕ and ψ are entailed by the agent's belief and goal base by Proposition 1. This means that the corresponding planning goal rule will be applied in $\langle\sigma,\gamma,\{\}\rangle$ resulting in adoption of π (transition to the configuration $\langle\sigma,\gamma,\{\pi\}\rangle$). By the lemma, there is a path from $\langle\sigma,\gamma,\{\pi\}\rangle$ to $\langle\sigma',\gamma',\{\}\rangle$.

Inductive step: assume that any path of length $k-1$ between two configurations with empty plan bases has a corresponding path in M described by a path in $\xi(\Lambda)$, which means that it is described by an $k-1$-long concatenation of expressions of the form $(f_g(\psi) \wedge f_b(\phi))?; f_p(\pi)$, for example $(f_g(\psi_1) \wedge f_b(\phi_1))?; f_p(\pi_1); \ldots; (f_g(\psi_{k-1}) \wedge f_b(\phi_{n-1}))?; f_p(\pi_{k-1})$. By the argument in the basis step, the last (kth) segment corresponds to a path described by $(f_g(\psi_k) \wedge f_b(\phi_k))?; f_p(\pi_k)$. Hence, the whole path is described by

$$(f_g(\psi_1) \wedge f_b(\phi_1))?; f_p(\pi_1); \ldots; (f_g(\psi_k) \wedge f_b(\phi_k))?; f_p(\pi_k),$$

which is in $\xi(\Lambda)$. □ (of Theorem)

1.5.2 Expressing the interleaved strategy

For an agent with an interleaved execution strategy, we need a version of PDL with an additional interleaving operator, $\|$ [1]. Strictly speaking, the interleaving operator does not increase the expressive power of PDL, but it makes the language more concise (every formula containing the interleaving operator has an equivalent formula without, however the size of that formula may be doubly exponential in the size of the original formula, see [1]).

Note that we are able to view regular models $M = (S, \{R_\rho : \rho \in \Upsilon\}, V)$ as models of the form $M' = (S, \tau, V)$ where $\tau(\alpha_i) \subseteq (S \times S)$ gives us the set of state transitions for α_i such that $(s,s') \in \tau(\alpha_i)$ iff $R_{\alpha_i}(s,s')$. We can extend this inductively to give us a set of *paths* $\tau(\rho) \subseteq (S \times S)^*$ in M corresponding to any PDL program expression ρ, including expressions with the interleaving operator $\rho_1 \| \rho_2$. By a path we mean a sequence $(s_1,s_2),(s_3,s_4),\ldots,(s_{n-1},s_n)$ $(n \geq 2)$ of pairs of states, where each pair is connected by an atomic action transition or a test transition. By a *legal* path we mean a path where for every even $i < n$ (the target of the transition), $s_i = s_{i+1}$ (the source of the next transition). Otherwise a path is called *illegal*. For example, $(s_1,s_2),(s_2,s_3)$ is a legal path and $(s_1,s_2),(s_3,s_4)$ where $s_2 \neq s_3$ is an illegal path.

Paths corresponding to PDL program expressions are defined as follows:

- $\tau(\phi?) = \{(s,s) : M, s \models \phi\}$
- $\tau(\rho_1 \cup \rho_2) = \{z : z \in \tau(\rho_1) \cup \tau(\rho_2)\}$
- $\tau(\rho_1;\rho_2) = \{z_1 \circ z_2 : z_1 \in \tau(\rho_1),\ z_2 \in \tau(\rho_2)\}$, where $\circ$ is concatenation of paths; here we allow illegal paths $p_1 \circ p_2$ where $p_1 = (s_0,s_1)...(s_n,s_{n+1})$ and $p_2 = (t_0,t_1)...(t_m,t_{m+1})$, with $s_{n+1} \neq t_0$.
- $\tau(\rho^*)$ is the set of all paths consisting of zero or finitely many concatenations of paths in $\tau(\rho)$.
- $\tau(\rho_1 \parallel \rho_2)$ is the set of all paths obtained by interleaving paths from $\tau(\rho_1)$ and $\tau(\rho_2)$.

The reason why we need illegal paths for PDL with the interleaving operator can be illustrated by the following example. Let $(s_1,s_2) \in \tau(\alpha_1)$ and $(s_3,s_4) \in \tau(\alpha_2)$, with $s_2 \neq s_3$). Then the illegal path $(s_1,s_2),(s_3,s_4) \in \tau(\alpha_1;\alpha_2)$. Let $(s_2,s_3) \in \tau(\alpha_3)$. Then $(s_1,s_2),(s_2,s_3),(s_3,s_4)$ is obtained by interleaving a path from $\tau(\alpha_1;\alpha_2)$ and $\tau(\alpha_3)$, and it is a legal path in $\tau(\alpha_1;\alpha_2 \parallel \alpha_3)$. Note that if the paths above are the only paths in $\tau(\alpha_1),\tau(\alpha_2)$ and $\tau(\alpha_3)$, then using an illegal path in $\tau(\alpha_1;\alpha_2)$ is the only way to define a legal interleaving in $\tau(\alpha_1;\alpha_2 \parallel \alpha_3)$.

We define the relation $\models$ of a formula being true in a state of a model as:

- $M, s \models Bp$ iff $p \in V_b(s)$
- $M, s \models G(-)p$ iff $(-)p \in V_g(s)$
- $M, s \models \neg\phi$ iff $M, s \not\models \phi$
- $M, s \models \phi \wedge \psi$ iff $M, s \models \phi$ and $M, s \models \psi$
- $M, s \models \langle\rho\rangle\phi$ iff there is a legal path in $\tau(\rho)$ starting in s which ends in a state s' such that $M, s' \models \phi$.
- $M, s \models [\rho]\phi$ iff for all legal paths $\tau(\rho)$ starting in s, the end state s' of the path satisfies ϕ: $M, s' \models \phi$.

In this extended language, we can define paths in the execution of an agent with an interleaved execution strategy and planning goal rules as

$$\xi^i(\Lambda) = \bigcup_{\Lambda' \subseteq \Lambda, \Lambda' \neq \emptyset} \|_{r_i \in \Lambda'} ((f_g(\kappa_i) \wedge f_b(\beta_i))?; f_p(\pi_i))^+$$

Theorem 1.3. *Assume that TS is a transition system defined by the operational semantics for an agent with a set of planning goal rules Λ with pre- and postconditions for basic actions $\mathbf{C}$ using an interleaved execution strategy, and M is a model in $\mathbf{M_C}$. Then if TS and M match, then a configuration c with an empty plan base is reachable from the initial configuration c_0 in TS iff a state s matching c is reachable from the initial state s_0 (matching c_0) along a path in $\tau(\xi^i(\Lambda))$.*

Proof. In order to prove the theorem, we need to show a correspondence between finite paths in TS and M. By a path in TS from c_0 to c, we will mean a legal path $(c_0, c_1), (c_1, c_2), \ldots, (c_{n-1}, c_n)$ where $c_n = c$, such that for every pair $(c_i, c_i + 1)$ on the path, there is a transition from c_i to c_{i+1} described by one of the operational semantics rules (1^i) - (8^i). By a path in M from s_0 to s we will mean a legal path $(s_0, s_1), (s_1, s_2) \ldots, (s_{n-1}, s_n)$ where $s_n = s$ such that for each step $p_i = (s_i, s_{i+1})$ on the path, s_i and s_{i+1} are in some R_α or $R_{\phi?}$ relation (in the latter case $s_i = s_{i+1}$). We will refer to α or $\phi?$ as the label of that step and denote it by $label(p_i)$. By a path in M from s_0 to s which is in $\tau(\xi^i(\Lambda))$ we will mean a path from s_0 to s where the labels on the path spell a word in $(\xi^i(\Lambda))$.

We will denote the steps on the path by $p_0, p_1, \ldots, p_{n-1}$, and refer to the first component of the pair p_i as p_i^0 and to the second component as p_i^1. If $p_i = (s_i, s_{i+1})$, then $p_i^0 = s_i$ and $p_i^1 = s_{i+1}$. Note that the same state can occur in different steps, and we want to be able to distinguish those occurrences. Since the path is legal, for all i, $p_i^1 = p_{i+1}^0$.

As an auxiliary device in the proof, we will associate with each component p_i^j ($j \in \{0, 1\}$) of each step p_i on the path in M a history $\rho(p_i^j)$ and a set of 'execution points' $E(p_i^j)$.

A history is a PDL program expression which is a concatenation of labels of the previous steps on the path. For example, consider a path $p_0 = (s_0, s_0), p_1 = (s_0, s_2)$ where $R_{\phi?}(s_0, s_0)$ and $R_\alpha(s_0, s_2)$. Then the history $\rho(p_0^0)$ is an empty string, $\rho(p_0^1) = \rho(p_1^0) = \phi?$ and the history $\rho(p_1^1) = \phi?;\alpha$. For an arbitrary point p_i^j on a path in $\tau(\xi^i(\Lambda))$, the history describes a prefix of a path in $\tau(\xi^i(\Lambda))$. Note that $\xi^i(\Lambda)$ is a union of $\|_{r_i \in \Lambda'} ((f_g(\kappa_i) \wedge f_b(\beta_i))?; f_p(\pi_i))^+$, so the history will consist of an interleaving of tests and actions which come from tracing expressions of the form $(f_g(\kappa_i) \wedge f_b(\beta_i))?; f_p(\pi_i)$ for $r_i \in \Lambda' \subseteq \Lambda$, some of them repeated several times. At the step where all the plan expressions $f_p(\pi_i)$ have been traced to their ends, the history describes a path in $\tau(\xi^i(\Lambda))$. Conversely, if the history in the last step of the path describes a path in $\tau(\xi^i(\Lambda))$, then the path is in $\tau(\xi^i(\Lambda))$.

A set of execution points $E(p_i^j)$ will contain execution points, which are PDL plan expressions of the form $f_p(\pi)$, where π is either a translation of some complete plan π_i for some $r_i \in \Lambda$, or of a suffix of such a plan. Intuitively, they correspond to a set of (partially executed) plans in the plan base corresponding to p_i^j, and are called execution points because they tell us where we are in executing those plans. We annotate p_i^j with sets of execution points using the following simple rule. When we are in p_i^0 and $E(p_i^0) = \{f_p(\pi_1), \ldots, f_p(\pi_k)\}$, then exactly one of the following three options apply. Either $label(p_i) = \alpha$, and one of the $f_p(\pi_j)$ is of the form $f_p(\alpha;\pi')$, in which case in $E(p_i^1)$, $f_p(\alpha;\pi')$ is replaced by $f_p(\pi')$. Or, $label(p_i) = \phi?$, and then one of the two possibilities apply. Either $\phi? = f_b(\beta?)$ and one of the $f_p(\pi_j)$ is of the form $f_p(\beta?;\pi')$, in which case in $E(p_i^1)$, $f_p(\beta?;\pi')$ is replaced by $f_p(\pi')$. Or, $\phi? = (f_g(\kappa_m!) \wedge f_b(\beta_m?))?$ (the test corresponding to a planning goal rule r_m) and $E((p_i^1) = E(p_i^0) \cup \{f_p(\pi_m)\}$. Essentially we take the first step α or $\phi?$ in tracing one of the expressions in $E(p_i^0)$, remove it, and append it to the history in the next state.

It is clear that if the sets of execution points along the path correspond to plan bases in the operational semantics, then the histories correspond to prefixes of paths in $\tau(\xi^i(\Lambda))$. Also, if on such a path $E(p_i^j) = \emptyset$ for $i > 0$, then $\rho(p_i^j)$ describes a path in $\tau(\xi^i(\Lambda))$.

We say that a set of execution points $E(p_i^j)$ and a plan base Π match if the execution points in $E(p_i^j)$ are the translations of plans in Π, that is $\Pi = \{\pi_1, \ldots, \pi_k\}$ and $E(p_i^j) = \{f_p(\pi_1), \ldots, f_p(\pi_k)\}$.

The idea of the proof is as follows. We show that

($TS \Rightarrow M$) For every path in TS from c_0 to a configuration c with an empty plan base, we can trace, maintaining a set of execution points and a history, a path in M from s_0 to a state s such that $s \sim c$ and the last step on the path has an empty set of execution points and a history in $\tau(\xi^i(\Lambda))$. This will show one direction of the theorem, that every path in TC has a corresponding path in M.

($M \Rightarrow TS$) States on every path in M from s_0 to s which is in $\tau(\xi^i(\Lambda))$ can be furnished with sets of execution points which correspond to plan bases of configurations on a corresponding path from c_0 to c such that $s \sim c$. This shows another direction of the theorem, that if we have a path in $\tau(\xi^i(\Lambda))$, we can find a corresponding path in TS.

To prove ($TS \Rightarrow M$), we first note that s_0 and c_0 have a matching set of execution points and plan base (empty). Then we consider a pair $s \sim c$ where the plan base in c and the set of execution points in $p_{n-1}^1 = s$ match, and show that if we can make a transition from c, then we can make a step $p_n = (s, s')$ from s, and end up with a matching state-configuration pair $s' \sim c'$ where $E(p_n^1)$ matches Π'. Note that if we match the plan base and the set of execution points at each step, and update the execution step according to the rule, then the history in guaranteed to be a prefix of a path in $\tau(\xi^i(\Lambda))$ (or a path in $\tau(\xi^i(\Lambda))$ if the set of execution points is empty).

Let us consider possible transitions from c. Either, a planning goal rule is applied, or one of the plans in Π is chosen and a suitable transition rule applied to execute its first step.

If a planning goal rule r_m is applied, then clearly the belief and goal conditions of r_m hold in c so by assumption they hold in s, hence in s, there is a transition by $R_{(f_g(\kappa_m) \wedge f_b(\beta_m))?}$ to the same s with the same belief and goal base. The transition in TS goes to c' with the same belief and goal base as c and Π' extended with π_m. In M, we make a step along the path to $s' = s$ and add $f_p(\pi_m)$ to the set of execution points $E(p_n^1)$. Clearly, Π' and $E(p_n^1)$ match.

If the transition rule corresponds to executing one of the plans π_i in Π, then we have the following cases.

- $\pi_i = \alpha; \pi$: we execute a belief update action α and transit to c'. This is possible only if in M there is an R_α transition to s' such that $s' \sim c'$. In c' in the plan base Π' we have π instead of $\alpha; \pi$. In the set of execution points for the next step $p_n^1 = s'$ we have $f_p(\pi)$ instead of $\alpha; f_p(\pi)$. Clearly, Π' and $E(p_n^1)$ match.

- $\pi_i = \beta?;\pi$ or $\pi_i = \kappa!;\pi$. A transition from c to a configuration c' where the plan base contains π instead of π_i is possible if and only if the test succeeds in σ, so by Proposition 1 if and only if in M there is a corresponding $R_{f_b(\beta)?}$ or $R_{f_g(\kappa)?}$ transition from s to itself, so $p_n = (s, s)$, $s \sim c'$, the execution point $E(p_n^1)$ contains $f_p(\pi)$ instead of $f_p(\pi_i)$, so Π' and $E(p_n^1)$ match.
- $\pi_i = (\texttt{if}\ \phi\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2);\pi$ and in $E(p_n^0)$ we have $(f_b(\phi)?; f_p(\pi_1)) \cup (\neg f_b(\phi)?; f_p(\pi_2)); f_p(\pi)$. Since $s \sim c$ either ϕ is true in both s and c or $\neg\phi$. Assume that ϕ is true (the case for ϕ false is analogous). In TS we transit to c' with $\pi_1;\pi$ in the plan base and in M we transit to s' by executing the test on the left branch of $\cup$ and replace $f_p(\pi_i)$ in the set of execution points with $f_p(\pi_1); f_p(\pi) = f_p(\pi_1;\pi)$.
- $\pi_i = \texttt{while}\ \phi\ \texttt{do}\ \pi_1;\pi$ and in $E(p_n^0)$ we have $(f_b(\phi)?; f_p(\pi_1))^*; \neg f_b(\phi)?; f_p(\pi)$. Since $s \sim c$ we have ϕ either true or false in both. If it is false then we transit in TS to c' with π in the plan base, and in M there is a $\neg f_b(\phi)?$ transition to s itself, but now we replace $(f_b(\phi)?; f_p(\pi_1))^*; \neg f_b(\phi)?; f_p(\pi)$ in the set of execution points with $f_p(\pi)$. If ϕ is true, then by the rule (6^i) we go to c' with Π' containing $\pi_1;\texttt{while}\ \phi\ \texttt{do}\ \pi_1;\pi$ and by $f_b(\phi)?$ in M we go to s' with the set of execution points containing $f_p(\pi_1); (f_b(\phi)?; f_p(\pi_1))^*; \neg f_b(\phi)?; f_p(\pi)$. Note that the new set of execution points and Π' match because $f_p(\pi_1); (f_b(\phi)?; f_p(\pi_1))^*; \neg f_b(\phi)?; f_p(\pi)$ is the same as $f_p(\pi_1;\texttt{while}\ \phi\ \texttt{do}\ \pi_1;\pi)$.

This achieves the desired correspondence in the direction from the existence of a path in the operational semantics to the existence of a corresponding path in the model; it is easy to check that the path to s' is described by its history, and that when we reach a state s corresponding to a configuration with an empty plan base, its set of execution points is also empty and the history describes a path in $\tau(\xi^i(\Lambda))$.

Let us consider the opposite direction ($M \Rightarrow TS$). Suppose we have a path in M from s_0 to s which is in $\tau(\xi^i(\Lambda))$. We need to show that there is a corresponding path in the operational semantics. For this direction, we only need to decorate each components of a step on the path in M with a set of execution points corresponding to the plan base in the matching configuration c'. (We do not need the history because we already know that the path in M is in $\tau(\xi^i(\Lambda))$.)

Clearly, in the initial state s_0, the set of execution points is empty, and $s_0 \sim c_0$. Now we assume that we have reached s and c such that $s \sim c$ and $E(p_{n-1}^1)$ and Π match, where p_{n-1} is the last step on the path and $p_{n-1}^1 = s$. Now we have to show how for the next step along the path in M to a state s' we can find a transition in TS to a configuration c' so that $s' \sim c'$ and the set of execution points in s' matches Π'.

Our task is made slightly harder by the fact that if we encounter, for example, a test transition on a path in M, we do not necessarily know whether it is a test corresponding to firing a new planning goal rule, or is an `if` or a `while` test in one of the plans (there could be several plan expressions in $E(p_n^0)$ starting with the same test $f_b(\phi?)$, for example). Similarly, if we have an α transition on the path, several plan expressions in $E(p_n^0)$ may start with an α (for example $E(p_n^0) = \{\alpha;\alpha_1,\ \alpha;\alpha_2\}$), so the question is how do we find a corresponding configuration in the operational semantics. In the example above, the plan base could be either $\Pi_1' = \{\alpha_1,\ \alpha;\alpha_2\}$ or

$\Pi'_2 = \{\alpha;\alpha_1,\ \alpha_2\}$. However, note that the path we are trying to match is in $\tau(\xi^i(\Lambda))$, so it contains transitions corresponding to a complete interleaved execution of all plans currently in the set of execution points until the end. So we can look ahead at the rest of the path and annotate ambiguous transitions with the corresponding plan indices. In the example above, if the next transition is α_1, we know that the ambiguous α belongs to the first plan; if the next transition is α_2, we annotate the ambiguous α with the index of the second plan; if the next transition is α, then there are two possible matching paths in TS, and we can pick one of them non-deterministically, as both α's have the same effect on the belief and goal base, for example annotate the first α with the index of the first plan and the second α with the index of the second plan.

Once we have indexed the ambiguous transitions, finding the corresponding path in TS can be done in exactly the same way as in the proof for $(TS \Rightarrow M)$. □

1.6 Example of using theorem proving to verify properties of an agent program

In this section we briefly illustrate how to prove properties of agents in our logic, using the vacuum cleaner agent as an example. We will use the following abbreviations: c_i for $\texttt{clean}_\texttt{i}$, r_i for $\texttt{room}_\texttt{i}$, b for `battery`, s for `suck`, c for `charge`, r for `moveR`, l for `moveL`. The agent has the following planning goal rules:

c`b1 <-` b `|` `if` r`b1` `then` $\{s\}$ `else` $\{l;\ s\}$
c`b2 <-` b `|` `if` r`b2` `then` $\{s\}$ `else` $\{r;\ s\}$
`<-` $-b$ `|` `if` r`b2` `then` $\{c\}$ `else` $\{r;\ c\}$

Under the non-interleaved execution strategy, these planning goal rules can be translated as the following PDL program expression:

$$\begin{aligned}\texttt{vac} =_{df}\ & ((Gc_1 \wedge Bb)?;(Br_1?;s) \cup (\neg Br_1?;l;s)) \cup \\ & ((Gc_2 \wedge Bb)?;(Br_2?;s) \cup (\neg Br_2?;r;s)) \cup \\ & (\neg Bb?;(Br_2?;c) \cup (\neg Br_2?;r;c))\end{aligned}$$

Given appropriate pre- and postconditions for the belief update actions in the example program (such as the pre- and postconditions of `moveR`, `moveL`, `charge` and `suck` given earlier in the paper), some of the instances of A3–A5 are:

A3r $\quad Bc_1 \wedge Br_1 \wedge \neg Bb \wedge Gc_2 \rightarrow [r](Br_2 \wedge Bc_1 \wedge \neg Bb \wedge Gc_2)$

A3s1 $\quad Gc_1 \wedge Gc_2 \wedge Br_1 \wedge Bb \rightarrow [s](Bc_1 \wedge \wedge Gc_2 \wedge Br_1 \wedge \neg Bb)$

A3s2 $\quad Gc_1 \wedge Gc_2 \wedge Br_2 \wedge Bb \rightarrow [s](Gc_1 \wedge \wedge Bc_2 \wedge Br_2 \wedge \neg Bb)$

A3c $\quad Br_2 \wedge Bc_1 \wedge \neg Bb \wedge Gc_2 \rightarrow [c](Br_2 \wedge Bc_1 \wedge Bb \wedge Gc_2)$

A4r $\neg Br_1 \rightarrow \neg\langle r\rangle\top$

A5s $Br_1 \wedge Bb \rightarrow \langle s\rangle\top$.

Using a PDL theorem prover such as MSPASS [246] (for properties without $*$) or PDL-TABLEAU [386], and instances of axioms A1-A5 such as those above, we can prove a liveness property that if the agent has goals to clean rooms 1 and 2, and starts in the state where its battery is charged and it is in room 1, it can reach a state where both rooms are clean, and a safety property that it is guaranteed to achieve its goal:

$$Gc_1 \wedge Gc_2 \wedge Bb \wedge Br_1 \rightarrow \langle \mathtt{vac}^3\rangle(Bc_1 \wedge Bc_2)$$

$$Gc_1 \wedge Gc_2 \wedge Bb \wedge Br_1 \rightarrow [\mathtt{vac}^3](Bc_1 \wedge Bc_2)$$

where $\mathtt{vac}^3$ stands for vac repeated three times. The MSPASS encoding of the first property is given in Appendix 1.9.1. Its verification using the web interface to MSPASS is virtually instantaneous.

We can also prove, using PDL-TABLEAU, a version of a blind commitment property which states that an agent either keeps its goal or believes it has been achieved:

$$Gc_1 \rightarrow [\mathtt{vac}^+](Bc_1 \vee Gc_1)$$

In the appendix we split the proof of this property into two parts to simplify the PDL-TABLEAU encoding: first we prove using MSPASS $Bc_1 \vee Gc_1 \rightarrow [\mathtt{vac}](Bc_1 \vee Gc_1)$ (see Appendix 1.9.2) and then prove blind commitment using this as a lemma (see Appendix 1.9.3).

The theorem prover encodings given in the appendix are produced by hand, but this process can be automated at the cost of making the encoding larger (at worst, exponential in the size of the agent's program). In the remainder of this section we sketch a naive approach to automating the encoding. Axioms A1, A2, A4 and A5 are straightforward. For every literal l occurring in the agent program, we can generate an instance of axioms A1 and A2. For every belief update action α occurring in the agent's program, we generate an instance of A4, and for each precondition of α, we generate an instance of A5. The difficult case is the axiom schema A3, which is a kind of frame axiom. It includes a formula Φ which intuitively encodes the information about the state which *does not* change after executing an action α. To generate a sufficient number of instances of A3 automatically, we have to use all possible complete state descriptions for Φ (more precisely, all combinations of the agent's beliefs and goals which are not affected by α). Then the instances of A3 will say, for every complete description of the agent's beliefs and goals, what the state of the agent after the performance of α will be. Clearly, this makes the encoding exponential in the number of possible beliefs and goals of the agent. In the vacuum cleaner example, the properties of the agent's state are: its belief about its location, its beliefs about the cleanliness of the two rooms, its belief about its battery, and two possible goals. Even with the domain axioms $\neg(Br_1 \wedge Br_2)$ and $Br_1 \vee Br_2$ (the agent is never in both rooms at the time and it is always in one of the rooms) which reduce the number of possible beliefs about the agent's location to 2, the number of

possible belief states is $2^4 = 16$ and the number of possible combined belief and goal states is $2^6 = 64$, requiring 64 instances of A3 for every action α. Note that this naive approach to automatisation of encoding more or less reduces the theorem proving task to a model-checking task (we use A3, A4 and A5 to specify the transition relation, and the number of instances of A3 is equal to the number of entries in the transition table). However, various ways of reducing the number of axiom instances can be used. For example, we may use an approach similar to slicing [73], or adopt a more efficient way of expressing frame conditions, following the work in situation calculus [383] or, for a language without quantification over actions, [149].

The example above illustrates verification of a program under the non-itnterleaved execution strategy. For verifying program under the interleaved strategy, a PDL theorem prover would need to be adapted to accept program expressions which contain the interleaving operator. Alternatively, the program expression containing the interleaving operator would need to be translated into PDL without interleaving.

1.7 Related Work

There has been a considerable amount of work on verifying properties of agent programs implemented in other agent programming languages such as ConGolog, MetateM, 3APL, 2APL, and AgentSpeak. Shapiro et al. in [393, 395] describe CASLve, a framework for verifying properties of agents implemented in ConGolog. CASLve is based on the higher-order theorem prover PVS and has been used to prove, e.g., termination of bounded-loop ConGolog programs. However, its flexibility means that verification requires user interaction in the form of proof strategies. Properties of agents implemented in programming languages based on executable temporal logics such as MetateM [174], can also easily be automatically verified. However these languages are quite different from languages like SimpleAPL, in that the agent program is specified in terms of temporal relations between states rather than branching and looping constructs. Other related attempts to bridge the gap between agent programs such as 3APL and 2APL on the one hand and verification logics on the other, e.g., [126, 130, 218], have yet to result in an automated verification procedure.

There has also been considerable work on the automated verification of multi-agent systems using model-checking [34, 290]. For example, in [66, 73], Bordini et al. describe work on verifying programs written in Jason, an extension of AgentSpeak(L). In this approach, agent programs together with the semantics of Jason semantics translated into either Promela or Java, and verified using Spin or JPF model checkers respectively. There has also been work on using model checking techniques to verify agent programming languages similar to SimpleAPL [21,370,421]. In this approach agent programs and execution strategies are encoded directly into the Maude term rewriting language, allowing the use of the Maude LTL model

checking tool to verify temporal properties describing the behaviour of agent programs.

The work reported here is an extended and revised version of [6]. It is also closely related to our previous work on using theorem proving techniques to verify agent deliberation strategies [7]. However in that work a fixed general execution strategy is constrained by the execution model to obtain different execution strategies, rather than different execution strategies being specified by different PDL program expressions as in this paper.

1.8 Conclusion

In this paper, we proposed a sound and complete logic which allows the specification of safety and liveness properties of SimpleAPL agent programs as well as their verification using theorem proving techniques. Our logic is a variant of PDL, and allows the specification of safety and liveness properties of agent programs. Our approach allows us to capture the agent's execution strategy in the logic, and we proved a correspondence between the operational semantics of SimpleAPL and the models of the logic for two example execution strategies. We showed how to translate agent programs written in SimpleAPL into expressions of the logic, and gave an example in which we show how to verify correctness properties for a simple agent program using theorem-proving. While we focus on APL-like languages and consider only single agent programs, our approach can be generalised to other BDI-based agent programming languages and the verification of multi-agent systems.

In future work, we would like to develop this verification framework further to deal with agent programming languages extended with plan revision mechanisms.

Acknowledgements

We would like to thank to Renate Schmidt for help with MSPASS and PDL-TABLEAU. Natasha Alechina and Brian Logan were supported by the Engineering and Physical Sciences Research Council [grant number EP/E031226].

1.9 Appendix: Encodings of properties in MSPASS

1.9.1 MSPASS encoding of the example

```
begin_problem(PDL_vacuum_cleaner_example 1).
list_of_descriptions.
 name(* PDL vacuum cleaner example 1 *).
 author(*N. Alechina, M. Dastani, B. Logan, and J.-J. Ch. Meyer *).
 description(* A formula which says that if the vacuum cleaner agent
       starts in room 1 with charged battery and its goal is to clean
       room 1 and room 2, then it will achieve its goals.
       *).
end_of_list.

list_of_symbols.
% Rr - moveRight, Rl - moveLeft, Rs - suck, Rc - charge,
% br1 - believes that in room1, br2 - in room2, bb - battery charged,
% bc1 - believes room1 clean, bc2 - room2 clean,
% gc1 - goal to clean room1, gc2 - clean room2.
predicates[ (Rr,2), (r,0), (Rl,2), (l,0), (Rs,2), (s,0), (Rc,2), (c,0),
           (br1,0), (br2,0), (bb,0), (bc1,0), (bc2,0), (gc1,0),
           (gc2,0) ].
% The following interprets dia(r,...) as accessible by Rr,
% dia(l,...) as accessible by Rl, etc.
translpairs[ (r,Rr), (l,Rl), (s, Rs), (c,Rc) ].
end_of_list.

list_of_special_formulae(axioms, eml).
% instances of A3
prop_formula(
 implies(and(gc1, gc2, br1, bb), box(s, and(bc1, gc2, br1, not(bb))))
).
prop_formula(
 implies(and(bc1, br1, not(bb), gc2),
             box(r, and(br2, bc1, not(bb), gc2)))
).
prop_formula(
 implies(and(br2, bc1, not(bb), gc2), box(c, and(br2, bc1, bb, gc2)))
).
prop_formula(
 implies(and(br2, bc1, bb, gc2), box(s, and(br2, bc1,not(bb), bc2)))
).

% instances of A5
prop_formula(
 implies(bb, dia(s, true))
).
prop_formula(
 implies(br1, dia(r, true))
).
prop_formula(
 implies(br2, dia(l, true))
```

```
).
prop_formula(
 implies(and(br2, not(bb)), dia(c, true))
).
end_of_list.

% The formula we want to prove below is
% gc1 & gc2 & br1 & bb -> <vac><vac><vac> (bc1 & bc2)
% where vac is the vacuum cleaner's program:
% (gc1?; bb?; (br1?;s) U ((not br1)?;l;s)) U
% (gc2?; bb?; (br2?;s) U ((not br2)?;r;s)) U
% ( (not bb)?; (br2?;c) U ((not br2)?;r;c)))

list_of_special_formulae(conjectures, EML).
prop_formula(
 implies(
   and (gc1, gc2, br1, bb),
   dia(
     or(
       % rule1
       comp(test(gc1),
            comp(test(bb),
                 or(comp(test(br1), s),
                    comp(test(not (br1)), comp(l,s))))),
       % rule2
       comp(test(gc2),
            comp(test(bb),
                 or(comp(test(br2), s),
                    comp(test(not (br2)), comp(r,s))))),
       % rule 3
       comp(test(not(bb)),
            or(comp(test(br2), c),
               comp(test(not(br2)), comp(r, c))))
     ), % end first vac or
     dia(
       or(
         % rule1
         comp(test(gc1),
              comp(test(bb),
                   or(comp(test(br1), s),
                      comp(test(not (br1)), comp(l,s))))),
         % rule2
         comp(test(gc2),
              comp(test(bb),
                   or(comp(test(br2), s),
                      comp(test(not (br2)), comp(r,s))))),
         % rule 3
         comp(test(not(bb)),
              or(comp(test(br2), c),
                 comp(test(not(br2)), comp(r, c))))
        ), % end second vac or
        dia(
          or(
            % rule1
```

```
            comp(test(gc1),
                 comp(test(bb),
                      or(comp(test(br1), s),
                         comp(test(not (br1)), comp(l,s)))),
            % rule2
            comp(test(gc2),
                 comp(test(bb),
                      or(comp(test(br2), s),
                         comp(test(not (br2)), comp(r,s)))),
            % rule 3
            comp(test(not(bb)),
                 or(comp(test(br2), c),
                    comp(test(not(br2)), comp(r, c))))
          ), % end third vac or
          and(bc1, bc2))))
 ) % end implies
).
end_of_list.

end_problem.
```

1.9.2 MSPASS encoding of a lemma for the proof of the blind commitment property of the vacuum cleaner agent

```
begin_problem(PDL_vacuum_cleaner_example3).
list_of_descriptions.
 name(* PDL example 3 *).
 author(*N. Alechina, M. Dastani, B. Logan and J.-J. Ch. Meyer*).
 description(* A formula which says that if we start with bc1 or gc1,
                then after each iteration of the program, bc1 or gc1.
       *).
end_of_list.

list_of_symbols.
% Rr - moveRight, Rl - moveLeft, Rs - suck, Rc - charge,
% br1 - believes that in room1, br2 - in room2, bb - battery charged,
% bc1 - believes room1 clean, bc2 - room2 clean,
% gc1 - goal to clean room1, gc2 - clean room2.
predicates[ (Rr,2), (r,0), (Rl,2), (l,0), (Rs,2), (s,0), (Rc,2), (c,0),
            (br1,0), (br2,0), (bb,0), (bc1,0), (bc2,0), (gc1,0), (gc2,0)].
% The following interprets dia(r,...) as accessible by Rr,
% dia(l,...) as accessible by Rl, etc.
translpairs[ (r,Rr), (l,Rl), (s, Rs), (c,Rc) ].
end_of_list.

list_of_special_formulae(axioms, eml).
% world axioms
prop_formula(
 not(and(br1,br2))
).
```

```
prop_formula(
 or(br1,br2)
).
% instances of A2
prop_formula(
 not(and(gc1,bc1))
).
prop_formula(
 not(and(gc2,bc2))
).
% instances of A3
prop_formula(
 implies(and(gc1, bb), box(s, or(bc1, gc1)))
).
prop_formula(
 implies(and(bc1, bb), box(s, or(bc1, gc1)))
).
prop_formula(
 implies(and(bc1, br1), box(r, and(bc1, br2)))
).
prop_formula(
 implies(and(gc1, br1), box(r, and(gc1, br2)))
).
prop_formula(
 implies(and(bc1, br2,not(bb)), box(c, and(bc1, br2, bb)))
).
prop_formula(
 implies(and(gc1, br2,not(bb)), box(c, and(gc1, br2, bb)))
).
prop_formula(
 implies(and(gc1, br2), box(l, and(gc1, br1)))
).
prop_formula(
 implies(and(bc1, br2), box(l, and(bc1, br1)))
).
% instances of A4
prop_formula(
 implies(not(bb), not(dia(s, true)))
).
prop_formula(
 implies(not(br1), not(dia(r, true)))
).
prop_formula(
 implies(not(br2), not(dia(l, true)))
).
prop_formula(
 implies(not(and(br2, not(bb))), not(dia(c, true)))
).

% instances of A5
prop_formula(
 implies(bb, dia(s, true))
).
prop_formula(
```

```
 implies(br1, dia(r, true))
).
prop_formula(
 implies(br2, dia(l, true))
).
prop_formula(
 implies(and(br2, not(bb)), dia(c, true))
).
end_of_list.

% The formula we want to prove below is
% bc1 v gc1 -> [vac] (bc1 v gc1)
% where vac is the vacuum cleaner's program:
% (gc1?; bb?; (br1?;s) U ((not br1)?;l;s)) U
% (gc2?; bb?; (br2?;s) U ((not br2)?;r;s)) U
% ( (not bb)?; (br2?;c) U ((not br2)?;r;c)))
list_of_special_formulae(conjectures, EML).
prop_formula(
 implies(
   or(bc1,gc1),
   box(
      or(
       % rule1
       comp(test(gc1),
            comp(test(bb),
                 or(comp(test(br1), s),
                    comp(test(not (br1)), comp(l,s))))),
        % rule2
        comp(test(gc2),
             comp(test(bb),
                  or(comp(test(br2), s),
                     comp(test(not (br2)), comp(r,s))))),
        % rule 3
        comp(test(not(bb)),
             or(comp(test(br2), c),
                comp(test(not(br2)), comp(r, c))))
      ), % end first vac or
          or(bc1,gc1) )
 ) % end implies
).
end_of_list.

end_problem.
```

1.9.3 PDL-TABLEAU *encoding of the blind committment property*

```
prove(
%axioms
[
 implies(and(gc1, br1, bb), box(s, and(bc1, br1, not(bb)))),
```

```
 implies(and(gc1, br2, bb), box(s, and(gc1, bc2, br2, not(bb)))),
 implies(and(bc1, br1, bb), box(s, and(bc1, br1, not(bb)))),
 implies(and(bc1, br2, bb), box(s, and(gc1, bc2, br2, not(bb)))),
 implies(and(bc1, br1), box(r, and(bc1, br2))),
 implies(and(gc1, br1), box(r, and(gc1, br2))),
 implies(and(bc1, br2,not(bb)), box(c, and(bc1, br2, bb))),
 implies(and(gc1, br2,not(bb)), box(c, and(gc1, br2, bb))),
 implies(and(gc1, br2), box(l, and(gc1, br1))),
 implies(and(bc1, br2), box(l, and(bc1, br1))),
 implies(not(bb), not(dia(s, true))),
 implies(not(br1), not(dia(r, true))),
 implies(not(br2), not(dia(l, true))),
 implies(not(and(br2, not(bb))), not(dia(c, true)))
],
% The formula we want to prove below is
% gc1 -> [vac*] (bc1 v gc1)
% where vac is the vacuum cleaner's program:
% (gc1?; bb?; (br1?;s) U ((not br1)?;l;s)) U
% (gc2?; bb?; (br2?;s) U ((not br2)?;r;s)) U
% ( (not bb)?; (br2?;c) U ((not br2)?;r;c)))
 implies(
   gc1,
   box(star(
      or(
       % rule1
       comp(test(gc1),
            comp(test(bb),
                 or(comp(test(br1), s),
                    comp(test(not (br1)), comp(l,s))))),
        % rule2
        comp(test(gc2),
             comp(test(bb),
                  or(comp(test(br2), s),
                     comp(test(not (br2)), comp(r,s))))),
        % rule 3
        comp(test(not(bb)),
             or(comp(test(br2), c),
                comp(test(not(br2)), comp(r, c))))
      )),
          or(bc1,gc1)))
). % end prove
```

Chapter 2
The Refinement of Multi-Agent Systems

L. Aştefănoaei and F.S. de Boer

Abstract This chapter introduces an encompassing theory of refinement which supports a top-down methodology for designing multi-agent systems. We present a general modelling framework where we identify different abstraction levels of BDI agents. On the one hand, at a higher level of abstraction we introduce the language BUnity as a way to specify "what" an agent can execute. On the other hand, at a more concrete layer we introduce the language BUpL as implementing not only what an agent can do but also "how" the agent executes. At this stage of individual agent design, refinement is understood as trace inclusion. Having the traces of an implementation included in the traces of a given specification means that the implementation is correct with respect to the specification.

We generalise the theory of agent refinement to multi-agent systems in the presence of new coordination mechanisms extended with real time. The generalisation is such that refinement is compositional. This means that refinement at the individual level implies refinement at the multi-agent system level. Compositionality is an important property since it reduces heavily the verification process. Thus having a theory of refinement is a crucial step towards the verification of multi-agent systems' correctness.

L. Aştefănoaei
CWI (Centrum Wiskunde en Informatica), The Netherlands e-mail: astefano@cwi.nl

F.S. de Boer
CWI (Centrum Wiskunde en Informatica), The Netherlands e-mail: F.S.de.Boer@cwi.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_2,

2.1 Introduction

In this chapter we describe a top-down methodology for designing multi-agent systems by refinement. We first focus on the design of individual agents. At this stage, *refinement* means to reduce the non determinism of high-level agent specification languages. Reducing the non determinism boils down to scheduling policies, i.e., to setting an order (possibly and time) of action executions. The agent specification language we consider is abstract with respect to scheduling policies. It is inspired by UNITY [94], a classical design methodology which emphasises the principles:

- "specify little in early stages of design" and
- "program design at early stages should not be based on considerations of control flow".

We place ourselves in the framework of BDI models [86]. As already a standard notion, an agent is defined in terms of *beliefs*, *desires*, *intentions*. Beliefs and desires (goals) usually represent the mental state of an agent, and intentions denote the deliberation phase of an agent (often concretising the choice of executing a *plan*).

We introduce BUnity as an extension of the UNITY language to the BDI paradigm. It is meant to represent an agent in the first stage of design. One only needs to specify initial beliefs and actions (*what* an agent can do). We make the observation that "specify little" implies non deterministic executions of BUnity agents (actions may be executed in any arbitrary order, for example). On the other hand, BUpL (**B**elief **U**pdate **p**rogramming **L**anguage) enriches BUnity constructions with the notions of *plans* and *repair rules*. These are meant to refine the early stage non determinism by specifying *how* and *when* actions are executed. In fact, plans implement scheduling policies. We have chosen BUpL as the representation of agents in the last stage of design. Having fixed the levels of abstraction as being BUnity and BUpL, we focus on the correctness of a given BUpL agent with respect to a BUnity specification. By correctness we mean refinement, which is usually understood as trace inclusion. A BUpL agent is correct with respect to a BUnity specification if any possible BUpL behaviour (trace) is also a BUnity one. Since we are interested in applying in practise our methodology, and since verifying trace inclusion is computationally hard, we further focus on simulation as a proof technique for refinement. Additionally, since agents might have infinite behaviours, some of which are unlikely to occur in practice, we provide a declarative approach to modelling fairness and show how simulation works in such a context.

A clear extension of the above framework consists of applying the same methodology to a multi-agent setting. A first step is to lift the notion of abstraction levels from individual agents to multi-agent systems. Considering that the behaviour of the multi-agent system is simply the sum of the behaviours of individual agents is a too unrealistic idea. Instead, we propose *action-based coordination* mechanisms, to which we refer as *choreographies*. They represent global synchronisation and ordering conditions restricting the execution of individual agents.

Introducing coordination while respecting the autonomy of the agents is still a challenge in the design of multi-agent systems. However, the advantage of the infrastructures we propose lies in their *exogenous* feature: the update of the agent's mental states is separated from the coordination pattern. Nobody changes the agent's beliefs but itself. Besides that choreographies are oblivious to mental aspects, they control without having to know the internal structure of the agent. For example, whenever a choice between plans needs to be taken, a BUpL agent is free to make its own decision. The degree of freedom can be seen also in the mechanism for handling action failures. The agent chooses one among possibly many available repair rules without being constraint by the choreography. In these regards, the autonomy of agents executed with respect to choreographies is preserved.

Extending the refinement relation from individual agents to multi-agent systems requires solving a new problem since choreographies may introduce deadlocks. It can be that though there is refinement at the individual agent level, adding a choreography deadlocks the concrete multi-agent system but not the abstract one. We take, as example, a choreography which specifies a BUpL agent to execute an action not defined in the agent program itself (but only in the BUnity specification). In this situation, refinement as trace inclusion trivially holds since the set of traces from a deadlocked state is empty. Our methodology in approaching this problem consists of, basically, formalising the following aspects. On the one hand, we define the semantics of multi-agent systems with choreographies as the set of *maximal traces*, where we make the distinction between a *success* and a *deadlock*. These traces consist of the parallel agents' executions guided by the choreography. We define multi-agent system refinement as maximal trace inclusion. On the other hand, agent refinement becomes *ready trace* inclusion, where a ready trace records not only the actions being executed, but also those ones which *might* be executed. We show that multi-agent system refinement is *compositional*. More precisely, the main result is that agent refinement implies multi-agent system refinement in the presence of *any* choreography. Furthermore, the refined multi-agent system does not introduce deadlocks with respect to the multi-agent system specification.

The last extension we propose regards time. A more expressive framework can be obtained when action synchronisations depend also on time, not only on the disposal of the agents to perform the actions. Thus, we address the problem of incorporating time into choreographies such that the compositionality result we have remains valid in the timed version. A first step is to extend choreographies by means of timed automata [10] such that they constrain the *timings* of the actions. Having *timed choreographies* requires, however, introducing time in agents themselves. Thus, in our case, BUnity and BUpL need to be extended such that they reflect the passing of time. In this respect, we have in mind that basic actions are a *common ontology* shared by all agents. Since the nature of basic actions does not specify *when* to be executed, our extension is such that the ontology remains timeless and "when" becomes part of the specific agent applications.

Our contribution consists of introducing a general framework for modelling and not programming agent languages. BUnity and BUpL are simple but expressive

agent languages, inspired by the already standard GOAL [60] and 3APL [222] languages. The operational semantics of the languages makes it easy to prototype them as rewrite systems in Maude [105]. Maude has the advantage that it offers both execution (by rewriting) and verification (by model-checking) of the prototyped systems. We stress the importance of prototyping before implementing complex agent platforms. It is a quick method for proving that the semantics fulfils the initial requirements. We emphasise that all the effort of introducing the formalism of multi-agent system refinement is motivated by the need to perform verification. Multi-agent systems are clearly more complex structures, and their verification tends to become harder. However, in our framework, given the compositionality result, it is enough to verify individual agents in order to conclude properties about the whole multi-agent system.

2.1.1 Related Works

The design methodology we propose integrates in a unifying approach different concepts and results from process theory [196]. Some aspects we deal with have been taken into account in different works, however, from a distinct angle. Considering verification techniques for multi-agent systems, there are already some notable achievements: [73] discusses model-checking AgentSpeak systems, [82] proposes Temporal Trace Language for analysing dynamics between agents, [355] refers to verifying deontic interpreted systems. However, we focus on the compositionality of the refinement relation which reduces the problem of verifying the whole multi-agent system to verifying the agents composing it. Concerning interaction in multi-agent systems, this is usually achieved by means of communication. Communication, in turn, is implemented by message passing, or channel-based mechanisms. This latter can be seen as the basis of implementing coordination artifacts. Such artifacts are usually built in terms of resource access relation in [368], or in terms of action synchronisation [18]. We also mention the language Linda [190] which has not yet been applied in a multi-agent setting but to service oriented services, where the notion of *data* plays a more important role than *synchrony*. Control can be also achieved by using social and organisational concepts (e.g., norms, roles, groups, responsibility) and mechanisms (monitoring agents' actions and sanctioning) [145]. Organisation-based coordination artifacts are recently discussed in [49, 124, 415]. The concepts of choreography and orchestration have already been introduced to web services (to the paradigm Service-oriented Computing), see [25, 311, 317] for different approaches. Though we use the same terminology, our notion of choreography is in essence different since we deliberately ignore communication issues. The choreography model we define is explicit whereas in the other works choreography is implicit in the communication protocol. Thus, we need to take into account deadlock situations that may appear because of "mall-formed" choreographies. Being external, the choreography represents, in fact, contexts while in the other approaches there is a distinction between the modularity and the contextuality of the communi-

cation operator. With respect to timed automata, we mention that its application in a multi-agent system is new. However, timed automata have already been applied to testing real-time systems specifications [214] or to scheduling problems [85].

2.2 From Specification to Implementation Agent Languages

In this section we introduce two agent languages, BUnity and BUpL, each corresponding to different levels of abstractions, with the latter being the more concrete one. We further focus on the correctness of a concrete BUpL agent with respect to a more abstract BUnity agent, where by correctness we understand refinement. In order to automatise such a correctness result we take advantage of simulation as being a sound and complete (under determinacy conditions) proof technique for refinement. Before presenting our methodology, we first recall a few elementary notions from process theory.

2.2.1 Preliminaries

We consider labelled transition systems (LTS) as tuples $(\Sigma, s_0, Act, \rightarrow)$, where Σ is a finite set of states, s_0 is an initial state, Act is a set of actions (labels), and $\rightarrow$ describes all possible transitions. We denote by τ a special action called *silent* action. $Act - \{\tau\}$ is the set of *visible* actions. We write $s \xrightarrow{a} s'$ when $(s,a,s') \in \rightarrow$, meaning "s may become s' by performing an action labelled a". It also means that transition a is enabled on s. We say that a transition system is *deterministic* when for any state s, and for any action a, $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ implies $s'' = s'$. We call $s \xrightarrow{\tau} s'$ an idling transition and we abbreviate it by $s \rightarrow s'$. The "weak" arrow $\Rightarrow$ denotes the reflexive and transitive closure of $\rightarrow$, and $\overset{a}{\Rightarrow}$ stands for $\Rightarrow\xrightarrow{a}\Rightarrow$. A *computation* in a transition system is defined to be a sequence of the form $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 ...$, where $l_i \in Act, i \in \mathbb{N}$. It can be either finite (when there is no possible transition from the last state), or infinite. For a computation σ, the corresponding *trace*, $tr(\sigma)$, is the sequence of visible actions (a word defined on $(Act - \{\tau\})^\omega$). The set of all traces of a system S (the traces corresponding to all computations starting with the initial state of the system) is denoted by $Tr(S)$.

2.2.2 Formalising Mental States and Basic Actions

In the current approach, the underlying logical framework of mental states is a fragment of Herbrand logic. We consider F and $Pred$ infinite sets of *function*, resp.

predicate symbols, with a typical element f, resp. P. *Variables* are denoted by the symbol x. Each function symbol f has associated a non-negative integer n, its *arity*. Function symbols with 0-arity are also called *constants*. *Terms*, usually denoted by the symbol t, are built from function symbols and variables. *Formulae*, usually denoted by the symbol φ, are built from predicates and the usual logical connectors. To sum up, the BNF grammar for terms and formulae is as follows:

$$\begin{aligned} t &::= x \mid f(t,\ldots,t) \\ \varphi &::= P(t,\ldots,t) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \end{aligned}$$

An *atom* is any formula $P(t,\ldots,t)$. A *literal* is either an atom or the negation of an atom. A term with no variables is called *ground*. A formula is either ground, or *open*. In our case, we consider that open formulae have no quantifiers, all variables are understood as being existential. The set of all ground atoms built upon F and *Pred* is a *Herbrand model*.

Mental states are characterised in terms of *beliefs*. In the current framework, we consider beliefs as ground atoms, organised in the so-called *belief bases* (subsets of the Herbrand model), which we denote by $\mathcal{B}$.

Given $\mathcal{B}$ a belief base, the satisfaction relation for ground formulae is defined using structural induction, as usually. For defining the satisfaction relation of open formulae, we consider the usual notion of *substitutions* as functions that replace variables with terms, denoted by $[x/t]\ldots[x/t]$. Given a syntactical expression e and a substitution θ, we denote the application of θ to e by $e\theta$, rather than by $\theta(e)$. The composition $\theta\theta'$ of two substitutions θ and θ' is defined as $\theta\theta'(x) := \theta(\theta'(x))$, and it is associative. The satisfaction relation for open formulae is as follows:

$$\mathcal{B} \models \varphi \text{ iff exists } \theta \text{ s.t. } \mathcal{B} \models \varphi\theta$$

The substitution θ is ground (the substituting terms have no variables). This is because the belief base $\mathcal{B}$ is ground. We make the remark that θ is obtained by solving a *matching* (and not *unification*) problem. From a complexity point of view, this is important since it is easier to implement a linear algorithm for matching than for unification. We say a term s *matches* a ground term t if there exists a substitution (called *matcher*) such that $s\theta$ is syntactically equal to t. The matching problem extends easily to formulae and belief bases. We consider $Sols(\mathcal{B} \models \psi)$ as the set of all matchers. We say the substitution is the identity function when ψ is a ground formula satisfied in $\mathcal{B}$.

Basic actions are functions defined as pairs (ψ,ξ), where ψ are formulae which we call *preconditions*, and ξ are sets of literals which we call *effects*. The following inclusions are required:

$$Var(\xi) \subseteq Var(\psi) = \{x_1,\ldots,x_n\},$$

where $Var(e)$ denotes the set of variables in a syntactic expression e. We use the symbol $\mathcal{A}$ for the set of basic actions' definitions. We refer to Act as the set of basic action names with typical element a. We further use the notation $a\theta$ to represent

action terms which result from the application of substitutions. By abuse of notation we sometimes refer to $a(\overline{x})$ (resp. $\psi(\overline{x})$, $\xi(\overline{x})$) as a (resp. ψ, ξ) when $\overline{x}$, the set of variables, is not relevant.

Given a basic action definition $a = (\psi,\xi)$, if matching ψ to $\mathcal{B}$ has a solution θ, then the effect of $a\theta$ is to update the belief base by adding or removing ground atoms from the set $\xi\theta$:

$$\begin{cases} \mathcal{B} \uplus l\theta = \mathcal{B} \cup l\theta, & l \in \xi \\ \mathcal{B} \uplus \neg l\theta = \mathcal{B} \setminus l\theta, & \neg l \in \xi \end{cases}$$

We write $\mathcal{B} \uplus \xi\theta$ to represent the result of an update operation, which is automatically guaranteed to be consistent since we add only positive literals.

2.2.3 BUnity Agents

BUnity language represents abstract agent specifications. Its purpose is to model agents at a coarse level, using a minimal set of constructions. A BUnity agent abstracts from specific orderings (for example, action planning). Hence her executions are highly non deterministic.

The mental state of a BUnity agent is simply a belief base. On top of basic actions, Bunity language allows a finer type of construction, *conditional actions*, which are organised in a set denoted by C. A conditional action is built upon a basic action. It is syntactically defined by $\phi \triangleright do(a)$, where ϕ is a query on the belief base, and a is the name of an action. Intuitively, conditional actions are like *await* statements in imperative languages: **await** ϕ **do** a, action a can be executed only when ϕ matches the current belief base.

Both basic and conditional actions have enabling conditions, and this might raise confusion when distinguishing them. The intuition lying behind the need to consider them both is that they demand information at different levels.The precondition of a basic action should be understood a built-in restriction which internally enables belief updates. It is independent of the particular choice of application. A conditional action is executed in function of certain external requirements (reflected in the mental state). Thus it is application dependent. The external requirements are meant to be independent of the built-in enabling mechanism. Whether is agent "A" or agent "B" executing an action a, the built-in condition should be the same for both of them. Nevertheless, each agent may have its own external trigger for action a.

A BUnity agent is defined as a tuple, $(\mathcal{B}_0, \mathcal{A}, C)$, where $\mathcal{B}_0$ is a set of initial beliefs. For such a configuration, we define an operational semantics in terms of labelled transition systems.

Definition 2.1 (BUnity Semantics). Let $(\mathcal{B}_0, \mathcal{A}, C)$ be a BUnity configuration. The associated LTS is $(\Sigma, \mathcal{B}_0, L, \rightarrow)$, where:

- Σ is a set of states (belief bases)
- $\mathcal{B}_0$ is the initial state (the initial belief base)
- L is a set of ground action terms
- $\rightarrow$ is the transition relation, given by the rule:

$$\frac{\phi \triangleright do(a) \in C \qquad a = (\psi, \xi) \in \mathcal{A} \qquad \theta \in Sols(\mathcal{B} \models (\phi \wedge \psi))}{\mathcal{B} \xrightarrow{a\theta} \mathcal{B} \uplus \xi\theta} \ (act)$$

We consider the meaning of a BUnity agent defined in terms of possible sequences of mental states, and its externally observable *behaviour* as sequences of executed actions. Equally, the meaning of a BUnity agent is the set of all possible computations of its associated LTS, and the behaviour, the trace set (as words on action terms). Our focus on visible actions (and not states) is motivated by the fact that, in studying simulation, we are interested in *what* we *see* and not *how* the agent *thinks*. We take the case of a robot: one simulates his physical actions, lifting or dropping a block, for example, and not the mental states of the robot.

The transition rule (*act*) captures the effects of performing the action a. It basically says that if there is a conditional action $\phi \triangleright do(a)$ and the query ϕ has a solution in the current mental state then if the precondition of a matches the current belief base new beliefs are added/deleted with respect to the effects of a.

We take, as an illustration, a known problem, which we first found in [222], of an agent building a tower of blocks. An initial arrangement of three blocks A, B, C is given there: A and B are on the floor, and C is on top of A. The goal[1] of the agent is to rearrange them such that A is on the floor, B on top of A and C on top of B. The only action an agent can execute is to move one block on the floor, or on top of another block, if the latter is free.

$$\begin{aligned}
\mathcal{B}_0 &= \{\ on(C,A),\ on(A,floor),\ on(B,floor),\\
&\qquad free(B),\ free(C),\ free(floor)\ \}\\
\mathcal{A} &= \{\ move(x,y,z) = (on(x,y) \wedge free(x) \wedge free(z),\\
&\qquad \{\ on(x,z),\ \neg on(x,y),\ \neg free(z)\ \}\ )\ \}\\
C &= \{\ \neg(on(B,A) \wedge on(C,B)) \triangleright do(move(x,y,z))\ \}
\end{aligned}$$

Fig. 2.1 A BUnity Toy Agent

The example from Figure 2.1 is taken in order to underline the difference between enabling conditions (for basic actions) and triggers (for conditional actions): on the one hand, it is possible to move a block x on top of another block z, if x and z are free; on the other hand, given the goal of the agent, moves are allowed only when the configuration is different than the final one.

[1] We do not explicitly model goals. Please check Section 2.2.7 for a discussion motivating our choice.

2.2.4 Why BUnity Agents Need Justice

In non deterministic systems that abstract from scheduling policies, some traces are improbable to occur in real computations. In this sense, the operational semantics (given by transition rules) is too general in practise: if the actions an agent can execute are always enabled, it should not be the case that the agent always chooses the same action. Such executions are usually referred to as being *unfair*.

For example, we imagine a scenario illustrative for cases where modelling fairness constraints is a "must". For this, we slightly complicate the "tower" problem from the previous section, by giving the agent described in Figure 2.1 an extra assignment to clean the floor, if it is dirty. Thus the agent have two alternatives: either to clean or to build. We add a basic action, $clean = (\neg\ cleaned, \{cleaned\})$. We enable the agent to execute this action at any time, by setting $\top$ as the query of the conditional action calling *clean*, i.e., $\top \triangleright do(clean)$.

We note that it is possible that the agent always prefers cleaning the floor instead of rearranging blocks, in the case that the floor is constantly getting dirty. We want to cast aside such traces and moreover, we want a declarative, and not imperative solution. Our option is to follow the approach from [296]: we constrain the traces by adding *fairness* conditions, modelled as linear temporal logic (LTL) properties. Fairness is there expressed either as a weak, or as a strong constraint. They both express that actions which are "many times" enabled on infinite execution paths should be infinitely often taken. The difference between them is in the definition of "many times" which is continuously (resp. infinitely often). Due to the semantics of conditional actions, it follows that the *choice* of executing one action cannot disable the ones not chosen and thus BUnity agents only need weak fairness.

Definition 2.2 (Justice [296]). A trace is just (weakly fair) with respect to a transition a if it is not the case that a is continually enabled beyond some position, but taken only a finite number of times.

To model such a definition as LTL formulae we need only two future operators, $\Diamond$ (*eventually*) and $\Box$ (*always*). Their satisfaction relation is defined as follows:

$$\sigma \models \Diamond\phi \text{ iff } (\exists i > k)(s_i \models \phi)$$
$$\sigma \models \Box\phi \text{ iff } (\forall i > k)(s_i \models \phi),$$

where $s_0, \ldots, s_k, \ldots$ are the states of a computation σ. By means of these operators we define weak fairness for BUnity as:

$$just_1 = \bigwedge_{a \in Act} (\Diamond\Box\ enabled(\phi \triangleright do(a)) \rightarrow \Box\Diamond\ taken(a)) \ .$$

where *enabled* and *taken*, predicates on the states of BUnity agents, are defined as:

$$\mathcal{B} \models enabled(\phi \triangleright do(a)) \text{ iff } a = (\psi, \xi) \wedge \mathcal{B} \models \phi \wedge \psi$$
$$\mathcal{B} \models taken(a) \qquad \text{iff } \mathcal{B} \xrightarrow{a\theta} \mathcal{B}'$$

Such a fairness condition ensures that all fair BUnity traces are of the form $(clean^*\ (move\theta)^*)^\omega$, or equally $\{(clean^n\ (move\theta)^m)^k \mid \forall n, m \in \mathbb{N}, k \in \mathbb{N} \cup \{\infty\}\}$. We note that the advantage of a declarative approach to modelling fairness is the fact that we do not need to commit to a specific scheduling policy as it is the case when implementing fairness by means of a scheduling algorithm, eg., Round-Robin. A scheduling policy would basically correspond to fixing the exponents n and m.

2.2.5 BUpL Agents

The BUnity agent described in Figure 2.1 is highly non deterministic. It is possible that the agent moves C on the floor, B on A, and C on B. This sequence represents, in fact, the shortest one to achieving the goal. However, it is also possible that the agent pointlessly move C from A to B and then back from B to A.

BUpL language allows the construction of *plans* as a way to order actions. We refer to $\mathcal{P}$ as a set of plans, with a typical element p, and to Π as a set of plan names, with a typical element π. Syntactically, a plan is defined by the following BNF grammar:

$$p ::= a(t,\ldots,t) \mid \pi(t,\ldots,t) \mid a(t,\ldots,t); p \mid p + p$$

with ';' being the *sequential composition* operator and '+' the *choice* operator, with a lower priority than ';'.

The construction $\pi(x_1,\ldots,x_n)$ is called *abstract plan*. It is a function of arity n, defined as $\pi(x_1,\ldots,x_n) = p$. Abstract plans should be understood as procedures in imperative languages: an abstract plan calls another abstract plan, as a procedure calls another procedure inside its body.

BUpL language provides a mechanism for handling the failures of actions in plans through constructions called *repair rules*. A plan fails when the current action cannot be executed. Repair rules replace such a plan with another. Syntactically, they have the form $\phi \leftarrow p$, and it means: if ϕ matches $\mathcal{B}$, then substitute the plan that failed for p.

A BUpL agent is a tuple $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$, where $\mathcal{B}_0$, $\mathcal{A}$ are the same as for a BUnity agent, p_0 is the *initial plan*, $\mathcal{P}$ is a set of plans and $\mathcal{R}$ is a set of repair rules.

Plans, like belief bases, have a dynamic structure, and this is why the mental state of a BUpL agent incorporates both the current belief base and the plan in execution. The operational semantics for a BUpL agent is as follows:

Definition 2.3 (BUpL Semantics). Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL configuration. Then the associated LTS is $(\Sigma, (\mathcal{B}_0, p_0), L, \rightarrow)$, where:

- Σ is a set of states, tuples $(\mathcal{B}, p)$
- $(\mathcal{B}_0, p_0)$ is the initial state

$$\frac{p = (a; p') \quad a = (\psi, \xi) \in \mathcal{A} \quad \theta \in Sols(\mathcal{B} \models \psi)}{(\mathcal{B}, p) \xrightarrow{a\theta} (\mathcal{B} \uplus \xi\theta, p'\theta)} \ (act)$$

$$\frac{(\mathcal{B}, p_i) \xrightarrow{\mu} (\mathcal{B}', p')}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\mu} (\mathcal{B}', p')} \ (sum_i)$$

$$\frac{(\mathcal{B}, a; p) \not\xrightarrow{a} \quad \phi \leftarrow p' \in \mathcal{R} \quad \theta \in Sols(\mathcal{B} \models \phi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \ (fail)$$

$$\frac{\pi(x_1, \ldots, x_n) := p}{(\mathcal{B}, \pi(t_1, \ldots, t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1, \ldots, t_n))} \ (\pi)$$

Fig. 2.2 BUpL Rules

- L is a set of labels, either ground action terms or τ
- $\rightarrow$ represents the transition rules given in Figure 2.2.

As it was the case for BUnity agents, we consider the meaning of a BUpL agent defined in terms of possible sequences of mental states, and its externally observable *behaviour* as sequences of executed actions.

The transition rule (*act*) captures the effects of performing the action a which is the head of the current plan. It basically says that if θ is a solution to the matching problem $\mathcal{B} \models \psi$ where ψ is the precondition of action a then the current mental state changes to a new one, where the current belief base is updated with the effects of a and the current plan becomes the "tail" of the previous one. The transition rule ($fail$) handles exceptions. If the head of the current plan is an action that cannot be executed (the set of solutions for the matching problem is empty) and if there is a repair rule $\phi \leftarrow p'$ such that the new matching problem $\mathcal{B} \models \phi$ has a solution θ then the plan is replaced by $p'\theta$. The transition rule (π) implements "plan calls". If the abstract plan $\pi(x_1, \ldots, x_n)$ defined as $p(x_1, \ldots, x_n)$ is instantiated with the terms $t_1, \ldots, t_n$ then the current plan becomes $p(t_1, \ldots, t_n)$ which stands for $p[x_1/t_1] \ldots [x_n/t_n]$. The transition rule ($sum_i$) replaces a choice between two plans by either one of them. The label μ can be either a ground action name or a τ step, in which case $\mathcal{B}' = \mathcal{B}$, and p' is a valid repair plan (if any).

We take as an example a BUpL agent that solves the *tower of blocks* problem. It has the same initial belief base and the same basic action as the BUnity agent.

The BUpL agent from Figure 2.3 is modelled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally telling the agent to move B on A. Similar scenarios can easily arise in multi-agent systems: imagine that initially C is on the floor, and the agent decides to move B on A; imagine also that another

$$\begin{aligned}
\mathcal{B}_0 &= \{\, on(C,A), on(A,floor), on(B,floor), \\
&\quad\ free(B), free(C), free(floor) \,\} \\
\mathcal{A} &= \{\, move(x,y,z) = (on(x,y) \wedge free(x) \wedge free(z), \\
&\quad\ \{\, on(x,z), \neg on(x,y), \neg free(z) \,\}\,) \,\} \\
\mathcal{P} &= \{\, p_0 = move(B,floor,A); move(C,floor,B) \,\} \\
\mathcal{R} &= \{\, on(x,y) \leftarrow move(x,y,floor); p_0 \,\}
\end{aligned}$$

Fig. 2.3 A BUpL Toy Agent

agent comes and moves C on top of A, thus moving B on A will fail. The failure is handled by $on(x,y) \leftarrow move(x,y,floor); p_0$. Choosing $[x/A][y/C]$ as a matcher, enables the agent to move C on the floor and after the initial plan can be restarted.

2.2.6 Why BUpL Agents Need Compassion

Though BUpL agents are meant to reduce the non determinism from BUnity agent specifications, unfair executions are not ruled out because of the non determinism in the choices between plans and/or repair rules. To illustrate this, we assign a *mission* plan to the BUpL agent described in Figure 2.3, $mission = cleanR + rearrange(B,A,C)$, where *cleanR* is a tail-recursive plan, $cleanR = clean; cleanR$, with *clean* being the action defined in Section 2.2.4. The plan *rearrange* generalises the previously defined p_0: $rearrange(x,y,z) = move(x,floor,y); move(z,floor,x)$. It consists of reorganising free blocks placed on the floor, such that they form a tower. This plan fails if not all the blocks are on the floor, and the failure is handled by the already defined repair rule, which we call r_1. We add a repair rule, r_2, $\top \leftarrow mission$, which simply makes the agent restart the execution of the plan *mission*.

As it was the case with the BUnity agent from the Section 2.2.4, it is possible, in the above scenario, that the BUpL agent always prefers cleaning the floor instead of rearranging blocks, though this is useless when the floor has already been cleaned. Nevertheless, such cases are disregarded if one requires that executions are fair. The only difference from the fairness condition imposed on the executions of BUnity agents is that plans need not be continuously but infinitely often enabled.

We consider two scenarios for defining fairness with respect to choices in repair rules and plans. The execution of *rearrange* has failed. Both repair rules r_1 and r_2 are enabled, and always choosing r_2 makes it impossible to make the rearrangement. This would not be the case if r_1 were triggered. It follows that the choice of repair rules should be weakly fair:

$$just_2 = \bigwedge_{p \in \mathcal{P}} (\Diamond\Box\, enabled(\phi \leftarrow p) \rightarrow \Box\Diamond\, taken(p)) \,.$$

The repair rule r_1 has been applied, and all three blocks are on the floor. Returning to the initial mission and being in favour of cleaning leads again to a failure (the floor is already clean). The only applicable repair rule is r_2 which simply tells the agent to return to the mission. Thus, it can be the case that, though rearranging the blocks is enabled, it will never happen, since the choice goes for the plan *clean* (which always fails). Therefore, because plans are not continuously enabled, their choice has to be strongly fair:

Definition 2.4 (Compassion [296]). A trace is compassionate (strongly fair) with respect to a transition a if it is not the case that a is infinitely often enabled beyond some position, but taken only a finite number of times.

As it was the case with justice, modelling the above definition as a linear temporal logic formula is straight-forward, however we refer to *plans* instead of actions:

$$compassionate = \bigwedge_{p \in \mathcal{P}} (\Box\Diamond\, enabled(p) \rightarrow \Box\Diamond\, taken(p))$$

In the above scenarios *enabled* and *taken* are defined similarly as in the case of actions for the language BUnity: (1) a repair rule is enabled when its precondition is satisfied in the belief base; (2) a plan is enabled when the precondition of its first action is satisfied; (3) a plan is taken when its first action is taken.

$$\begin{aligned}
&(\mathcal{B}, a; p) \models enabled(a; p) && \text{iff } a = (\psi, \xi) \wedge \mathcal{B} \models \psi \\
&(\mathcal{B}, a; p) \models enabled(\phi \leftarrow p') && \text{iff } \mathcal{B} \models \phi \\
&(\mathcal{B}, a; p) \models taken(a; p) && \text{iff } (\mathcal{B}, a; p) \xrightarrow{a} (\mathcal{B}', p)
\end{aligned}$$

The fairness conditions ensure that all fair BUpL traces are of the form $(clean^* (move\theta)^*)^\omega$ which is exactly the same as in the case of the BUnity agent. This is a positive result, since we are interested in the fair refinement between the BUpL and the BUnity agent.

2.2.7 Appraising Goals

We have deliberately cast aside *goals* in BUnity and BUpL. This is mainly for simplicity reasons. The usual way ([60, 222]) to explicitly incorporate goals is to fix a particular representation, for example, as a conjunction of ground atoms (which we might understand as a special case of a belief base). The corresponding change in the semantics is to extend the queries of BUnity conditional actions and of BUpL repair rules such that they do not interrogate only belief bases but also goals. Additionally, plan calls should be extended such that goals can trigger plan executions.

Given that our focus is on verification, being able to represent goals implicitly is acceptable enough in our framework. Furthermore, the expressive power of the

languages is not necessarily decreased. We can, without changing the syntax and the semantics of the languages, have a declarative modelling of goals as LTL formulae. In such a situation, we would be interested in any agent execution which satisfies a given goal. This problem can be equally stated as a reachability problem and the answer can be provided by verification. More precisely, model-checking the negation of the goal returns, in fact, a counter-example denoting a successful trace (leading to the achievement of the goal) in the case that there exists one.

For example, we can define, with respect to the scenario introduced in the previous sections, the LTL predicates

$$goal_1 = \Diamond fact(cleaned)$$
$$goal_2 = \Diamond(fact(on(A, floor)) \wedge fact(on(B,A)) \wedge fact(on(C,B)))$$

where *fact* is a predicate defined on the mental states of either BUnity or BUpL agents in the following way:

$$(\mathcal{B}, p) \models fact(P) \text{ iff } \mathcal{B} \models P.$$

Model-checking that the property $\neg$ ($goal_1 \wedge$ $goal_2$) holds in a state reachable from the initial one returns a counterexample representing the minimal trace *clean* $move(C, floor)$ $move(B,A)$ $move(C,B)$. This execution leads to a state where both $goal_1$and $goal_2$ are satisfied.

2.3 The Refinement of Individual Agents

We have already mentioned in the introduction that we understand BUnity as a typical abstract specification language, and BUpL as an implementation language. Since control aspects are ignored in BUnity models, BUnity is a "highly" non deterministic language. Such non determinism is reduced in BUpL agents. We are interested in the correctness of a BUpL agent with respect to a BUnity agent, or equally stated, in the refinement between a BUpL and a BUnity agent. Refinement is usually defined as trace inclusion, all the traces of the implementation are contained among the traces of the specification.

Definition 2.5 (Refinement). Let $(\mathcal{B}_0, \mathcal{A}, C)$ be a BUnity agent with its initial mental state $\mathcal{B}_0$ and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state $(\mathcal{B}_0, p_0)$. We say that the fair executions of the BUpL agent refine the fair executions of the BUnity agent ($(\mathcal{B}_0, p_0) \subseteq \mathcal{B}_0$) iff every trace of the BUpL agent is also a trace of the BUnity agent, that is $Tr((\mathcal{B}_0, p_0)) \subseteq Tr(\mathcal{B}_0)$.

Being that we are interested only in fair agent executions, we consider also refinement in terms of fair trace inclusion where we take into account the definitions of $just_1$, $just_2$ and *compassionate* as introduced in the previous sections.

Definition 2.6 (Fair Refinement). Let $(\mathcal{B}_0, \mathcal{A}, C)$ be a BUnity agent with its initial mental state $\mathcal{B}_0$ and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state $(\mathcal{B}_0, p_0)$. We say that the fair executions of the BUpL agent refine the fair executions of the BUnity agent ($(\mathcal{B}_0, p_0) \subseteq_f \mathcal{B}_0$) iff every fair trace of the BUpL agent is also a fair trace of the BUnity agent, that is $(\forall tr \in Tr((\mathcal{B}_0, p_0)))\ (\sigma_{tr} \models \mathit{compassionate} \wedge \mathit{just}_2) \Rightarrow (tr \in Tr(\mathcal{B}_0) \wedge \sigma'_{tr} \models \mathit{just}_1)$, where σ_{tr} (resp. σ'_{tr}) is any corresponding computation path in the transition system associated to the BUpL (resp. BUnity) agent.

We note that in the above definitions we have used the same symbols for both initial belief bases ($\mathcal{B}_0$) and sets of action definitions ($\mathcal{A}$). This is not a restriction, it only simplifies the notation.

Proving refinement by definition is not practically feasible because the set of traces may be considerably large. We would need to check that for any solution to matching problems the corresponding trace belongs to both implementation and specification. Instead, refinement is usually proved by means of simulation, which has the advantage of locality of search: one looks for checks at the immediate (successor) transitions that can take place. We recall that the possible transitions for BUpL and BUnity agents are either τ steps (corresponding to choices between plans and repair rules) or steps labelled with action terms. Since we are interested in simulating only visible actions, we refer to weak simulation, which is oblivious with respect to τ steps.

Definition 2.7 (Weak Simulation). Let $(\mathcal{B}_0, \mathcal{A}, C)$ be a BUnity agent with its initial mental state $\mathcal{B}_0$ and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state $(\mathcal{B}_0, p_0)$. Let Σ, Σ' be the sets of mental states for each agent ($\mathcal{B}_0 \in \Sigma$, $(\mathcal{B}_0, p_0) \in \Sigma'$) and let R be a relation, $R \subseteq \Sigma \times \Sigma'$. R is called a weak simulation if, whenever $\mathcal{B}_0\ R\ (\mathcal{B}_0, p_0)$, if $(\mathcal{B}_0, p_0) \stackrel{a}{\Rightarrow} (\mathcal{B}, p)$, then it is also the case that $\mathcal{B}_0 \stackrel{a}{\rightarrow} \mathcal{B}$ and $\mathcal{B}\ R\ (\mathcal{B}, p)$.

Definition 2.8. Let $(\mathcal{B}_0, \mathcal{A}, C)$ be a BUnity agent with its initial mental state $\mathcal{B}_0$ and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state $(\mathcal{B}_0, p_0)$. We say that the BUnity agent weakly simulates the BUpL agent ($(\mathcal{B}_0, p_0) \lesssim \mathcal{B}_0$) if there exists a weak simulation R such that $\mathcal{B}_0\ R\ (\mathcal{B}_0, p_0)$.

We recall that in general simulation is a sound but not necessarily complete proof technique for refinement. We take the classical situation from Figure 2.4 as a counter-example. However, simulation is complete when the simulating system is deterministic (see, for example, [196]). We make the remark that in the case of finite transition systems it is always possible to determinise a non deterministic system by means of a power set construction ([9] for the case of finite traces, and [319, 382] for the case of infinite traces). However, "determinization" is computationally hard ($2^{O(nlogn)}$ in the number of states [382]) and thus usually unfeasible when the focus is on verification.

In our case, the simulating agent is a BUnity one. BUnity agents, though highly non deterministic with respect to control issues, have the property that they are modelled by *deterministic* (see the definition from Section 2.2.1) transition systems. This

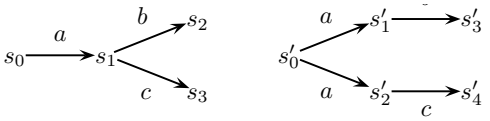

Fig. 2.4 Refinement but not simulation

is because though a BUnity agent makes arbitrary decisions regarding which action to execute, the mental state reflecting the effect of the chosen action is uniquely determined, thus *actions themselves are deterministic*. It follows that, in our framework, simulation is not only a sound but also a complete proof technique for refinement.

We want to reduce the problem of deciding simulation between a BUpL and a BUnity agent to a verification problem. For this, we give a modal characterisation to simulation by constructing the synchronised product of a BUpL and BUnity agent and by defining an LTL property on the states of the product. The property is basically satisfied when the product reaches a deadlock state. We show that it is sufficient and necessary to detect the existence of a deadlocked state in order to prove simulation, and thus refinement.

Definition 2.9 (BUpL-BUnity Synchronised Product). Let $(\mathcal{B}_0, \mathcal{A}, C)$ be a BUnity agent with its initial mental state $\mathcal{B}_0$ and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state $(\mathcal{B}_0, p_0)$. If $I = (\Sigma, (\mathcal{B}_0, p_0), Act \cup \{\tau\}, \rightarrow_1)$ and $S = (\Sigma', \mathcal{B}_0, Act, \rightarrow_2)$ are the corresponding transition systems to the BUpL, resp. BUnity agent, then their left synchronised product is $I \otimes S = (\Sigma \times \Sigma', \langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle, Act, \rightarrow)$. The semantics is given by the following transition rule:

$$\frac{(\mathcal{B}, p) \overset{a}{\Rightarrow}_1 (\mathcal{B}', p') \qquad \mathcal{B} \overset{a}{\rightarrow}_2 \mathcal{B}'}{\langle (\mathcal{B}, p), \mathcal{B} \rangle \overset{a}{\rightarrow} \langle (\mathcal{B}', p'), \mathcal{B}' \rangle}$$

Mathematically, the choice between either first the BUpL agent performs the step and then the BUnity performs the same step or the other way around is insignificant. However, from an implementation point of view, it is better to make the transition rule conditional. Only if the BUpL agent can fire an action the product changes state depending on whether the BUnity agent can mimic the BUpL agent. We say that the BUpL agent drives the simulation. We further say that if the BUnity agent can execute the same action, the product reaches a "good" state. Otherwise, the product is in a deadlocked state.

Definition 2.10 (Deadlock). Let $\perp$ be the property $((\mathcal{B}, p) \overset{a}{\Rightarrow}_1 (\mathcal{B}', p') \wedge \mathcal{B}'' \not\mu\rightarrow_2)$. The state $\langle (\mathcal{B}, p), \mathcal{B}' \rangle$ has a *deadlock* when $\perp$ holds. That is:

$$\langle (\mathcal{B}, p), \mathcal{B}' \rangle \models \perp \text{ iff } (\mathcal{B}, p) \overset{a}{\Rightarrow}_1 (\mathcal{B}', p') \wedge \mathcal{B}'' \not\mu\rightarrow_2.$$

We say that the product is *deadlock-free* if it has no deadlocks.

We note that we make the difference between a deadlocked and a terminal product state, where the only possible transition for the BUpL agent is the idling transition. We further make the remark that, since it basically depends on the BUnity agent being able to perform a certain action, the definition of deadlock introduces *asymmetry* in the executions of the BUpL and BUnity product.

Proposition 2.1. *Let* $(\mathcal{B}_0, \mathcal{A}, C)$ *be a BUnity agent with its initial mental state* $\mathcal{B}_0$ *and let* $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ *be a BUpL agent with its initial mental state* $(\mathcal{B}_0, p_0)$*. We then have that the BUpL agent refines the BUnity agent* $((\mathcal{B}_0, p_0) \subseteq \mathcal{B}_0)$ *iff* $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \models \Box\neg\bot$*, where* $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle$ *is the initial state of the BUpL-BUnity left synchronised product.*

Proof. Since the proof is basically a simplification of the one we present for Theorem 2.1, we leave it to the reader.

In what follows we focus on the "fair" version of Proposition 2.1.

Theorem 2.1. *Let* $(\mathcal{B}_0, \mathcal{A}, C)$ *be a BUnity agent with its initial mental state* $\mathcal{B}_0$ *and let* $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ *be a BUpL agent with its initial mental state* $(\mathcal{B}_0, p_0)$*. We then have that the BUpL agent fairly refines the BUnity agent* $((\mathcal{B}_0, p_0) \subseteq_f \mathcal{B}_0)$ *iff* $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \models$ *compassionate* $\wedge$ *just*$_2 \rightarrow$ *just*$_1 \wedge \Box\neg\bot$*, where* $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle$ *is the initial state of the BUpL-BUnity left synchronised product.*

Proof. We recall that an LTL property holds in a state s if and only if it holds for any computation path σ beginning with s.
"$\Rightarrow$":
Assume that $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \not\models$ *compassionate* $\wedge$ *just*$_2 \rightarrow$ *just*$_1 \wedge \Box\neg\bot$. This means that there exists a computation path (σ, σ') in the BUpL-BUnity product such that $\sigma \models$ *compassionate* $\wedge$ *just*$_2$ (*) and either (1) $\Diamond\bot$ or (2) $\neg$*just*$_1$ holds. From (*) we have that $tr(\sigma)$ is a fair BUpL trace. From the hypothesis $(\mathcal{B}_0, p_0) \subseteq_f \mathcal{B}_0$ we have that there exists a BUnity computation path (which must be σ' since BUnity is deterministic) such that $tr(\sigma) = tr(\sigma')$ and $\sigma' \models$ *just*$_1$ thus (2) cannot be true. Let us now consider (1). In order to have that $\Diamond\bot$ holds for (σ, σ') there must be a deadlocked state $\langle(\mathcal{B}, p), \mathcal{B}\rangle$ on this path. But this implies that there is a fair BUpL trace $tr(\sigma)a$ which does not belong to the set of fair BUnity traces, thus contradicting the hypothesis.
"$\Leftarrow$":
Assume that $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \models$ *compassionate* $\wedge$ *just*$_2$, meaning that any product path is fair with respect to the BUpL path. We make the remark that we do not need to worry about the "vacuity" problem (*compassionate* $\wedge$ *just*$_2$ being always false) since there always exists a fair computation path (any scheduling algorithm will provide one). We now need to prove that $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \models$ *just*$_1$ (*) and $\langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \models \Box\neg\bot$ (**) implies fair refinement. We do this by proving that (**) implies simulation (thus

also refinement). Since we have (*), the product paths are also fair with respect to BUnity.

We now construct a relation R containing all pairs of BUpL and BUnity reachable states and we prove R is a simulation relation. Let $R = \{((\mathcal{B}, p), \mathcal{B}) \mid \langle(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \rightarrow^* \langle(\mathcal{B}, p), \mathcal{B}\rangle\}$. Let $((\mathcal{B}, p), \mathcal{B}) \in R$ s.t. $(\mathcal{B}, p) \stackrel{a}{\Rightarrow} (\mathcal{B}', p')$. It is then the case that also $\mathcal{B} \stackrel{a}{\rightarrow} \mathcal{B}'$ otherwise $\langle(\mathcal{B}, p), \mathcal{B}\rangle \models \bot$. We further need to prove that $\langle(\mathcal{B}', p), \mathcal{B}'\rangle$ is in R. This is, indeed, true since $(\mathcal{B}_0, p_0), \mathcal{B}_0\rangle \rightarrow^* \langle(\mathcal{B}, p), \mathcal{B}\rangle \stackrel{a}{\rightarrow} \langle(\mathcal{B}', p), \mathcal{B}'\rangle$ thus $\langle(\mathcal{B}', p), \mathcal{B}'\rangle$ is a reachable state of the product. □

Remark 2.1. Refinement does not necessarily imply fair refinement. We consider a BUpL agent which can continuously perform only action a while the BUnity specification can additionally perform b. Refinement trivially holds ($\{a^\omega\} \subset \{(a^*b^*)^\omega\}$) however a^ω is unfair with respect to BUnity.

We take, for example, the BUpL and BUnity agents building the *ABC* tower. Any visible action that BUpL executes can be mimicked by the BUnity agent, thus in this case BUnity simulates BUpL and refinement is guaranteed. If we now pose the question whether fair executions of the BUpL agent refine fair executions of the BUnity agent, we have that, conforming to Theorem 2.1, the answer is positive if the formula $(compassionate \wedge just_2) \rightarrow (just_1 \wedge \Box\neg\bot)$ is satisfied in the left product. This is because the traces of the product are of the form $(clean^* \ (move\theta)^*)^\omega$ and thus satisfying the fairness constraints for both BUpL and BUnity.

2.4 Towards Multi-Agent Systems

If previously it was enough to refer to an agent by its current mental state, this is no longer the case when considering multi-agent systems. This is why we associate with each agent an identifier and we consider a multi-agent system as a finite set of such identifiers. We further denote a state of a multi-agent system by $\mathcal{M} = \{(i, ms_i) \mid i \in \mathcal{I}\}$, where $\mathcal{I}$ is the set of agent identifiers and ms_i is a mental state for the agent i. For the moment, we abstract from *what is* the mental state of an agent. The choice of representation is not relevant, we only need to consider that the way to change (update) the mental state of an agent is by performing actions. However, we will instantiate such generic ms_i by either a BUnity or a BUpL mental state whenever the distinction is necessary.

In order to control the behaviour of a multi-agent system we introduce *action-based choreographies*. We understand them as protocols which dictate the way agents behave by imposing ordering and synchrony constraints on their action executions. They represent *exogenous* coordination patterns and they can be seen as an alternative to message passing communication, with the potential advantage of not needing to establish a "common communication language". Choreographies are useful in scenarios where *action synchrony* is more important than *data*.

2.4.1 Action-based Choreographies

For the ease of presentation, we represent choreographies as regular expressions where the basic elements are pairs (i,a). Such pairs denote that the agent i performs the action a. They can be combined by sequence, parallel, choice or Kleene operators, with the usual meaning: $(i_1,a_1);(i_2,a_2)$ models orderings, agent i_1 executes a_1 which is followed by agent i_2 executing o_2; $(i_1,a_1) \parallel (i_2,a_2)$ models synchronisations between actions, agent i_1 executes a_1 while i_2 executes a_2; $(i_1,a_1)+(i_2,a_2)$ models non-deterministic choices, either i_1 executes a_1 or i_2 executes a_2; $(i,a)^*$ models iterated execution of a by i. The operators respect the usual precedence relation[2]. The BNF grammar defining a choreography is as follows:

$$c ::= (id,a) \mid c+c \mid c \parallel c \mid c;c \mid c^*$$

In order to describe the transitions of a multi-agent system in the presence of a choreography c, we first associate a transition system $\mathcal{S}^c$ to the choreography. We do this in the usual way, inductively on the size of the choreography such that the labels are of the form $\|_{i\in\mathcal{I}}\ (i,a_i)$. Such a transition system always exists (see [239] or [90] for a direct deterministic construction using the derivatives of a given regular expression).

We take, for instance, the transition system from Figure 2.5 which is associated with the choreography c defined as the following regular expression:

$$\begin{aligned} c = {} & (i_1, clean) \parallel (i_2, move(C,A,floor)); \\ & (i_1, move(B,floor,A)); ((i_1, move(B,floor,A)) \parallel (i_2, clean)) + \\ & (i_2, move(B,floor,A)); ((i_2, move(B,floor,A)) \parallel (i_1, clean)). \end{aligned}$$

The choreography specifies that two agents i_1, i_2 work together in order to build the tower ABC and that furthermore, while one is building the tower the other one is cleaning the floor. More precisely, the definition of the choreography says that first i_2 deconstructs the initial tower (by moving the block C on $floor$) while i_1 is synchronously cleaning; next, either i_1 constructs the final tower while i_2 cleans or the other way around; afterwards, the system is in a final state. Further variations (like for example, in the case of a higher tower, one agent builds an intermediate shorter tower leaving the other to finish the construction) are left to the imagination of the reader.

We denote by $\mathcal{S}^c \otimes \mathcal{I}$ the synchronised product of a choreography c and a multi-agent system $\mathcal{I}$. The states of $\mathcal{S}^c \otimes \mathcal{I}$ are pairs $(cs,\mathcal{M})$ where cs is a state of $\mathcal{S}^c$ and $\mathcal{M}$ is a state of the multi-agent system $\mathcal{I}$. The transition rule for $\mathcal{S}^c \otimes \mathcal{I}$ is given as follows:

$$\frac{cs \xrightarrow{l} cs' \qquad \bigwedge_{j\in\mathcal{J}} ms_j \overset{a_j}{\Rightarrow} ms'_j}{(cs,\mathcal{M}) \xrightarrow{l} (cs',\mathcal{M}')}(mas)$$

[2] If we denote $\leq_p$ the precedence relation, then we have '+' $\leq_p$ '$\|$' $\leq_p$ ';' $\leq_p$ '*'

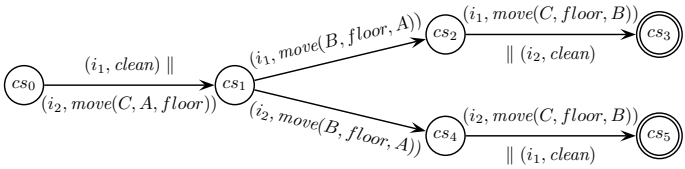

Fig. 2.5 The LTS associated to a choreography

where cs, cs' are states of $\mathcal{S}^c$, l is a choreography label of the form $\|_{j\in\mathcal{J}}\ (j, a_j)$ with $\mathcal{J}$ being a subset of $\mathcal{I}$, ms_j, ms'_j are mental states of agent j and $\mathcal{M}, \mathcal{M}'$ are states of the multi-agent system with $\mathcal{M}'$ being $\mathcal{M} \setminus \{(j, ms_j) \mid j \in \mathcal{J}\} \cup \{(j, ms'_j) \mid j \in \mathcal{J}\}$. The notation $ms_j \overset{a_j}{\Rightarrow} ms'_j$ is used to denote that agent j performs action a_j (eventually with τ steps) in ms_j resulting in ms'_j. "Eventually τ steps" is needed for agents performing internal actions, like making choices among plans or handling failures in the case of BUpL agents. In the case of agents "in the style of BUnity", $\overset{a}{\Rightarrow}$ is simply $\overset{a}{\rightarrow}$ since Bunity agents do not have τ steps.

The transition rule (*mas*) says that the multi-agent system advances one step when the agents identified by $\mathcal{J}$ perform the actions specified by the label l. The new state of the multi-agent system reflects the updates of the mental states of the individual agents.

2.4.2 A Finer Notion of Refinement

We would like to have the result that if the agents (for example BUpL) in a multi-agent system $\mathcal{I}_1$ are refining the (BUnity) agents in $\mathcal{I}_2$ then $\mathcal{S}^c \otimes \mathcal{I}_1$ is a refinement of $\mathcal{S}^c \otimes \mathcal{I}_2$. When refinement is defined as trace inclusion, this is, indeed, the case, as we can shortly prove in Proposition 2.2.

Proposition 2.2. *Given two multi-agent systems $\mathcal{I}_1$, $\mathcal{I}_2$ such that $(\forall i_1 \in \mathcal{I}_1)(\exists i_2 \in \mathcal{I}_2)$ $(ms_{i_1} \subseteq ms_{i_2})$ and a choreography as $\mathcal{S}^c$ we have that $\mathcal{S}^c \otimes \mathcal{I}_1 \subseteq \mathcal{S}^c \otimes \mathcal{I}_2$.*

Proof. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the initial states of the multi-agent systems $\mathcal{I}_1$ and $\mathcal{I}_2$. Let also cs_0 be the initial state of the transition system $\mathcal{S}^c$ associated to the choreography c. It is enough to notice that $Tr((cs_0, \mathcal{M}_1)) = Tr(cs_0) \cap Tr(\mathcal{M}_1)$ and that $ms_{i_1} \subseteq ms_{i_2}$ for all $i_1 \in \mathcal{I}_1$ implies $Tr(\mathcal{M}_1) \subseteq Tr(\mathcal{M}_2)$.

However, adding choreographies to a multi-agent system may introduce deadlocks. On the one hand, we would like to be able to infer from the semantics when

a multi-agent system is in a deadlock state. On the other hand, we would like to have that the refinement of multi-agent systems does not introduce deadlocks. Trace semantics is a too coarse notion with respect to deadlocks. There are two consequences: neither is it enough to define the semantics of a multi-agent system as the set of all possible traces, nor is it satisfactory to define agent refinement as trace inclusion. We further illustrate these affirmations by means of simple examples.

We take, for instance, the choreography $c = (i, move(B, floor, C))$, where i symbolically points to the BUpL agent from Section 2.2.5. Looking at the plans and repair rules of the BUpL agent we see that such an action cannot take place. Thus, conforming to the transition rule (*mas*), there is no possible transition for the product $\mathcal{S}^c \otimes \mathcal{I}$. Just by analysing the behaviour (the empty trace) we cannot infer anything about deadlocked states: is it that the agent has no plan, or is it that the choreography asks for an impossible execution? This is the reason why, in order to distinguish between successful and deadlocked executions, we explicitly define a transition label $\surd$ different from any other action relations. We then define for the product $\mathcal{S}^c \otimes \mathcal{I}$ an operational semantics $O^{\surd}(\mathcal{S}^c \otimes \mathcal{I})$ as the set of maximal (in the sense that no further transition is possible) traces, ending with $\surd$ when the last state is successful:

$$\{tr\surd \mid (cs_0, \mathcal{M}_0) \xrightarrow{tr} (cs, \mathcal{M}) \nrightarrow, cs \in F(\mathcal{S}^c)\} \cup$$
$$\{tr \mid (cs_0, \mathcal{M}_0) \xrightarrow{tr} (cs, \mathcal{M}) \nrightarrow, cs \notin F(\mathcal{S}^c)\} \cup \{\epsilon \mid (cs_0, \mathcal{M}_0) \nrightarrow\},$$

where tr is a trace with respect to the transition rule (*mas*), $\mathcal{M}_0$ (resp. cs_0) is the initial state of $\mathcal{I}$ (resp. $\mathcal{S}^c$) and ϵ denotes that there are no possible transitions from the initial state.

We can now define the refinement of multi-agent systems with respect to the above definition of the operational semantics O.

Definition 2.11 (MAS Refinement). Given a choreography c, we say that two multi-agent systems $\mathcal{I}_1$ and $\mathcal{I}_2$ are in a refinement relation if and only if the set of maximal traces of $\mathcal{S}^c \otimes \mathcal{I}_1$ are included in the set of maximal traces of $\mathcal{S}^c \otimes \mathcal{I}_2$. That is, $O^{\surd}(\mathcal{S}^c \otimes \mathcal{I}_1) \subseteq O^{\surd}(\mathcal{S}^c \otimes \mathcal{I}_2)$.

We now approach the problem that appears when considering agent refinement defined as trace inclusion. It can be the case that the agents in the concrete system refine (with respect to trace inclusion) the agents in the abstract system, nevertheless the concrete system deadlocks for a particular choreography. We take, for instance, the BUnity and BUpL agents from Sections 2.2.3 and 2.2.5. For the ease of reference, we identify the BUnity agent by i_a (since it is more abstract) and the BUpL agent by i_c (since it is more concrete). We can easily design a choreography which works fine with i_a (does not deadlock) and on the contrary, it deadlocks with i_c. Such a choreography is for example the one mentioned in the beginning of the section, $c = (i, move(B, floor, C))$, where, i points now to either i_a or i_c up to a renaming. We recall that i_c is a refinement of i_a. However, we have already mentioned, i_c cannot execute the move (since the move is irrelevant for building the ABC tower and at

implementation time it matters to be as precise as possible), while i_a can (since in a specification "necessary" is more important than "sufficiency").

What the above illustration implies is that refinement as trace inclusion, though being a satisfactory definition at individual agent level, is not a strong enough condition to ensure refinement at a multi-agent level, in the presence of an arbitrary choreography. It follows that we need to redefine individual agent refinement such that multi-agent system refinement (as maximal trace inclusion) is compositional with respect to any choreography. In this sense, a choreography is more like a context for multi-agent systems, meaning that whatever the context is, it should not affect the visible results of the agents' executions but restrict them by activating only certain traces (the other traces still exist, however, they are inactive).

In order to have a proper definition of agent refinement we look for a finer notion of traces. The key ingredient lies in *enabling* conditions for actions. Given a mental state ms, we look at all the actions enabled to be executed from ms. We denote them by $E(ms) = \{a \in \mathcal{A} \mid \exists ms'(ms \xrightarrow{a} ms')\}$ and we call $E(ms)$ a *ready* set. We can now present *ready traces* as possibly infinite sequences $X_1, a_1, X_2, a_2, \ldots$ where $ms_0 \xrightarrow{a_1} ms_1 \xrightarrow{a_2} ms_2 \ldots$ and $X_{i+1} = E(ms_i)$. We denote the set of all ready traces from a state ms_0 as $RT(ms_0)$. Compared to the definition of traces, ready traces are a much more finer notion in the sense that they record not only actions which have been executed but also sets of actions which are *enabled* to be executed at each step.

Definition 2.12 (Ready Agent Refinement). We say that two agents with initial mental states ms and ms' are in a ready refinement relation (i.e., $ms \subseteq_{rt} ms'$) if and only if the ready traces of ms are included in the ready traces of ms' (i.e., $RT(ms) \subseteq RT(ms')$).

We can now present our main result which states that refinement is compositional, in the sense that if there is a ready refinement between the agents composing two multi-agent sytems it is then the case that one multi-agent system refines the other in the presence of any choreography.

Theorem 2.2. *Let $\mathcal{I}_1$, $\mathcal{I}_2$ be two multi-agent systems such that $(\forall i_1 \in \mathcal{I}_1)\ (\exists i_2 \in \mathcal{I}_2)$ $(ms_{i_1} \subseteq_{rt} ms_{i_2})$ and a choreography c with the associated LTS $\mathcal{S}^c$. We have that $\mathcal{I}_1$ refines $\mathcal{I}_2$, that is, $O^{\surd}(\mathcal{S}^c \otimes \mathcal{I}_1) \subseteq O^{\surd}(\mathcal{S}^c \otimes \mathcal{I}_2)$.*

Proof. What we need to further prove with respect to Proposition 2.2 is that the set of enabled actions is a key factor in identifying failures in both implementation and specification. Assume a maximal trace tr in $O^{\surd}(\mathcal{S}^c \otimes \mathcal{I}_1)$ leading to a non final choreography state cs. Given cs_0 and $\mathcal{M}_1$ as the initial states of $\mathcal{S}^c$, $\mathcal{I}_1$, we have that $(cs_0, \mathcal{M}_1) \xrightarrow{tr} (cs, \mathcal{M})$ $(cs, \mathcal{M}) \not\xrightarrow{l}$ for all $l = \|_{j \in \mathcal{J}} (j, a_j)$ such that $cs \xrightarrow{l} cs'$. By rule (mas) this implies that there exists an agent identified by j which cannot perform the action indicated. Thus the corresponding trace of j ends with a ready set X with the property that a_j is not included in it. We know that each implementation agent has a corresponding specification, be it j', such that j ready refines j'. If we, on the

other hand, assume that j' can, on the contrary, execute a_j we would have that in a given state j' has besides the ready set X another ready set Y which includes a_j. This contradicts the maximality of the ready set. □

As a direct consequence of the above theorem, we are able to infer the absence of deadlock in the concrete system from the absence of deadlock in the abstract one:

Corollary 2.1. *Let $\mathcal{I}_1$, $\mathcal{I}_2$ be two multi-agent systems with initial states $\mathcal{M}_1$ and $\mathcal{M}_2$. Let c be a choreography with the associated LTS $\mathcal{S}^c$ and initial state cs_0. We have that if $\mathcal{I}_1$ refines $\mathcal{I}_2$ ($O^{\surd}(\mathcal{S}^c \otimes \mathcal{I}_1) \subseteq O^{\surd}(\mathcal{S}^{c'} \otimes \mathcal{I}_2)$) and c does not deadlock the specification ($(cs_0, \mathcal{M}_2) \models \Box\neg\bot$) it is then also the case that c does not deadlock the implementation ($(cs_0, \mathcal{M}_1) \models \Box\neg\bot$).*

As we have already explained in Section 2.3, proving refinement by deciding trace inclusion is an inefficient procedure. This is also the case with ready refinement, thus a more adequate approach is needed. If previously we have adopted simulation as a proof technique for refinement, now we consider *weak ready simulation*.

Definition 2.13 (Weak Ready Simulation). We say that two agents with initial mental states ms and ms' are in a (weak) ready simulation relation ($ms \lesssim_{rs} ms'$) if and only if $ms \lesssim ms'$ and the corresponding ready sets are equal ($E(ms) = E(ms')$).

As it is the case for simulation being a sound and complete proof technique for refinement, analogously we can have a similar result for ready simulation. We recall that determinacy plays an important role in the proof for completeness.

Proposition 2.3. *Given two agents with initial mental states ms and ms′, where the one with ms is deterministic, we have that $ms \lesssim_{rs} ms'$ iff $ms' \subseteq_{rt} ms$.*

Remark 2.2. For the sake of generality, in the definitions from this section we have used the symbolic notations ms, ms'. BUnity and BUpL agents can be seen as (are, in fact) instantiations. Proposition 2.3 relates to Proposition 2.1. The only difference is that, for simplification, Proposition 2.3 refers directly to ready simulation and not to its modal characterisation, as it was the case for simulation in Proposition 2.1. It is not difficult to adapt Definition 2.9 to the ready simulation. One needs only to change the condition on the transition (mas) from $(\mathcal{B}, p) \stackrel{a}{\Rightarrow}_1 (\mathcal{B}', p')$ to the conjunction $(\mathcal{B}, p) \stackrel{a}{\Rightarrow}_1 (\mathcal{B}', p') \wedge E((\mathcal{B}, p)) = E((\mathcal{B}))$ which checks also the equality on the ready sets.

Recalling the BUpL and BUnity agents i_c and i_a, we note that though i_a simulates i_c it is not also the case that it ready simulates. This is because the ready set of the BUnity agent is always larger than the one of the BUpL agent. One basic argument is that i_a can always "undo a block move", while i_a cannot. However, let us see what would have happened if we were to consider changing i_a by replacing the conditional action set from Figure 2.1 with the set from Figure 2.6:

$$C = \{\ \neg on(B,A) \triangleright do(move(B,floor,A)),$$
$$\neg on(C,B) \wedge on(B,A) \triangleright do(move(C,floor,B)),$$
$$\neg(on(B,A) \wedge on(C,B)) \triangleright do(move(X,Y,floor))\ \}$$

Fig. 2.6 Adapting i_a to ready simulate i_c

We now have a BUnity agent which is less abstract. Basically, the instantiation from the first two conditional actions disallows any spurious "to and fro" sequence of moves like $move(X,Y,Z)$ followed by $move(X,Z,Y)$ which practically undoes the previous step leading to exactly the previous configuration. The instantiation is obvious when one looks at the final "desired" configuration. The last conditional action allows "destructing" steps by moving blocks on the floor. It can still be considered as a specification. It provides no information about the order of executing the moves since this is not important at the abstraction level. With the above change, the new BUnity agent ready simulates i_c. To see this, it suffices to notice that the only BUpL ready trace is $\{move(C,A,floor)\}$, $move(C,A,floor)$, $\{move(B,floor,A)\}$, $move(B,floor,A)$, $\{move(C,floor,B)\}$, $move(C,floor,B)$ which is also the only BUnity ready trace. The same equality of ready traces holds when we consider the additional *clean* action.

We recall the choreography from Figure 2.5 and we consider a BUnity multi-agent system which consists of two copies of i_a (enabled to execute also *clean*). For either branch, the executions (with respect to the transition (*mas*)) of the multi-agent system are successful (the choreography reaches a final state). Since i_c ready refines i_a, by Corollary 2.1 we can deduce that also the executions of a multi-agent system which consists of two i_c copies are successful.

2.5 Timing Extensions of MAS

Modelling time in multi-agent systems make them more expressive. For example, timing constraints can be used to enforce delays between actions and to time restrict action execution, that is, to force action execution to happen before certain time invariants are violated. Our approach in adding time to multi-agent systems consists of adapting the theory of timed-automata [10]. A *timed automaton* is a finite transition system extended with real-valued clock variables. Time advances only in states since transitions are instantaneous. Clocks can be reset at zero simultaneously with any transition. At any instant, the reading of a clock equals the time elapsed since the last time it was reset. States and transitions have *clock constraints*, defined as:

$$\phi_c ::= x \leq t \mid t \leq x \mid x < t \mid t < x \mid \phi_c \wedge \phi_c,$$

where $t \in \mathbb{Q}$ is a constant and x is a clock. When a clock constraint is associated with a state, it is called *invariant*, and it expresses that time can elapse in the state as long as the invariant stays true. When a clock constraint is associated with a transition,

it is called *guard*, and it expresses that the action may be taken only if the current values of the clocks satisfy the guard.

In our multi-agent setting, *timed choreographies* are meant to impose time constraints on the actions executed by the agents. We model them as timed automata. We take, as an example, the choreography from Figure 2.7. There is a single clock x. The initial state cs_0 has no invariant constraint and this means that an arbitrary amount of time can elapse in cs_0. The clock x is always reset with the transition from cs_0 to cs_1. The invariant $x < 5$ associated with the state cs_1 ensures that the synchronous actions *clean* and $move(C, A, floor)$ must be executed within 5 units of time. The guard $x > 6$ associated with the transition from cs_2 to cs_3 ensures that the agents cannot spend an indefinite time in cs_2 because they must finish their tasks after 6 units of time.

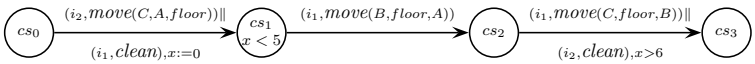

Fig. 2.7 A timed choreography

We now approach the issue of modelling time in BUnity and BUpL agents. In this regard, we consider that agents have a set of local clocks and that clock valuations can be performed by an observer. We further pose the problem of how agents make use of clocks. We recall the design principle: "the specification of basic actions does not come with time", thus actions are instantaneous. This implies that, in order to make the time pass, we need to extend the syntax of the agent languages with new application specific constructions such that the ontology of basic actions remains timeless (basic actions being specified only in terms of pre/post conditions). This is why we introduce *delay actions*, $\phi \rightarrow I$, where ϕ is a query on the belief base and I is an invariant like $x \leq 1$. Basically, their purpose is to make time elapse in a mental state where certain beliefs hold. As long as the invariant is true, the agent can stay in the same state while time passes. We refer to $\mathcal{D}$ as the set of delays of either a BUnity or a BUpL agent. This is because, as it is the case for basic actions, delays are syntactical constructions belonging to both BUpL and BUnity languages. In what follows, we discuss the time extension for each language separately.

2.5.1 Adding Time to BUnity

We now focus on time extending BUnity conditional actions. First, the queries of conditional actions are defined both on belief bases and clock valuations. Second, conditional actions specify the set of clocks to be reset after the execution of basic actions. Their syntax becomes $\{\phi \wedge \phi_c\} \triangleright do(a), \lambda$. Timed conditional actions are

meant to say that if certain beliefs ϕ hold in the current mental state of a BUnity agent (as before) and additionally, certain clock constraints ϕ_c are satisfied, then the basic action a is executed and the clocks from the set λ are reset to 0. Taking into account the previous discussion of the mechanism of delay actions, the corresponding changes in the semantics are reflected in Figure 2.8: where λ is the set of clocks reset

$$\frac{\phi \to I \qquad \mathcal{B} \models \phi \qquad (\forall \delta \in \mathbb{R}_+)(\nu + \delta \in I)}{\mathcal{B}, \nu \xrightarrow{\delta} \mathcal{B}, \nu + \delta} \ (delay) \qquad \frac{\{\phi \wedge \phi_c\} \triangleright do(a), \lambda \qquad a = (\varphi, \xi) \qquad \theta \in Sols(\mathcal{B} \models (\phi \wedge \varphi)) \qquad \nu \in \phi_c}{\mathcal{B}, \nu \xrightarrow{a\theta} \mathcal{B} \uplus \xi\theta, \nu[\lambda := 0]} \ (act)$$

Fig. 2.8 Transition Rules for Timed BUnity

by performing action a and ν represents the current clock valuations. We use the notation $\nu \in I$ (resp. $\nu \in \phi_c$) to say that the clock valuations from ν satisfy the invariant I (resp. the constraint ϕ_c). When ϕ_c is absent we consider that trivially $\nu \in \phi_c$ holds. We make a short note that our design decision is to separate the implementation of delays from the one of conditional actions. This is because a construction like $\{\phi\} \triangleright I, do(a), \lambda$ is ambiguous. If ϕ holds, it can either be the case that time elapses with respect to the invariant I and a is suspended, or that a is immediately executed. However, it sometimes is important to *ensure* that "time passes in a state", instead of leaving this only as a non deterministic choice.

To illustrate the above constructions we recall the BUnity agent i_a from Figure 2.6. We basically extend the BUnity agent such that the agent has one clock, be it y, which is reset by conditional actions, and such that the agent can delay in given states, thus letting the time pass.

$$\begin{aligned} \mathcal{C} = \{\ & \top \triangleright (do(clean), y := 0),\ \neg on(B,A) \triangleright do(move(B, floor, A)), \\ & \neg on(C,B) \wedge on(B,A) \triangleright do(move(C, floor, B)), \\ & \neg(on(B,A) \wedge on(C,B)) \triangleright (do(move(X,Y, floor)), y := 0)\ \} \end{aligned}$$

$$\mathcal{D} = \{\ on(C, floor) \vee cleaned \leftarrow (y < 9),\ on(B,A) \vee cleaned \leftarrow (y < 10)\ \}$$

Fig. 2.9 Extending i_a with clock constraints

Figure 2.9 shows a possible timed extension. The clock y is reset after either performing *clean* or moving a block on the floor. The agent can delay until the clock valuates to 9 (resp. 10) units of time after moving C on the $floor$ (resp. B on A).

2.5.2 Adding Time to BUpL

The timed extension of BUpL concerns changing plans such that previous calls $a; p$ are replaced by $(\phi_c, a, \lambda); p$ and $(\phi \rightarrow I); p$, where ϕ_c is time constraining the execution of action a and λ is the set of clocks to be reset. To simplify notation, if clock constraints and clock resets are absent we use a instead of (a).

We make the remark that if previously actions failed when certain beliefs did not hold in a given mental state, it is now the case that actions fail also when certain clock constraints are not satisfied. Consider, for example, the plan $((x < 1), a, [x := 0]); ((x > 2), b, \emptyset)$. There is no delay action between a and b, thus the time does not pass and x remains 0, meaning that b cannot be executed. Such situations are handled by means of the general semantics of the repair rules. There are two possibilities: either to execute an action with a time constraint that holds, or to make time elapse. The latter is achieved by triggering a repair rule like $true \leftarrow \delta$, where for example δ is a delay action $true \rightarrow true$ which allows an indefinite amount of time to pass. The corresponding changes in the semantics are reflected in Figure 2.10:

$$\frac{p = (\phi \rightarrow I); p' \qquad \mathcal{B} \models \varphi \qquad (\forall \delta \in \mathbb{R}_+)(\nu + \delta \in I)}{\mathcal{B}, p, \nu \xrightarrow{\delta} \mathcal{B}, p', \nu + \delta} \ (delay) \qquad \frac{p = (\phi_c, a, \lambda); p' \qquad a = (\varphi, \xi) \qquad \theta \in Sols(\mathcal{B} \models \varphi) \qquad \nu \in \phi_c}{\mathcal{B}, p, \nu \xrightarrow{a\theta} \mathcal{B} \uplus \xi\theta, p', \nu[\lambda := 0]} \ (act)$$

Fig. 2.10 Transition Rules for Timed BUpL

To see a concrete example, we recall the BUpL agent from Section 2.2.6. We consider two delay actions $true \rightarrow (y < 9)$ and $true \rightarrow (y < 10)$. We further make the delays and the clock resets transparent in the plans. The plan *cleanR* changes to $(true, clean, y := 0)$;*cleanR* such that the clock y is reset after a clean action. The plan $rearrange(x_1, x_2, x_3)$ changes to $true \rightarrow (y < 9)$;$move(x_1, floor, x_2)$; $true \rightarrow (y < 10)$; $move(x_3, floor, x_1)$ such that time passes between moves.

The observable behaviour of either timed BUnity or BUpL agents is defined in terms of timed traces. A *timed trace* is a (possibly infinite) sequence $(t_1, a_1)\ (t_2, a_2)\ \ldots (t_i, a_i)\ \ldots$ where $t_i \in \mathbb{R}_+$ with $t_i \leq t_{i+1}$ for all $i \geq 1$. We call t_i a *time-stamp* of action a_i since it denotes the absolute time that passed before a_i was executed. We then have that a timed BUnity or BUpL agent computation over a timed trace $(t_1, a_1)(t_2, a_2)\ldots(t_i, a_i)\ldots$ is a sequence of transitions:

$$ms_0, \nu_0 \xrightarrow{\delta_1}\xrightarrow{a_1} ms_1, \nu_1 \xrightarrow{\delta_2}\xrightarrow{a_2} ms_2, \nu_2 \ \ldots$$

where ms_i is a BUnity (BUpL) mental state and t_i are satisfying the condition $t_i = t_{i-1} + \delta_i$ for all $i \geq 1$.

For example, a possible timed trace for either the timed BUpL or BUnity agent is $(0, clean), (7, move(C,A,floor)), (8, move(B,floor,A)), (9, move(C,floor,B))$. It is, in fact, the case that any BUpL timed trace is also a BUnity timed trace, thus the two agents are again in a refinement relation. We elaborate more on timed refinement in the next section.

2.5.3 A Short Note on Timed Refinement

We recall that a key element in having simulation as a proof technique for individual agent refinement was the determinacy of BUnity agents. We note that in order to have a similar "timed" result we only need to impose the restriction that clock constraints associated with the same action must be disjoint. This ensures determinacy of timed automata. A weaker restriction (which nevertheless requires a "determinisation" construction) is to require that *each basic action is associated with at most one clock* and that *conditional actions can only reset the clock corresponding to the basic action being executed*; however, the guards in conditional actions may consult different clocks. Under the disjointness condition, we have that timed BUnity agents are deterministic, thus the same proof technique as in Section 2.3 can be applied. We make the remark that *timed simulation* differs from simulation in only one aspect: we further need to consider simulating δ steps and not only a steps.

As for multi-agent system refinement, we recall that in Section 2.4.1 we considered that agents are associated with identifiers. We now need to consider that agents have also a set of clocks to manipulate. We thus update the definition of a multi-agent state $\mathcal{M}$ as being $\{(i, ms_i, \nu_i) \mid i \in \mathcal{I}\}$, where $\mathcal{I}$ is the set of agent identifiers, ms_i is a mental state for the agent i and ν_i represents the clock valuations of i.

We further recall that we gave semantics to multi-agent systems by means of a transition rule for the synchronised product of the system and a given choreography. Adding time constructions to both agent languages and to the choreography needs to be reflected by changing the transition rule (*mas*) correspondingly. More precisely, we first need a transition for passing time which corresponds to delays in both choreography and agent states. We then need to change the rule (*mas*) such that a transition of the system is enabled not only if certain agents can perform the actions specified by the choreography, but also if the clock valuations of the acting agents satisfy the clock constraints of the choreography. The changes are illustrated in Figure 2.11, where l is $\|_{j \in \mathcal{J}} (j, a_j)$, g is the clock constraint associated with the label l, $I(cs)$ denotes the invariant associated with the state cs and $\mathcal{M}'$ is $\mathcal{M} \setminus \{(j, ms_j, \nu_j) \mid j \in \mathcal{J}\} \cup \{(j, ms'_j, \nu'_j) \mid j \in \mathcal{J}\}$.

We note that the transition (*delay*) can take place only if all agents are able to delay. No deadlocks are introduced since delays are not compulsory.

Similarly as in Section 2.4.1 the semantics of a timed agent system together with a timed choreography is defined as the set of maximal *timed* computations where

$$\frac{\bigwedge_{i\in\mathcal{I}} ms_i, \nu_i \xrightarrow{\delta} ms_i, \nu_i + \delta \qquad \bigwedge_i \nu_i + \delta \in I(cs)}{((cs,\nu),\mathcal{M}) \xrightarrow{\delta} ((cs,\nu+\delta),\{(i,ms_i,\nu_i+\delta)\})} \quad \text{if } cs,\nu \xrightarrow{\delta} cs,\nu+\delta \ \ (delay)$$

$$\frac{\bigwedge_{j\in\mathcal{J}} ms_j, \nu_j \overset{a_j}{\Rightarrow} ms'_j, \nu'_j \qquad \nu\nu_j \in g \qquad \nu'\nu'_j \in I(cs')}{((cs,\nu),\mathcal{M}) \xrightarrow{l} ((cs',\nu'),\mathcal{M}')} \quad \text{if } cs,\nu \xrightarrow{l,g} cs',\nu' \ \ (t\text{-}mas)$$

Fig. 2.11 Transition Rules for a Timed Agent System

we make the distinction between a success and a deadlock. This was needed (and it still is) in order to reason about deadlock. We recall that we further needed to change refinement (resp. simulation) to ready refinement (resp. ready simulation). The reason was that in order to have compositionality of multi-agent systems refinement one needs to make sure that any choreography which does not deadlock the abstract system cannot deadlock the concrete one. The same reasoning applies in the case of timed agent systems. We need to investigate if further deadlock situations can arise which are not taken into account in the framework of ready refinement. As we have already mentioned, the transition (*delay*) does not introduce deadlock situations. Thus, we only need to consider the transition (*t-mas*). What is new with respect to the previous transition (*mas*) is the additional condition which asks that the clock valuations satisfy the guard of the current transition in the choreography. We note that the fact that a clock constraint in the implementation does not satisfy the guard of the choreography at a given time is visible at the level of the timed ready sets. Thus, the timed version of ready refinement suffices in order to have compositionality of timed agent systems refinement. Baring this in mind, we only need to focus on timed ready simulation as being a proof technique for timed agent systems refinement, which previously was a valid statement as a consequence of the determinacy of the choreography. Thus, along the same line as before, we only need to require that clock constraints associated with the same action are disjoint such that timed choreographies are deterministic.

As an observation, a timed BUpL multi-agent system consisting of two instances of the agent described in Section 2.5.2 running under the timed choreography from Figure 2.7 is a timed refinement of a timed BUnity multi-agent system consisting of two instances of the agent described in Section 2.5.1 running under the same choreography. Furthermore, both systems are deadlock free. To illustrate this latter affirmation, we present a small experiment in UPPAAL [31], a tool for verifying timed automata. At a more abstract and syntactic level, we have modelled the timed choreography and the timed BUpL agent as timed automata in UPPAAL. We have then verified that the value of the clocks are always greater than 6. This implies that the choreography always reaches the final state. Figure 2.12 illustrates the timed BUpL system consisting of two instances of the BUpL agent and the choreography. The BUpL agent is parametrised by `id_b`, a bounded integer variable which is in our case 0 or 1. We note that we had to "approximate" and implement the parallel

operator using an interleaving mechanism (first one agent cleans and after the other one moves *C* on the floor). The synchronisation between the choreography and the BUpL agents is in the CCS style (e.g., `clean[1-e]!` and `clean[1-e]?`).

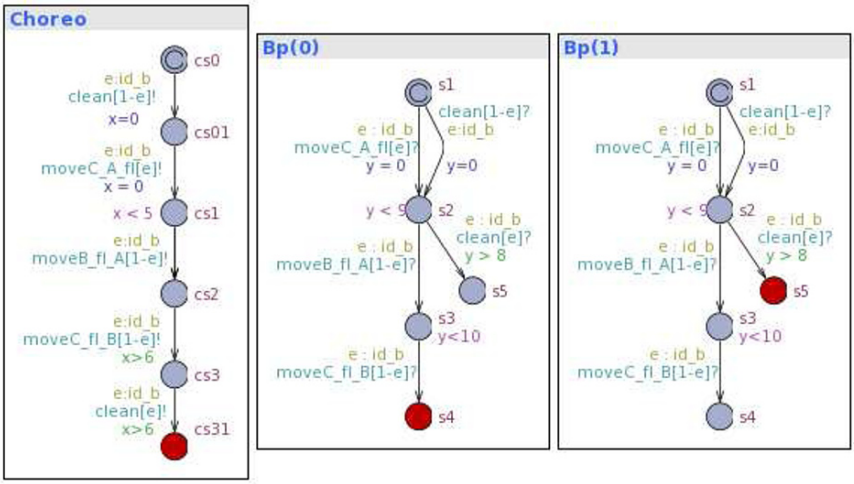

Fig. 2.12 A Timed BUpL System Modelled in UPPAAL

Using the UPPAAL `Simulate` command one can experiment with different timed executions of the system. Figure 2.13 represents one of them. The trace shows that the BUpL instance `Bp(0)` is the first to execute clean followed by `Bp(1)` executing the destructing step (*C* on the floor). Fom this point `Bp(0)` finishes the *ABC* tower. Finally, `Bp(1)` executes, at its turn, the action clean.

2.6 Conclusion

We have addressed the problem of multi-agent system refinement. We have first focused on individual agent refinement where we relied on fair simulation as a proof technique for fair trace inclusion. We have then extended the notion of refinement of individual agents to multi-agent systems, where the behaviour of the agents composing the systems is coordinated by choreographies. Our approach to introducing choreographies to multi-agent systems consisted of defining them as action-based coordination mechanisms. In such a framework, we have the results that agent refinement is a sufficient condition for multi-agent system refinement and that this latter notion preserves deadlock freeness. We have further illustrated a timed extension of multi-agent systems by means of timed automata where the same refinement methodology can be adapted.

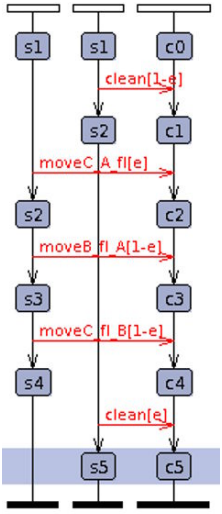

Fig. 2.13 A Resulting Timed Trace

We have stressed the importance of verification from the introduction. Our goal was to describe a general methodology for a top-down design of multi-agent systems which makes it simple to execute and verify agent programs. Concerning this practical side we mention that we have already implemented our formalism in Maude. Maude is an encompassing framework where we prototyped the agent languages we described such that it is possible to (1) execute agents by rewriting; (2) verify agents by means of simulation, model-checking, searching, or testing. Since we were mainly interested in refinement, the properties we focused on were correctness properties, i.e., model-checking for the absence of deadlock in the product of a BUpL and BUnity agent. However, we have experimented with different other safety and liveness properties. In this regard, please see Chapter [421] for further details and references. The current version of the implementation (also including the timed languages prototyped in Real-Time Maude [326]) can be found at `http://homepages.cwi.nl/~astefano/agents`. Further extensions with respect to model-checking timed agents and automatically generating test cases for verifying infinite state agents need to be investigated.

Chapter 3
Model Checking Agent Communication

J. Bentahar, J.-J. Ch. Meyer, and W. Wan

Abstract Model checking is a formal and automatic technique used to verify computational systems (e.g. communication protocols) against given properties. The purpose of this chapter is to describe a model checking algorithm to verify communication protocols used by autonomous agents interacting using dialogue games, which are governed by a set of logical rules. We use a variant of Extended Computation Tree Logic CTL* for specifying these dialogue games and the properties to be checked. This logic, called ACTL*, extends CTL* by allowing formulae to constrain actions as well as states. The verification method uses an on-the-fly efficient algorithm. It is based on translating formulae into a variant of alternating tree automata called Alternating Büchi Tableau Automata (ABTA). We present a tableau-based version of this algorithm and provide the soundness, completeness, termination and complexity results. Two case studies are discussed along with their respective implementations to illustrate the proposed approach. The first one is about an agent-based negotiation protocol and the second one considers a modified version of the NetBill protocol.

J. Bentahar, W. Wan
Concordia University, Concordia Institute for Information Systems Engineering, Canada e-mail: bentahar@ciise.concordia.ca

J.-J. Ch. Meyer
Utrecht University, Department of Computer Science, The Netherlands e-mail: jj@cs.uu.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_3,

3.1 Introduction

Model checking is a formal verification method widely used to check complex systems involving concurrency and communication protocols by verifying some desirable properties. *Deadlock-freedom* (it is false that two or more processes are each waiting for another to release a resource), *safety* (some bad situation may never occur), and *reachability* (some particular situation can be reached) are examples of such properties. Model checking techniques offer the possibility of obtaining an early integration of verification in the design process and reducing the verification time. However, they are only applicable for finite state systems and they generally operate on system models and not on the actual system. In fact, the system is represented by a finite model M and the specification is represented by a formula ϕ using an appropriate logic. The verification method consists of computing whether the model M satisfies ϕ (i.e. $M \models \phi$) or not (i.e. $M \not\models \phi$).

Recently, model checking Multi-Agent Systems (MASs) has seen an increasing interest [33, 61, 62, 232, 266, 267, 337, 354, 356, 440]. However, although research in agent communication has received much attention during the past years, only few research works tried to address the verification of agent protocols [4, 24, 163, 195, 244, 430]. Several dialogue game protocols have been proposed for specifying agent communication interactions [37, 304, 307, 381]. These games aim at offering more flexibility by combining different small games to construct complete and more complex protocols. Dialogue games can be thought of as interaction games in which each agent plays a move in turn by performing utterances according to a pre-defined set of rules.

The verification problem of agent communication protocols is fundamental for the MASs community. Endriss et al. [163] have proposed abductive logic-based agents and some means of determining whether or not these agents behave in conformance with agent communication protocols. Baldoni et al. [24] have addressed the problem of verifying that a given protocol implementation using a logical language conforms to its AUML specification. Alberti et al. [4] have considered the problem of verifying on the fly the compliance of the agents' behavior to protocols specified using a logic-based framework. These approaches are different from the technique presented in this chapter in the sense that they are not based on model checking techniques and they do not address the problem of verifying if a protocol satisfies given properties. Giordano et al. [195] have addressed the problem of specifying and verifying agent interaction protocols using a Dynamic Linear Time Temporal Logic (DLTL). The authors have addressed three kinds of verification problems: 1) the compliance of a protocol execution to its specification; 2) the satisfaction of a property in the protocol; 3) the compliance of agents to the protocol. They have shown that these problems can be solved by model checking DLTL. This model checking technique uses a tableau-based algorithm for obtaining a Büchi automaton from a formula in DLTL and the construction of this automaton uses proof rules. However, the protocols are only specified in an abstract way in terms of the effects of communicative actions and some precondition laws.

In this chapter, we describe model checking-based verification of dialogue game protocols for agent communication using an action and temporal logic (ACTL*) based on the Extended Computation Tree Logic CTL*. Using a model checking technique for this verification is motivated by the fact that model-checking is a successful technique for automatically and computationally verifying protocol specifications using a suitable logic. This technique can be used to verify the protocol correctness in the sense that the protocol satisfies the expected properties. It allows us to verify agent communication properties specified using ACTL* logic. Therefore, we can specify the protocol in a logical way and verify its correctness in terms of the satisfaction of the expected properties. The definition of a new logic is motivated by the fact that dialogue game protocols should be specified using not only temporal properties, but also action properties. In addition, in these protocols, actions that agents perform by communicating are expressed in terms of "Social Commitments" (SCs) and arguments. These protocols are specified as transition systems (TSs) using ACTL* logic and Commitment and Argument Network (CAN) [38]. These TSs are labeled with actions that agents perform on SCs and SC contents [115,182,404].

The model checking technique we describe in this chapter is based on the translation of the formula expressing the property to be verified into a variant of alternating tree automata called Alternating Büchi Tableau Automata (ABTA). This technique is an extension of the ABTA-based algorithm for CTL* proposed in [44]. The choice of this technique is motivated by the fact that unlike other model checking techniques, this technique allows us to check temporal and action formulas. In addition, this technique is one of the most efficient techniques proposed in the literature. The translation procedure uses a set of inference rules called tableau rules. Like automata-based model checking of Linear Temporal Logic LTL, our technique is based on the product graph of the model and the automaton representing the formula to be verified (Fig. 3.1). This technique allows us to verify not only that the dialogue game protocol satisfies a given property, but also that this protocol respects the decomposition rules of the communicative acts. Consequently, if agents respect these protocols, then they also respect the decomposition semantics of the communicative acts. Thus, we have only one procedure to verify both:

1. the correctness of the protocols relative to the properties that the protocols should satisfy;
2. the conformance of agents to the decomposition semantics of the communicative acts.

The rest of this chapter is organized as follows. Section 3.2 presents an overview of model checking MASs. Section 3.3 introduces tableau-based algorithms for model checking, which we use in the verification procedure. Section 3.4 presents the ACTL* logic: syntax, semantics and associated tableau rules. In Section 3.5, we use this logic to define the TS that we use to specify dialogue game protocols. The problem of verifying these protocols is addressed in Section 3.6. The ABTA's definition that we use in our verification technique along with some running examples of the model checking steps are presented in this section. Section 3.7 presents two

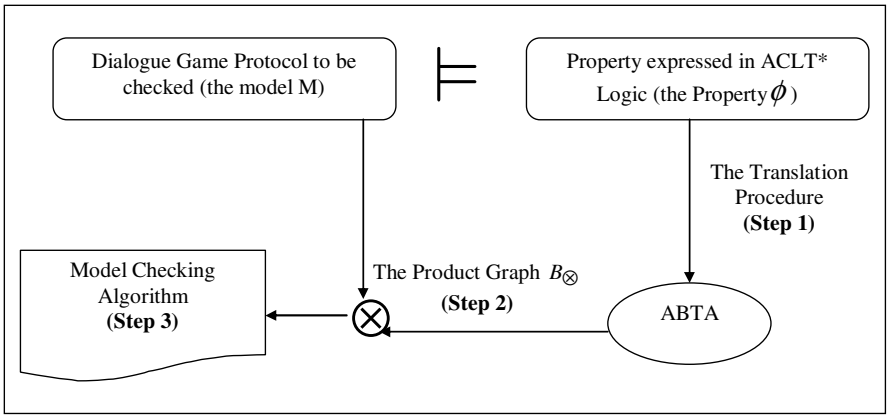

Fig. 3.1 The model checking approach

case studies and Section 3.8 concludes the chapter by discussing open challenges in the area of verifying MASs and identifying some directions for future work.

3.2 Brief Overview of Model Checking Multi-Agent Systems

3.2.1 Extending and Adapting Existing Model Checkers

Bordini and his colleagues [61, 62, 73] have addressed the problem of verifying MASs specified using the AgentSpeak(F) language (a simplified version of AgentSpeak) against BDI specifications. They have shown how programs written in AgentSpeak(F) can be automatically transformed into Promela and into Java and how the BDI specifications are transformed into LTL formulae. The Spin model checker[1] based on Promela [236] and Java PathFinder 2 (JPF2) model checker[2] based on translating Java to Promela [211] are then used to verify the MAS specifications. The idea behind using AgentSpeak(F) instead of the original AgentSpeak is to make the system to be checked finite in terms of state space, which is a fundamental condition of using model checking techniques. To this end, the maximum sizes of types, data structures and communication channels are specified. Examples of these maximum sizes are: M_{Term}: maximum number of terms in a predicate or an action; M_{Conj}: maximum number of conjuncts (literals) in a plan's context;

[1] The Spin model checker can be downloaded from: http://spinroot.com/spin/Man/README.html

[2] The JPF2 model checker is open source and can be downloaded from: http://javapathfinder.sourceforge.net/

M_{Var}: maximum number of different variables in a plan; M_{Bel}: maximum number of beliefs an agent can have at any moment in time in its belief base; and M_{Msg}: maximum number of messages (generated by inter-agent communication) that an agent can handle at a time.

The main constructs in a Promela program are Promela channels and in order to translate AgentSpeak(F) into Promela, the following channels are used to capture the data structures used in an AgentSpeak(F) program: (1) channel b for the agent's belief base with M_{Bel} messages as maximum size and each message has $M_{Terms} + 1$ as maximum size; (2) channel p for the environment's percepts where the maximum size is the same as for channel b; (3) channel m for sending agent communication messages where the bound is M_{Msg} messages; (4) channel e for events, which are related to intentions; (5) channel i for scheduling intentions; and channel a for storing actions. Promela inline procedures are used to code the bodies of agents' plans. The environment is implemented as a Promela process type defined by the user.

Channel m is used to handle messages when the agent interpretation cycle starts, and channels p and b are used by the agent to run its belief revision. Events are handled according to FIFO policy: when new events are generated, they are inserted in the end of channel e, and the first message in that channel is selected as the event to be handled in the current cycle. Translating a formula that appears in a plan body is done as follows: basic actions are appended to channel a; addition and deletion of beliefs is translated as adding or removing messages to/from channel b; and test goals are simply an attempt to match the associated predicate with any message from channel b.

To check BDI properties, BDI modalities are interpreted in terms of Promela data structures associated to an agentSpeak(F) agent. For instance, an AgentSpeak(F) agent believes a formula ϕ iff it is included in the agent's belief base, and this agent intends ϕ iff it has ϕ as an achievement goal that currently appears in its set of intentions, or ϕ is an achievement goal that appears in the (suspended) intentions associated with the set of events.

In the same line of research, Rao and Georgeff [356] have proposed an adaptation of CTL and CTL* model checking to verify BDI (beliefs, desires and intentions) logics. Furthermore, van der Hoek and Wooldridge [232] have reduced the problem of model checking knowledge for multi-agent systems to linear temporal logic model checking using the logic of local propositions [165]. The Spin model checker is then used to check temporal epistemic properties. In [440], Wooldridge et al. have presented the translation of the MABLE language for the specification and verification of MASs into Promela. MABLE is an imperative and agent-oriented programming language where agents have mental states consisting of beliefs, desires and intentions and communicate using *request* and *inform* performatives. The inputs of the MABLE compiler are the MABLE system and associated claims expressed in $\mathcal{MORA}$, a BDI logic. As output, MABLE generates a description of the MABLE system in Promela and a translation of the claims into LTL. In another work, Huget and Wooldridge [244] have used a variation of the MABLE language to define a semantics of agent communication and have shown that the compliance to

this semantics can be reduced to a model checking problem. In [430], Walton has applied model checking techniques in order to verify the correctness of agent protocol communication using the SPIN model checker. Benerecetti and Cimatti [33] have proposed a general approach for model-checking MASs together with modalities for BDI attitudes by extending symbolic model checking and using NuSMV[3] [98], a model checker for computation tree logic CTL. In [355], Lomuscio et al. have introduced a methodology for model checking multi-dimensional temporal-epistemic logic CTLK by extending NuSMV. The methodology is based on reducing the model checking of CTLK to the problem of model checking ARCTL, an extension of CTL with action labels and operators to reason about actions [335].

3.2.2 Developing New Algorithms and Tools

To model MASs, the authors in [354, 355] use the formalism of interpreted systems [205]. This formalism is defined as follows. Assume a set of agents $Ag = \{1, ..., n\}$, where each agent i is characterized by a finite set of local states L_i and possible actions Act_i together with a protocol $P_i : L_i \rightarrow 2^{Act_i}$. The set $S = L_1 \times \cdots \times L_n \times L_E$ represents global states for the system where L_E is the set of local states associated to the environment. Agents' local states evolve in time according to the evolution function $t_i : L_i \times L_E \times Act \rightarrow L_i$, where $Act = Act_1 \times \cdots \times Act_n$. Given a set of initial global states $I \subseteq S$, the set of reachable states $Rs \subseteq S$ is generated by the possible runs of the system using the evolution function and the protocol. An interpretation system is then a tuple: $IS = \langle (L_i, Act_i, P_i, t_i)_{i \in Ag}, I, V \rangle$, where $V : S \rightarrow 2^{AP}$ is the evaluation function over the set of atomic propositions AP. The MAS is analyzed using a logic combining epistemic logic $S5_n$ with CTL logic. The syntax is as follows:
$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E[\varphi U \varphi] \mid K_i\varphi$.
$K_i\varphi$ means i knows φ. The meanings of the other operators are as in CTL, where E is the existential path quantifier, X is the next operator, G is the globally operator and U is the until operator.

To evaluate the formulae, a Kripke model $M_{IS} = (Rs, I, R_t, \sim_1, \ldots, \sim_n, V)$ is associated with a given interpreted system IS. The temporal relation $R_t \subseteq Rs \times Rs$ is obtained using the protocols P_i and the evolutions functions t_i, and the epistemic relations $\sim_1, \ldots, \sim_n$ are defined by checking the equality of the i-th local component of two global states (i.e., $(l_1, \ldots, l_n) \sim_i (l'_1, \ldots, l'_n)$ iff $l_i = l'_i$). The semantics is defined in M_{IS} in the standard way.

To check the desired properties, the authors use symbolic model checking based on ordered binary decision diagrams (OBDDS). The model and formula to be checked are not represented as automata, but symbolically using boolean functions. This makes the technique efficient to deal with large systems. NuSMV [98] is

[3] The NuSMV2 model checker is open source and can be downloaded from: http://nusmv.fbk.eu/NuSMV/download/getting-v2.html

the most popular symbolic model checker based on OBDDS. The MCMAS model checker[4] proposed in [355] is an extension of NuSMV for the epistemic properties. The idea is to represent the elements of the interpreted system M_{IS} by means of boolean formulas and then develop a propositional satisfiability solver (SAT) based on this representation for the verification of the properties associated with the interpreted system.

Agents' local states and actions are encoded as boolean vectors, which are identified by boolean formulae. Protocols and evolutions functions associated with local states and actions are also represented via boolean formulae. The SAT algorithm is an extension of CTL SAT solver for the knowledge operator $K_i\varphi$ whose semantics is defined using the accessibility relation $\sim_i$. Let R_i be the boolean function representing $\sim_i$, the SAT component of this operator is defined as follows:

$SAT_K(\varphi, i)\{$
$X = SAT(\neg\varphi)$
$Y = \{s | R_i(s) \cap X = \emptyset\}$
return $Y \cap Rs$
$\}$

The idea of the algorithm is to compute the set of global states X in which the negation of φ holds. Then, the set Y of states of which the $\sim_i$ accessible states are not in X is computed. This means that these states satisfy the semantics of $K_i\varphi$. Among these states, the algorithm returns those are reachable (i.e. those in Rs).

MCMAS model checker takes as input an interpreted system, which is parsed using Lex and Yacc parser. OBDDs are then built for the input parameters. The formula to be checked is then parsed and the SAT algorithm is executed to compute the set of states in which the formula holds, which is then compared with the set of reachable states. The tool is developed in C++.

In the same research direction, Penczek and Lomuscio [337] have developed a bounded model checking algorithm for branching time logic for knowledge (CTLK). In a similar way, Kacprzak et al. [266] have investigated the problem of verifying epistemic properties using CTLK by means of an unbounded model checking algorithm. Kacprzak and Penczek [267] have addressed the problem of verifying game-like structures by means of unbounded model checking. Recently, Cohen et al. [108] have introduced a new abstraction-based model checking technique for MASs aiming at saving representation space and verification time. The MAS is defined in the interpreted systems framework and the abstraction is performed by simplifying and collapsing the local states, local protocol and local evolution function of each agent in the system. Thus, the set L_i of local states of agent i is partitioned into equivalence classes called abstract local states of agent i. Similarly, the set ACT_i of possible actions of agent i is partitioned into equivalence classes called abstract actions of agent i. Local protocols and local evolution functions are abstracted by uniformly replacing any local state with its equivalence class

[4] The MCMAS model checker can be downloaded from:
http://www-lai.doc.ic.ac.uk/mcmas/download.html

and replacing any action with its equivalence class. The authors have shown that the resulting abstract system simulates the concrete system so that if a temporal-epistemic specification holds on the abstract system, the specification also holds on the concrete one.

3.3 Tableau-based Model Checking Dialogue Games

Unlike traditional proof systems which are bottom-up approaches, tableau-based algorithms used for model checking work in a *top-down* or *goal-oriented* fashion [106]. In the tableau-based approach, *tableau rules* are used in order to prove a certain formula by inferring when a state in a Kripke structure satisfies such a formula. According to this approach, we start from a goal (a formula), and we apply a tableau rule and determine the sub-goals (sub-formulae) to be proven. The tableau rules are designed so that the goal is true if all the sub-goals are true. The advantage of this method is that the state space to be checked is explored in a need-driven fashion [44]. The model checking algorithm searches only the part of the state space that needs to be explored to prove or disprove a certain formula. The state space is constructed while the algorithm runs. This kind of model checking algorithms is referred to as *on-the-fly* or *local* algorithms [44, 45, 106, 408].

The tableau decision algorithm that we use in our verification technique provides a systematic search for a model which satisfies a particular formula expressed using ACTL* logic. It is a graph construction algorithm. Nodes of the graph are sets of ACTL* formulae and tableau rule names. The interpretation of vertex labeling is that for the vertex to be satisfied, it must be possible to satisfy all the formulae in the set together. Each edge in the graph represents a satisfaction step of the formula contained in the starting vertex. These steps correspond to the application of a set of tableau rules. These rules express how the satisfaction of a particular formula (the goal) can be obtained by the satisfaction of its constituent formulae (sub-goals).

3.4 ACTL* Logic

3.4.1 Syntax

In this section, we present ACTL* logic that we use to specify dialogue game protocols and express the properties to be verified (See Fig. 3.1). This specification will be addressed in Section 3.5. ACTL* is a simplification of our logic for agent communication [38]. ACTL* extends CTL* by allowing formulae to constrain actions as well as propositions. The difference between ACTL* and CTL* is that the former contains action formulae and two new operators: SC for social commitments and $\therefore$

for arguments. The set of atomic propositions is denoted Γp. The set of action labels is denoted Γa. In what follows we use $p, p_1, p_2, \ldots$ to range over the set of atomic propositions and $\theta, \theta_1, \theta_2, \ldots$ to range over action labels. The syntax of this logic is as follows:

$$\mathcal{S} ::= p \mid \neg\mathcal{S} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid A\mathcal{P} \mid E\mathcal{P} \mid SC(Ag_1, Ag_2, \mathcal{P})$$

$$\begin{aligned}\mathcal{P} ::= {} & \theta \mid \neg\mathcal{P} \mid \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid X\mathcal{P} \mid \mathcal{P}U\mathcal{P} \mid \mathcal{P} \therefore \mathcal{P} \\ & \mid ACT_1(Ag_1, SC(Ag_1, Ag_2, \mathcal{P})) \mid ACT_2(Ag_2, SC(Ag_1, Ag_2, \mathcal{P})) \\ & \mid ACT_1^+(Ag_1, SC(Ag_1, Ag_2, \mathcal{P}), \mathcal{P}) \mid ACT_2^+(Ag_2, SC(Ag_1, Ag_2, \mathcal{P}), \mathcal{P})\end{aligned}$$

$$ACT_1 ::= Cr \mid Wit \mid Sat \mid Vio$$

$$ACT_2 ::= Ac \mid Ref \mid Ch$$

$$ACT_1^+ ::= Def \mid Jus$$

$$ACT_2^+ ::= At$$

The formulae generated by $\mathcal{S}$ are called state formulae, while those generated by $\mathcal{P}$ are called path formulae. We use $\psi, \psi_1, \psi_2, \ldots$ to range over state formulae and $\phi, \phi_1, \phi_2, \ldots$ to range over path formulae. The formula $A\phi$ (respectively $E\phi$) means in all paths (resp. some paths) starting from the current state ϕ is satisfied. The formula $SC(Ag_1, Ag_2, \phi)$ means that agent Ag_1 commits towards agent Ag_2 that the path formula ϕ is true. Committing to path formulae is more expressive than committing to state formulae since state formulae are path formulae. In fact, by committing to path formulae, agents can commit to state formulae and express commitments toward the future, for example committing that $X\phi$ (ϕ holds from the next state), $\phi_1 U \phi_2$ (ϕ_1 holds until ϕ_2 becomes true) and $EF\phi$ (there is a path such that in its future ϕ holds)[5]. Ag_1 and Ag_2 are respectively called the *debtor* and *creditor* of the commitment. The formula $\phi_1 \therefore \phi_2$ means that ϕ_1 is an argument for ϕ_2. We can read this formula: ϕ_1, so ϕ_2. This operator introduces argumentation as a logical relation between path formulae. $Action(Ag, SC(Ag_1, Ag_2, \phi))$ and $Action^+(Ag, SC(Ag_1, Ag_2, \phi), \phi_1)$, where $Action$ corresponds to ACT_1 and ACT_2 and $Action^+$ corresponds to ACT_1^+ and ACT_2^+, indicate the action an agent Ag ($Ag \in \{Ag_1, Ag_2\}$) performs on $SC(Ag_1, Ag_2, \phi)$. The actions we consider are: Cr (create), Wit (withdraw), Sat (satisfy), Vio (violate), Ac (accept), Ref (refuse), Ch (challenge), At (attack), Def (defend) and Jus (justify).

[5] Operator F (in the future) is an abbreviation defined from operator U: $F\phi \equiv trueU\phi$

3.4.2 Semantics

Semantically, this logic is interpreted with respect to the model M defined as follows: $M = \langle S_m, Lab, Act_m, \xrightarrow{Act_m}, Agt, R_{sc}, s_{m_0} \rangle$ where: S_m is a set of states; $Lab : S_m \rightarrow 2^{\Gamma_p}$ is the labeling state function; Act_m is a set of actions; $\xrightarrow{Act_m} \subseteq S_m \times Act_m \times S_m$ is the transition relation; Agt is a set of communicating agents; $R_{sc} : S_m \times Agt \times Agt \rightarrow 2^{\sigma}$ with σ is the set of all paths in M is an accessibility modal relation that associates to a state s_m the set of paths along which an agent can commit towards another agent; s_{m_0} is the start state. The paths that path formulae are interpreted over have the form $x = s_{m_0} \xrightarrow{\alpha_1} s_{m_1} \xrightarrow{\alpha_2} s_{m_2} \ldots$ where $x \in \sigma$, $s_{m_0}, s_{m_1}, \ldots$ are states and $\alpha_1, \alpha_2, \ldots$ are actions. $x^i = s_{m_i} \xrightarrow{\alpha_{i+1}} s_{m_{i+1}} \ldots$ is the suffix of the path x starting from the *ith* state. The set of paths starting from a state s_m is denoted σ_m. $x[i]$ is the *ith* state in the path x. In the rest, $\Rightarrow$ stands for implies.
$s_m \models_M p$ iff $p \in Lab(s_m)$
$s_m \models_M \neg\psi$ iff not$(s_m \models_M \psi)$
$s_m \models_M \psi_1 \wedge \psi_2$ iff $s_m \models_M \psi_1$ and $s_m \models_M \psi_2$
$s_m \models_M \psi_1 \vee \psi_2$ iff $s_m \models_M \psi_1$ or $s_m \models_M \psi_2$

A state s_m satisfies $A\phi$ ($E\phi$) if every path (some path) emanating from this state satisfies ϕ. Formally:
$s_m \models_M A\phi$ iff $\forall x \in \sigma_m \; x \models_M \phi$
$s_m \models_M E\phi$ iff $\exists x \in \sigma_m \; x \models_M \phi$

A state s_m satisfies $SC(Ag_1, Ag_2, \phi)$ if every accessible path to Ag_1 towards Ag_2 from this state using R_{sc} satisfies ϕ. Formally:
$s_m \models_M SC(Ag_1, Ag_2, \phi)$ iff $\forall x \in R_{sc}(s_m, Ag_1, Ag_2) \; x \models_M \phi$.

A path satisfies a state formula if the initial state in the path does. Formally:
$x \models_M \psi$ iff $s_{m_0} \models_M \psi$

To define the satisfiability of action labels over paths, we introduce the notation $\theta \trianglerighteq \alpha_i$ where $i \geq 1$ to indicate that the action label θ becomes true when performing the action α_i, that is α_i brings about θ (for example, by performing the action of opening the door the action label "door is open" becomes true. If not, we write $\theta \not\trianglerighteq \alpha_i$. A path x satisfies an action label θ if θ is in the label of the first transition on this path and this path is not a *deadlocked* path. A path is deadlocked if it has no transitions. A path satisfies $\neg\theta$ if either θ is not in the label of the first transition on this path or this path is a deadlocked path. Formally:
$x \models_M \theta$ iff $\theta \trianglerighteq \alpha_1$ and *x is not a deadlocked path*
$x \models_M \neg\theta$ iff $\theta \not\trianglerighteq \alpha_1$ or *x is a deadlocked path*
where the action α_1 is the label of the first transition on the path x.
$x \models_M \neg\phi$ iff not$(x \models_M \phi)$

$x \models_M \phi_1 \wedge \phi_2$ iff $x \models_M \phi_1$ and $x \models_M \phi_2$
$x \models_M \phi_1 \vee \phi_2$ iff $x \models_M \phi_1$ or $x \models_M \phi_2$

X represents the next time operator and has the usual semantics when the path is not deadlocked. On a deadlocked path, $X\phi$ holds if the current state satisfies ϕ. Formally:
$x \models_M X\phi$ iff (x *is not a deadlocked path* $\Rightarrow x^1 \models_M \phi$) and
(x *is a deadlocked path* $\Rightarrow x[0] \models_M \phi$)

In the rest, the path x is supposed non-deadlocked. Along this path, $\phi_1 U \phi_2$ holds if ϕ_1 remains true along this path until ϕ_2 becomes true (strong until). Formally:
$x \models_M \phi_1 U \phi_2$ iff $\exists i \geq 0 : x^i \models_M \phi_2$ and $\forall j < i, x^j \models_M \phi_1$

Along a path x, $\phi_1 \therefore \phi_2$ holds if ϕ_1 is true and at next time if ϕ_1 is true then ϕ_2 is true. Formally:
$x \models_M \phi_1 \therefore \phi_2$ iff $x \models_M \phi_1$ and $x^1 \models_M \phi_1 \Rightarrow \phi_2$
Because the semantics of $\therefore$ operator is defined using existing operators, it is introduced here as a useful abbreviation, which will be used to define the semantics of some actions performed on SCs.

To specify dialogue game protocols in this logic according to the CAN framework, we use a set of actions performed by the communicating agents on SCs and SC contents. The idea behind the CAN framework is that agents communicate by performing actions on SCs (for example creating, accepting and challenging SCs) and by supporting these actions by argumentation relations (attack, defense, and justification). Such an approach, called the social approach [318] is considered as an alternative to the private approach based on the agents' mental states like beliefs, desires, and intentions [109]. The semantics of the action formulae is defined as follows:
$x \models_M Cr(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Cr$ and $s_{m_1} \models_M SC(Ag_1, Ag_2, \phi)$
$x \models_M Wit(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Wit$ and $s_{m_1} \models_M \neg SC(Ag_1, Ag_2, \phi)$
$x \models_M Sat(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Sat$ and $s_{m_1} \models_M \phi$
$x \models_M Vio(Ag_1, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Vio$ and $s_{m_1} \models_M \neg\phi$
$x \models_M Ac(Ag_2, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Ac$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, \phi)$
$x \models_M Ref(Ag_2, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Ref$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, \neg\phi)$
$x \models_M Ch(Ag_2, SC(Ag_1, Ag_2, \phi))$ iff $\alpha_1 = Ch$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, ?\phi)$
$x \models_M At(Ag_2, SC(Ag_1, Ag_2, \phi_1), \phi_2)$ iff $\alpha_1 = At$ and $s_{m_1} \models_M SC(Ag_2, Ag_1, \phi_2 \therefore \neg\phi_1)$
$x \models_M Def(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)$ iff $\alpha_1 = Def$ and $s_{m_1} \models_M SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1)$
$x \models_M Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)$ iff $\alpha_1 = Jus$ and $s_{m_1} \models_M SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1)$

$Cr(Ag_1, SC(Ag_1, Ag_2, \phi))$ is satisfied along the path x iff the first transition is labeled by Cr and the underlying commitment holds in the next state on that path. The semantics of the other formulae is defined in the same way. The commitment is withdrawn iff after performing the action, the commitment does not hold in the next state. It is satisfied (resp. violated) iff after the action, the content becomes true

(resp. false) in the next state. When Ag_2 accepts (resp. refuses) the commitment content, it becomes committed to the same content (resp. the negation of the same content) in the next state. For simplification reasons, the semantics of challenge is defined by introducing a syntactical construct "?" to indicate that the debtor Ag_2 of the resulting commitment $SC(Ag_2, Ag_1, ?\phi)$ does not have an argument supporting ϕ or $\neg\phi$. For the purpose of model checking dialogue games, this syntactical construct is useful for the tableau-based verification technique we will present in Section 3.6. The content ϕ_1 of Ag_1's commitment is attacked by Ag_2 using ϕ_2 iff after performing the attack action, Ag_2's commitment about $\phi_2 \therefore \neg\phi_1$ holds in the next state. Ag_1 defends its commitment (against an attack) and justifies it (against a challenge) iff after performing the action, the Ag_1's commitment about $\phi_2 \therefore \phi_1$ holds in the next state.

ACTL* logic is the fusion of CTL* logic and a logic for commitments. The logic for commitments has the following properties, where $\rightarrow$ is the classical implication:

1. R_{sc} is serial (axiom D);
2. R_{sc} is reflexive (axiom M) because accessible paths start from the current state where the commitment has been made and a formula is satisfied along a path if it is satisfied in the initial state of this path, which means on an accessible path we have $SC(Ag_1, Ag_2, \phi) \rightarrow \phi$
3. R_{sc} is transitive (axiom 4): $SC(Ag_1, Ag_2, \phi) \rightarrow SC(Ag_1, Ag_2, SC(Ag_1, Ag_2, \phi))$.

This makes the logic an $S4$ system.

3.4.3 Tableau Rules

In this section, we present the tableau rules that we use to translate the ACTL* property to be verified to an ABTA (see Fig. 3.1). The definition of ABTA and the translation procedure will be presented in Sections 3.6.1 and 3.6.2. The tableau rules allow us to build the ABTA representing the formula to be verified. These rules [106] are specified in terms of the decomposition of formulae to sub-formulae. They enable us to define top-down proof systems. The idea is: given a formula (the top part of the rule), we apply a tableau rule and determine the sub-formulae (the down part of the rule) to be proven (see Section 3.3). Tableau rules are inference rules used in order to prove a formula by proving all the sub-formulae. The labels of these rules are the labels of states in the ABTA constructed from the given formula (Section 3.6.1). These rules are presented in Table 3.1. In these rules, Φ is any set of path formulae. The symbol "," indicates a conjunction. For example, $E(\Phi, \psi)$ means that, there is a path along which the set of path formulae Φ and the state formula ψ are true. Adding the set Φ to these rules allows us to deal with any form of formulae written under the form of any set of path formulae and a formula of our logic. We

Table 3.1 Tableau rules

$$R1\ \wedge : \frac{\psi_1 \wedge \psi_2}{\psi_1 \psi_2} \quad R2\ \vee : \frac{\psi_1 \vee \psi_2}{\psi_1}, \frac{\psi_1 \vee \psi_2}{\psi_2} \quad R3\ \vee : \frac{E(\psi)}{\psi} \quad R4\ \neg : \frac{\neg\psi}{\psi} \quad R5\ \neg : \frac{A(\Phi)}{E(\neg\Phi)}$$

$$R6 < Cr > : \frac{E(\Phi, Cr(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_1, Ag_2, \phi))} \quad R11 < Ac > : \frac{E(\Phi, Ac(Ag_2, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_2, Ag_1, \phi))}$$

$$R7 < Wit > : \frac{E(\Phi, Wit(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, \neg SC(Ag_1, Ag_2, \phi))} \quad R12 < Ref > : \frac{E(\Phi, Ref(Ag_2, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_2, Ag_1, \neg\phi))}$$

$$R8 < Sat > : \frac{E(\Phi, Sat(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, \phi)} \quad R13 < Jus > : \frac{E(\Phi, Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))}{E(\Phi, SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1))}$$

$$R9 < Vio > : \frac{E(\Phi, Vio(Ag_1, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, \neg\phi)} \quad R14 < At > : \frac{E(\Phi, At(Ag_2, SC(Ag_1, Ag_2, \phi_1), \phi_2))}{E(\Phi, SC(Ag_2, Ag_1, \phi_2 \therefore \neg\phi_1))}$$

$$R10 < Ch > : \frac{E(\Phi, Ch(Ag_2, SC(Ag_1, Ag_2, \phi)))}{E(\Phi, SC(Ag_2, Ag_1, ?\phi))} \quad R15 < Def > : \frac{E(\Phi, Def(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))}{E(\Phi, SC(Ag_1, Ag_2, \phi_2 \therefore \phi_1))}$$

$$R16\ [SC_{Ag_1}] : \frac{E(\Phi, SC(Ag_1, Ag_2, \phi))}{E(\Phi, \phi)} \quad R17 <\equiv> : \frac{E(\Phi, \Psi)}{E(\Phi) E(\Psi)} \quad R18\ \wedge : \frac{E(\Phi, \phi_1 \wedge \phi_2)}{E(\Phi, \phi_1, \phi_2)}$$

$$R19\ \vee : \frac{E(\Phi, \phi_1 \vee \phi_2)}{E(\Phi, \phi_1) E(\Phi, \phi_2)} \quad R20\ X : \frac{E(\Phi, X\phi_1, ..., X\phi_n)}{E(\Phi, \phi_1, ..., \phi_n)} \quad R21\ \wedge : \frac{E(\Phi, \phi_1 \therefore \phi_2)}{E(\Phi, \phi_1, X(\neg\phi_1 \vee \phi_2))}$$

$$R22\ \vee : \frac{E(\Phi, \phi_1 U \phi_2)}{E(\Phi, \phi_2) E(\Phi, \phi_1, X(\phi_1 U \phi_2))}$$

also recall that we use $\psi, \psi_1, \psi_2, \ldots$ to range over state formulae and $\phi, \phi_1, \phi_2, \ldots$ to range over path formulae.

Rule $R1$ labeled by "$\wedge$" indicates that ψ_1 and ψ_2 are the two sub-formulae of $\psi_1 \wedge \psi_2$. This means that, in order to prove that a state labeled by "$\wedge$" satisfies the formula $\psi_1 \wedge \psi_2$, we have to prove that the two children of this state satisfy ψ_1 and ψ_2 respectively. According to rule $R2$, in order to prove that a state labeled by "$\vee$" satisfies the formula $\psi_1 \vee \psi_2$, we have to prove that one of the two children of this state satisfies ψ_1 or ψ_2. $R3$ labeled by "$\vee$" indicates that ψ is the sub-formula to be proved in order to prove that a state satisfies $E(\psi)$. E is the existential path-quantifier. According to $R4$, the formula $\neg\psi$ is satisfied in a state labeled by "$\neg$" if this state has a successor representing the sub-formula ψ, which is not satisfied. $R5$ is defined in the usual way.

The label "$<Cr>$" ($R6$) is the label associated with the creation action of a social commitment. According to this rule, in order to prove that a state labeled by "$<Cr>$" satisfies $Cr(Ag_1, SC(Ag_1, Ag_2, \phi))$, we have to prove that the child state satisfies the sub-formula $SC(Ag_1, Ag_2, \phi)$. The idea is that by creating a social commitment, this commitment becomes true in the child state. In the model representing the dialogue game protocol, the idea behind the creation action is that by creating a social commitment, this commitment becomes true in the accessible state via the transition labeled by the creation action. The label "$<Wit>$" ($R7$) is the label associated with the withdrawal action of a social commitment. According to this rule, in order to prove that a state labeled by "$<Wit>$" satisfies $Wit(Ag_1, SC(Ag_1, Ag_2, \phi))$, we have to prove that the child state satisfies the sub-formula $\neg SC(Ag_1, Ag_2, \phi)$. Rules $R8$ to $R15$ are defined in the same way. For example, the idea of rule $R11$ is that by accept-

ing a social commitment whose content is ϕ by an agent Ag_2, this agent commits about this content in the child state. In this state, the commitment of Ag_2 becomes true. In rule $R10$, we use the syntactical construct "?" introduced in Section 3.4.2 meaning that the debtor Ag_2 does not have an argument supporting ϕ or $\neg\phi$. The idea of this rule is that by challenging a social commitment, Ag_2 commits in the child state that it does not have an argument for or against the content ϕ. Because "?" is only a syntactical construct, $?\phi$ does not have a sub-formula, so there is no rule for "?".

Rule $R16$ indicates that $E(\phi)$ is the sub-formula of $E(SC(Ag_1, Ag_2, \phi))$. Thus, in order to prove that a state labeled by "$[SC_{Ag_1}]$" satisfies formula $E(SC(Ag_1, Ag_2, \phi))$, we have to prove that the child state satisfies the sub-formula $E(\phi)$. According to the semantics of social commitments (Section 3.4), the idea of this rule is that if an agent commits about a content along a path, this content is true along this path (we recall that the commitment content is a path formula).

Rules $R17$, $R18$, and $R19$ are straightforward. According to rule $R20$ and in accordance with the semantics of "X", in order to prove that a state labeled with "X" satisfies $E(X\phi)$, we have to prove that the child state satisfies the sub-formula $E(\phi)$. According to $R21$ and in accordance with the semantics of "$\therefore$" (Section 3.4), in order to prove that a state labeled with "$\wedge$" satisfies $E(\phi_1 \therefore \phi_2)$, we have to prove that the child state satisfies the sub-formula $E(\phi_1 \wedge X(\neg\phi_1 \vee \phi_2))$. This mean that the support is true and next if the support is true then the conclusion is true. Finally, rule $R22$ is defined in accordance with the usual semantics of *until* operator "U".

3.5 Dialogue Game Protocols as Transition Systems

In Section 3.4, we presented ACTL* logic and CAN-based actions. In this section, we specify the dialogue game protocols to be checked as models for this logic (see Fig. 3.1). This specification uses CAN-based actions and the labels of the tableau rules that we will present in Section 3.4.3. Dialogue game protocols are specified as a set of rules describing the entry condition, the dynamics and the exit condition [37]. These rules can be specified as CAN-based actions.

Dialogue game protocols are defined as TSs. The purpose of these TSs is to describe not only the sequence of the allowed actions (classical TSs), but also the tableau rules-based decomposition of these actions (Section 3.4.3). The states of these systems are sub-TSs (that we call decomposition TSs) describing the tableau rules-based decomposition of the actions labeling the entry transitions. Defining TSs in such a way allows us to verify: (1) The correctness of the protocol (if the model of the protocol satisfies the properties that the protocol should specify); (2) The compliance to the decomposition semantics of the communicative actions (if the specification of the protocol respects the decomposition semantics). In Section 3.6, we present a model checking procedure in order to verify both (1) and (2) at

the same time. The definition of the TSs of dialogue game protocols is given by the following definitions:

Definition 3.1 (Decomposition TSs). A decomposition transition system (DT) describing the tableau-rules-based decomposition semantics of a CAN based-action formula is a 7-tuple $\langle S', Lab', F, L', R, \xrightarrow{R}, s'_0 \rangle$ where: S' is a set of states; $Lab' : S' \rightarrow 2^{\Gamma_p}$ is the labeling state function; F is a set of ACTL* formulae; $L' : S' \rightarrow 2^F$ is a function associating a set of formulae to a state; $R \in \{\wedge, \vee, \neg, <\equiv>, X, SC_{Ag}\}$ is a tableau rule label (without the rules for CAN-based action formulae) (see Section 3.4.3); $\xrightarrow{R} \subseteq S' \times R \times S'$ is the transition relation; s'_0 is the start state.

Intuitively, states S' contain the sub-formulae of the CAN-based action formulae, and the transitions are labeled by operators associated with the formula of the starting state. Decomposition TSs enable us to describe the decomposition semantics of formulae by sub-formulae connected by logical operators. Thus, there is a transition between states S'_i and S'_j iff $L'(S'_j)$ is a sub-formula of $L'(S'_i)$.

Definition 3.2 (TSs for Dialogue Game Protocols). A transition system T for a dialogue game protocol is a 7-tuple $\langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0 \rangle$ where: S is a set of states; $Lab : S \rightarrow 2^{\Gamma_p}$ is the labeling state function; $\wp$ is a set of decomposition TSs with $\varepsilon \in \wp$ is the empty decomposition TS; $L : S \rightarrow \wp$ is the function associating to a state $s \in S$ a decomposition transition system $DT \in \wp$ describing the tableau-based decomposition of the CAN-based action labeling the entry transition; Act is the set of CAN-based actions; $\xrightarrow{Act} \subseteq S \times Act \times S$ is the transition relation; s_0 is the start state with $L(s_0) = \varepsilon$.

We write $s \xrightarrow{\bullet} s'$ instead of $< s, \bullet, s' > \in \xrightarrow{Act}$ where $\bullet \in Act$. Fig. 3.2 illustrates a part of a TS for a dialogue game protocol. According to this protocol, if Ag_1 plays a creation game ($a1$), Ag_2 can, for instance, play a challenge game ($a2$). Thereafter, Ag_1 must plays a justification game ($a3$), etc.

States $S1$, $S2$, and $S3$ are decomposition TS associated respectively with creation, challenge, and justification actions. For example, for the creation action ($S1$), the first state ($s1.0$) is associated with the SC formula according to the rule $R6$ (Table 3.1, Section 3.4.3). The next state is associated with the SC content according to the rule $R16$ (Table 3.1). The transition is labeled with the label of this rule. An example of the properties to be verified in this protocol is:

$$AG(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \Rightarrow F(Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))) \qquad (3.1)$$

This property says that in all paths (A) globally (G)[6], if an agent Ag_2 challenges (Ch) the content of a SC made by an agent Ag_1, then in the future (F), Ag_1 justifies (Jus) the content of its SC. In the rest of this chapter, we refer to this formula as **Formula 1**.

[6] Operator G (globally in the future) is an abbreviation defined from operator F: $G\phi \equiv \neg F \neg \phi$

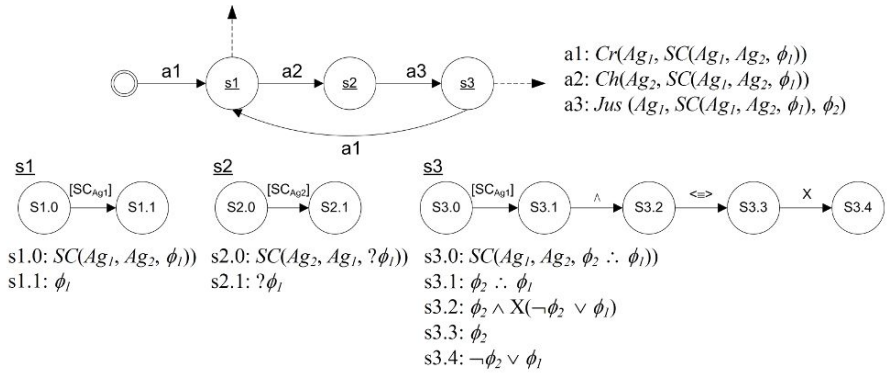

Fig. 3.2 A part of a transition system for a dialogue game protocol

3.6 Verification of Dialogue Game Protocols

In previous sections, we presented the elements needed for the verification of dialogue game protocols: the logic along with the associated tableau rules and the specification of dialogue game protocols. In this section, we present the verification technique, which is based upon (1) the ABTA for ACTL* logic (Section 3.6.1); and (2) the translation of the property to be verified to an ABTA (Section 3.6.2) (see Fig. 3.1). This translation is the step 1 of Fig. 3.1. The step 2, which is the construct of the product graph of the model and the ABTA is addressed in Section 3.6.3. Finally, the model checking algorithm acting on the product graph (step 3) is presented in Section 3.6.4. Examples illustrating each step are also presented.

*3.6.1 Alternating Büchi Tableau Automata (ABTA) for ACTL**

As a kind of Büchi automata, ABTAs [44] are used in order to prove properties of *infinite* behavior. These automata can be used as an intermediate representation for system properties. Let Γp be the set of atomic propositions and let $\mathfrak{R}$ be a set of tableau rule labels defined as follows: [7]
$\mathfrak{R} = \{\wedge, \vee, \neg\} \cup \mathfrak{R}_{Act} \cup \mathfrak{R}_{\neg Act} \cup \mathfrak{R}_{SC} \cup \mathfrak{R}_{Set}$ where: $\mathfrak{R}_{Act} = \{<Cr>, <Wit>, <Sat>, <Vio>, <Ch>, <Ac>, <Ref>, <Jus>, <At>, <Def>\}$, $\mathfrak{R}_{SC} = \{[SC_{Ag}]\}$, and $\mathfrak{R}_{Set} = \{<\equiv>, X\}$.
We define ABTAs for ACTL* logic as follows:

[7] The partition of the set of tableau rule labels is only used for readability and organizational reasons.

Definition 3.3 (ABTA). An ABTA for ACTL* is a 5-tuple $\langle Q, l, \rightarrow, q_0, F \rangle$, where: Q is a finite set of states; $l : Q \rightarrow \Gamma p \cup \Re$ is the state labeling; $\rightarrow \subseteq Q \times Q$ is the transition relation; q_0 is the start state; $F \subseteq 2^Q$ is the acceptance condition[8].

ABTAs allow us to encode "*top-down proofs*" for temporal formulae. Indeed, an ABTA encodes a proof schema in order to prove, in a goal-directed manner, that a TS satisfies a temporal formula. Let us consider the following example. We would like to prove that a state s in a TS satisfies a temporal formula of the form $F_1 \wedge F_2$, where F_1 and F_2 are two formulae. Regardless of the structure of the system, there would be two sub-goals. The first would be to prove that s satisfies F_1, and the second would be to prove that s satisfies F_2. Intuitively, an ABTA for $F_1 \wedge F_2$ would encode this "proof structure" using states for the formulae $F_1 \wedge F_2$, F_1, and F_2. A transition from $F_1 \wedge F_2$ to each of F_1 and F_2 should be added to the ABTA and the labeling of the state for $F_1 \wedge F_2$ being "$\wedge$" which is the label of a certain rule. Indeed, in an ABTA, we can consider that: 1) states correspond to "formulae", 2) the labeling of a state is the "logical operator" used to construct the formula, and 3) the transition relation represents a "sub-goal" relationship.

3.6.2 Translating ACTL into ABTA (Step 1)*

The procedure for translating an ACTL* formula $p = E(\phi)$ to an ABTA B uses goal-directed rules in order to build a tableau from this formula. Indeed, these proof rules are conducted in a top-down fashion in order to determine if states satisfy properties. The tableau is constructed by exhaustively applying the tableau rules presented in Table 3.1 to p. Then, B can be extracted from this tableau as follows. First, we generate the states and the transitions. Intuitively, states will correspond to state formulae, with the start state being p. To generate new states from an existing state for a formula p', we determine which rule is applicable to p', starting with $R1$, by comparing the form of p' to the formula appearing in the "*goal position*" of each rule. Let $rule(q)$ denote the rule applied at node q. The labeling function l of states is defined as follows. If q does not have any successor, then $l(q) \in \Gamma p$. Otherwise, the successors of q are given by $rule(q)$. The label of the rule becomes the label of the state q, and the sub-goals of the rule are then added as states related to q by transitions.

A tableau for a ACTL* formula p is a maximal proof tree having p as its root and constructed using our tableau rules (see Section 3.4.3). If p' results from the application of a rule to p, then we say that p' is a child of p in the tableau. The height of a tableau is defined as the length of the longest sequence $< p_0, p_1, \ldots >$, where p_{i+1} is the child of p_i [106].

[8] The notion of acceptance condition is related to the notion of accepting run that we define in Section 3.6.3.

Example 3.1. In order to illustrate the translation procedure and the construction of an ABTA from an ACTL* formula, let us consider our formula Formula 1 given in Section 3.5. Table 3.2 is the tableau to build for translating Formula 1 into an ABTA. The form of Formula 1 is: $AG(p \Rightarrow q)(\equiv AG(\neg p \vee q))$ (the root of Table 3.2). The first rule we can apply is $R5$ labeled by $\neg$ in order to transform *all paths* to *exists a path*. We also use the equivalence $(F(p) \equiv \neg G(\neg p))$. We then obtain the child number (2). The next rule we can apply is $R22$ labeled by $\vee$ because F is an abbreviation of U $(F(p) \equiv True\ U\ p)$. Consequently, we obtain two children (3) and (4). From the child (3) we obtain the child (5) by applying the rule $R10$, and from the child (4) we obtain the child (2) by applying the rule $R20$ etc. The ABTA obtained from this tableau is illustrated by Fig. 3.3. States are labeled by the child's number in the tableau and the label of the applied rule according to Table 3.2.

Table 3.2 The tableau of Formula 1

<table>
<tr><td colspan="4">$\neg : AG(\neg Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \vee F(Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (1)</td></tr>
<tr><td colspan="4">$\vee : EF(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (2)</td></tr>
<tr><td colspan="2">$<Ch>: E(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (3)</td><td colspan="2">$<X>: EX(F(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))))$ (4)</td></tr>
<tr><td colspan="2">$[SC_{Ag_2}] : E(SC(Ag_2, Ag_1, ?\phi_1) \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (5)</td><td colspan="2">$EF(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (2)</td></tr>
<tr><td colspan="2">$<\equiv>: E(?\phi_1 \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (6)</td><td colspan="2"></td></tr>
<tr><td>$?\phi_1$ (7)</td><td colspan="3">$\vee : E(G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (8)</td></tr>
<tr><td></td><td colspan="3">$<\neg Jus>: E(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2), XG(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (9)</td></tr>
<tr><td colspan="4">$[SC_{Ag_1}] : E(SC(Ag_1, Ag_2, \phi_1 \therefore \phi_2), XG(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (10)</td></tr>
<tr><td colspan="4">$\wedge : E(\phi_2 \therefore \phi_1, XG(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (11)</td></tr>
<tr><td colspan="4">$<\equiv>: E(\phi_2, X(\neg\phi_2 \vee \phi_1), XG(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (12)</td></tr>
<tr><td>ϕ_2 (13)</td><td colspan="3">$X : E(X(\neg\phi_2 \vee \phi_1), XG(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (14)</td></tr>
<tr><td></td><td colspan="3">$<\equiv>: E((\neg\phi_2 \vee \phi_1), XG(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (15)</td></tr>
<tr><td colspan="2">$\neg\phi_2 \vee \phi_1$ (16)</td><td colspan="2">$X : E(XG(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (17)</td></tr>
<tr><td colspan="2"></td><td colspan="2">$\vee : E(G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2)))$ (8)</td></tr>
</table>

The termination proof of the translation procedure is based on the finiteness of the tableau. This proof is based on the length of formulae and an ordering relation between these formulae. The proof is detailed in [35].

3.6.3 Run of an ABTA on a Transition System (Step 2)

Like the automata-based model checking of LTL, in order to decide about the satisfaction of formulae, we use the notion of the *accepting runs*. In our technique, we need to define accepting runs of an ABTA on a TS. Firstly, we have to define the

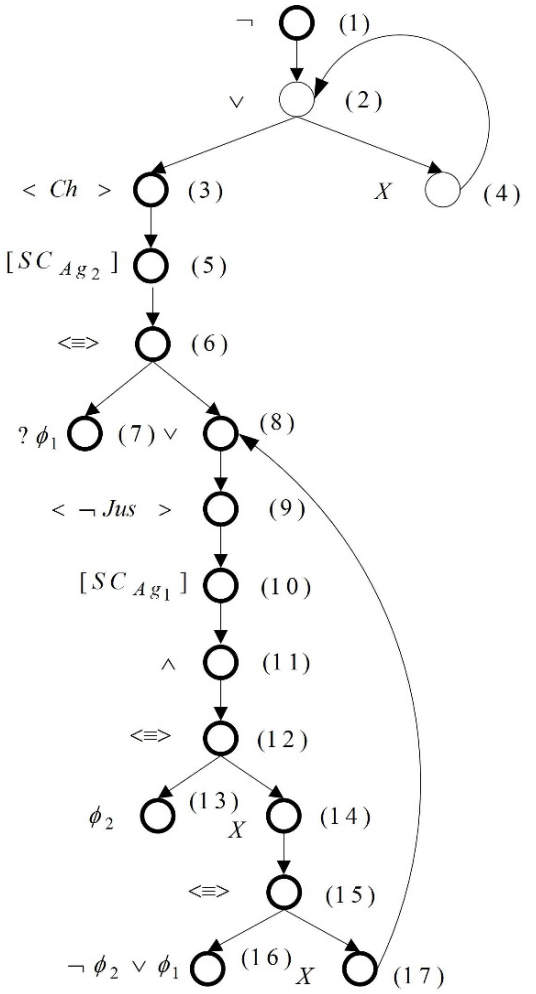

Fig. 3.3 The ABTA of Formula 1

notion of *ABTA's run*. For this reason, we need to introduce two types of nodes: *positive* and *negative*. Intuitively, nodes classified positive are nodes that correspond to a formula without negation, and negative nodes are nodes that correspond to a formula with negation. Definition 3.4 gives the definition of this notion of run. In this definition, elements of the set S of states are denoted s_i or t_i.

Definition 3.4 (Run of an ABTA). A run of an ABTA $B = \langle Q, l, \rightarrow, q_0, F \rangle$ on a transition system $T = \langle S, Lab, \wp, L, Act, \stackrel{Act}{\longrightarrow}, s_0 \rangle$ is a graph in which the nodes are classified as positive or negative and are labeled by elements of $Q \times S$ as follows:

1. The root of the graph is a positive node and is labeled by $< q_0, s_0 >$.

2. For a positive node φ with label $<q, s_i>$:

   a. If $l(q) = \neg$ and $q \rightarrow q'$, then φ has one negative successor labeled $<q', s_i>$ and vice versa.

   b. If $l(q) \in \Gamma p$, then φ is a leaf.

   c. If $l(q) \in \{\wedge, <\equiv>\}$ and $\{q' | q \rightarrow q'\} = \{q_1, \ldots, q_m\}$, then φ has positive successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q_j, s_i>$ $(1 \leq j \leq m)$.

   d. If $l(q) = \vee$, then φ has one positive successor φ' labeled by $<q', s_i>$ for some $q' \in \{q' | q \rightarrow q'\}$.

   e. If $l(q) = X$ and $q \rightarrow q'$ and $\{s' | s_i \xrightarrow{\bullet} s'\} = \{t_1, \ldots, t_m\}$ where $\bullet \in Act$, then φ has positive successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q', t_j>$ $(1 \leq j \leq m)$.

   f. If $l(q) = <\bullet>$ where $\bullet \in Act$ and $q \rightarrow q'$, and $s_i \xrightarrow{\bullet} s_{i+1}$, then φ has one positive successor φ' labeled by $<q', s_{i+1,0}>$ where $s_{i+1,0}$ is the initial state of the decomposition TS of s_{i+1}.

   g. If $l(q) = <\bullet>$ where $\bullet \in \neg Act$ and $q \rightarrow q'$, and $s_i \xrightarrow{\bullet'} s_{i+1}$ where $\bullet \neq \bullet'$ and $\bullet' \in Act$, then φ has one positive successor φ' labeled by $<q', s_{i+1}>$.

3. For a negative node φ labeled by $<q, s_i>$:

   a. If $l(q) \in \Gamma p$, then φ is a leaf.

   b. If $l(q) \in \{\vee, <\equiv>\}$ and $\{q' | q \rightarrow q'\} = \{q_1, \ldots, q_m\}$, then φ has negative successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q_j, s_i>$ $(1 \leq j \leq m)$.

   c. If $l(q) = \wedge$, then φ has one negative successor φ' labeled by $<q', s_i>$ for some $q' \in \{q' | q \rightarrow q'\}$.

   d. If $l(q) = X$ and $q \rightarrow q'$ and $\{s' | s_i \xrightarrow{\bullet} s'\} = \{t_1, \ldots, t_m\}$ where $\bullet \in Act$, then φ has negative successors $\varphi_1, \ldots, \varphi_m$ with φ_j labeled by $<q', t_j>$ $(1 \leq j \leq m)$.

   e. If $l(q) = <\bullet>$ where $\bullet \in Act$ and $q \rightarrow q'$, and $s_i \xrightarrow{\bullet} s_{i+1}$, then φ has one negative successor φ' labeled by $<q', s_{i+1,0}>$ where $s_{i+1,0}$ is the initial state of the decomposition TS of s_{i+1}.

   f. If $l(q) = <\bullet>$ where $\bullet \in \neg Act$ and $q \rightarrow q'$, and $s_i \xrightarrow{\bullet'} s_{i+1}$ where $\bullet \neq \bullet'$ and $\bullet' \in Act$, then φ has one negative successor φ' labeled by $<q', s_{i+1}>$.

4. Otherwise, for a positive (negative) node φ labeled by $<q, s_{i,j}>$:

   a. If $l(q) = <\equiv>$ and $\{q' | q \rightarrow q'\} = \{q_1, q_2\}$ such that q_1 is a leaf, and $s_{i,j}$ has a successor $s_{i,j+1}$, then φ has one positive leaf successor φ' labeled by $<q_1, s_{i,j}>$ and one positive (negative) successor φ'' labeled by $<q_2, s_{i,j+1}>$.

   b. If $l(q) = <\equiv>$ and $\{q' | q \rightarrow q'\} = \{q_1, q_2\}$ such that q_1 is a leaf, and $s_{i,j}$ has no successor, then φ has one positive leaf successor φ' labeled by $<q_1, s_{i,j}>$ and one positive (negative) successor φ'' labeled by $<q_2, s_i>$.

c. If $l(q) \in \{\wedge, \vee, X, [S\,C_{Ag}]\}$ and $\{q' | q \rightarrow q'\} = \{q_1\}$, and $s_{i,j} \xrightarrow{r} s_{i,j+1}$ such that $r = l(q)$, then φ has one positive (negative) successor φ' labeled by $< q_1, s_{i,j+1} >$.

The notion of run of an ABTA on a TS is a non-synchronized product graph of the ABTA and TS (see Fig. 3.1). This run uses the label of nodes in the ABTA ($l(q)$), transitions in the ABTA ($q \rightarrow q'$), and transitions in the TS ($s_i \xrightarrow{\bullet} s_j$). The product is not synchronized in the sense that it is possible to use transitions in the ABTA while staying in the same state in the TS (this is the case for example of clauses 2.*a*, 2.*c*, and 2.*d*).

The clause 2.*a* in the definition says that if we have a positive node φ in the product graph such that the corresponding state in the ABTA is labeled with $\neg$ and we have a transition $q \rightarrow q'$ in this ABTA, then φ has one negative successor labeled with $< q', s_i >$. In this case we use a transition from the ABTA and we stay in the same state of the TS. In the case of a positive node and if the current state of the ABTA is labeled with $\wedge$, all the transitions of this current state of the ABTA are used (clause 2.*c*). However, if the current state of the ABTA is labeled with $\vee$, only one arbitrary transition from the ABTA is used (clause 2.*d*). The intuitive idea is that in the case of $\wedge$, all the sub-formulae must be true in order to decide about the formula of the current node of the ABTA. However, in the case of $\vee$ only one sub-formula must be true.

The cases in which a transition of the TS is used are:

1. The current node of the ABTA is labeled with X (which means a next state in the TS). This is the case of clauses 2.*e* and 3.*d*. In this case we use all the transitions from the current state s_i to next states of the TS.
2. The current state of the ABTA and a transition from the current state of the TS are labeled with the same action. This is the case of clauses 2.*f* and 3.*e*. In this case, the current transition of the ABTA and the transition from the current state s_i of the TS to a state $s_{i+1,0}$ of the associated decomposition TS are used. The idea is to start the parsing of the formula coded in the decomposition TS.
3. The current state of the ABTA and a transition from the current state of the TS are labeled with different actions where the state of the ABTA is labeled with a negative formula. This is the case of clauses 2.*g* and 3.*f*. In this case, the formula is satisfied. Consequently, the current transition of the ABTA and the transition from the current state s_i of the TS to a next state s_{i+1} are used. Finally, clauses 4.*a*, 4.*b*, and 4.*c* deal with the case of verifying the structure of the commitment formulae in the sub-TS. In these clauses, transitions $s_{i,j} \xrightarrow{r} s_{i,j+1}$ are used. We note here that when $s_{i,j}$ has no successor, the formula contained in this state is an atomic formula or a boolean formula whose all the sub-formulae are atomic (for example $p \wedge q$ where p and q are atomic).

Example 3.2. Fig. 3.4 illustrates an example of the run of an ABTA. This figure illustrates a part of the automaton $B_{\otimes}$ resulting from the product of the TS of Fig. 3.2 and the ABTA of Fig. 3.3. According to the clause 1 (Definition 3.4), the root is

a positive node and it is labeled by $<\neg, s_0>$ because the label of the ABTA's root is $\neg$ (Fig. 3.3). Consequently, according to the clause 2.*a*, the successor is a negative node and it is labeled by $<\vee, s_0>$. According to the clause 3.*b*, the second node has two negative successors labeled by $<<Ch>, s_0>$ and $<X, s_0>$ etc.

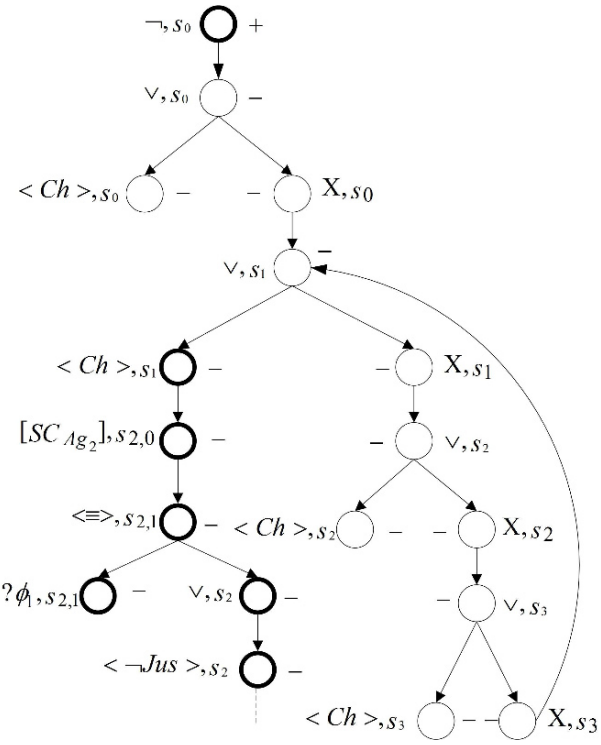

Fig. 3.4 An example of an ABTA's run

In an ABTA, every infinite path has a suffix that contains either positive or negative nodes, but not both. Such a path is referred to as positive in the former case and negative in the latter. Now we can define the notion of accepting runs (or successful runs). Let $p \in \Gamma p$ and let s_i be a state in a TS T. Then $s_i \models_T p$ iff $p \in Lab(s_i)$ and $s_i \models_T \neg p$ iff $p \notin Lab(s_i)$. Let $s_{i,j}$ be a state in a decomposition TS of a TS T. Then $s_{i,j} \models_T p$ iff $p \in Lab'(s_{i,j})$ and $s_{i,j} \models_T \neg p$ iff $p \notin Lab'(s_{i,j})$.

Definition 3.5 (Successful Run). Let r be a run of an ABTA $B = \langle Q, l, \rightarrow, q_0, F\rangle$ on a TS $T = \langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0\rangle$. The run r is successful iff every leaf and every infinite path in r is successful. A successful leaf is defined as follows:

1. A positive leaf labeled by $<q, s_i>$ is successful iff $s_i \models_T l(q)$ or $l(q) = <\bullet>$ where $\bullet \in Act$ and there is no s_j such that $s_i \xrightarrow{\bullet} s_j$.

2. A positive leaf labeled by $<q, s_{i,j}>$ is successful iff $s_{i,j} \models_T l(q)$
3. A negative leaf labeled by $<q, s_i>$ is successful iff $s_i \models_T \neg l(q)$ or $l(q) = <\bullet>$ where $\bullet \in Act$ and there is no s_j such that $s_i \xrightarrow{\bullet} s_j$.
4. A negative leaf labeled by $<q, s_{i,j}>$ is successful iff $s_{i,j} \models_T \neg l(q)$

A successful infinite path is defined as follows:

1. A positive path is successful iff $\forall f \in F, \exists q \in f$ such that q occurs infinitely often in the path. This condition is called the Büchi condition.
2. A negative path is successful iff $\exists f \in F, \forall q \in f, q$ does not occur infinitely often in the path. This condition is called the co-Büchi condition.

We note here that a positive or negative leaf labeled by $<q, s>$ such that $l(q) = <\bullet>$ where $\bullet \in Act$ and there is no s' such that $s \xrightarrow{\bullet} s'$ is considered a successful leaf. The reason is that it is possible to find a transition labeled by $\bullet$ and starting from another state s'' in the TS. In fact, if we consider such a leaf unsuccessful, then even if we find a successful infinite path, the run will be considered unsuccessful, which is false.

An ABTA B accepts a TS T iff there exists a successful run of B on T. In order to compute the successful run of the generating ABTA, we should compute the acceptance states F. For this purpose we use the following definition.

Definition 3.6 (Acceptance States). Let q be a state in an ABTA B and Q the set of all states. Suppose $\phi = \phi_1 U \phi_2 \in q$ [9]. We define the set F_ϕ as follows: $F_\phi = \{q' \in Q | (\phi \notin q' \text{ and } X\phi \notin q') \text{ or } \phi_2 \in q'\}$. The acceptance set F is defined as follows: $F = \{F_\phi | \phi = \phi_1 U \phi_2 \text{ and } \exists q \in B, \phi \in q\}$.

According to this definition, a state that contains the formula ϕ or the formula $X\phi$ is not an acceptance state. The reason is that according to Definition 3.4, there is a transition from a state containing ϕ to a state containing $X\phi$ and vice versa. Therefore, according to Definition 3.5, there is a successful run in the ABTA B. However, we can not decide about the satisfaction of a formula using this run. The reason is that in an infinite cycle including a state containing ϕ and a state containing $X\phi$, we can not be sure that a state containing ϕ_2 is reachable. However, according to the semantics of U, the satisfaction of ϕ needs that a state containing ϕ_2 is reachable while passing by states containing ϕ_1.

Example 3.3. In order to compute the acceptance states of the ABTA of Fig. 3.3, we use the formula associated with the child number (2) in Table 3.2:

$$F(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \wedge G(\neg Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1)\phi_2)))$$

[9] Here we consider until formula because it is the formula that allows paths to be infinite.

We consider this formula, denoted ϕ, instead of the root's formula because its form is $E(\phi)$ (see Section 3.6.2). Consequently, state (1) and states from (3) to (17) are the acceptance states according to Definition 3.6. For example, state (1) is an acceptance state because ϕ and $X\phi$ are not in this state, and state (3) is an acceptance state because ϕ_2 is in this state. States (2) and (4) are not acceptance states. Because only the first state is labeled by $\neg$, all finite and infinite paths are negative paths. Consequently, the only infinite path that is a valid proof of Formula 1 is (1, (2, 4)*). In this path there is no acceptance state that occurs infinitely often. Therefore, this path satisfies the Büchi condition. The path visiting the state (3) and infinitely often the state (9) does not satisfy Formula 1 because there is a challenge action (state (3)), and globally no justification action of the content of the challenged commitment (state (9)).

3.6.4 Model Checking Algorithm (Step 3)

Our model checking algorithm (see Fig. 3.5) for verifying that a dialogue game protocol satisfies a given property and that it respects the decomposition semantics of the underlying communicative acts is inspired by the procedure proposed by [44]. Like the algorithm proposed by [117], our algorithm explores the product graph of an ABTA representing an ACLT* formula and a TS for a dialogue game protocol. This algorithm is on-the-fly (or local) algorithm that consists of checking if a TS is accepted by an ABTA. This ABTA-based model checking is reduced to the emptiness of the Büchi automata [422]. The emptiness problem of automata is to decide, given an automaton A, whether its language $L(A)$ is empty. The language $L(A)$ is the set of words accepted by A.

Let $T = \langle S, Lab, \wp, L, Act, \stackrel{Act}{\longrightarrow}, s_0 \rangle$ be a TS for a dialogue game and let $B = \langle Q, l, \rightarrow, q_0, F \rangle$ be an ABTA for ACTL*. The procedure consists of building the ABTA product $B_\otimes$ of T and B while checking if there is a successful run in $B_\otimes$. The existence of such a run means that the language of $B_\otimes$ is non-empty. The automaton $B_\otimes$ is defined as follows: $B_\otimes = \langle Q \times S, \rightarrow_{B_\otimes}, q_{0B_\otimes}, F_{B_\otimes} \rangle$. There is a transition between two nodes $< q, s >$ and $< q', s' >$ iff there is a transition between these two nodes in some run of B on T. Intuitively, $B_\otimes$ simulates all the runs of the ABTA. The set of accepting states $F_{B_\otimes}$ is defined as follows: $q_{0B_\otimes} \in F_{B_\otimes}$ iff $q \in F$.

Unlike the algorithms proposed in [44, 117], our algorithm uses only one depth-first search (DFS) instead of two. This is due to the fact that our algorithm explores directly the product graph using the sign of the nodes (positive or negative). In addition, our algorithm does not distinguish between recursive and non-recursive nodes. Therefore, we do not take into account the strongly-connected components in the ABTA, but we use a marking algorithm that directly works on the product graph.

The idea of this algorithm is to construct the product graph while exploring it. The construction procedure is directly obtained from Definition 3.4. The algorithm

uses the label of nodes in the ABTA, and the transitions in the product graph obtained from the TS and the ABTA as explained in Definition 3.4. In order to decide if the ABTA contains an infinite successful run, all the explored nodes are marked "visited". Thus, when the algorithm explores a visited node, it returns false if the infinite path is not successful. If the node is not already visited, the algorithm tests if it is a leaf. In this case, it returns false if the node is a non-successful leaf. If the explored node is not a leaf, the algorithm explores recursively the successors of this node. If this node is labeled by "$\wedge$", and signed positively, then it returns false if one of the successors is false. However, if the node is signed negatively, it returns false if all the successors are false. A dual treatment is applied when the node is labeled by "$\vee$".

Example 3.4. In order to check if the language of the automaton illustrated by Fig. 3.4 is empty, we check if there is a successful run. The idea is to verify if $B_{\otimes}$ contains an infinite path visiting the state (3) and infinitely often the state (9) of the ABTA of Fig. 3.3. If such a path exists, then we conclude that Formula 1 is not satisfied by the TS of Fig. 3.2. Indeed, the only infinite path of $B_{\otimes}$ is successful because it does not touch any accepted state and all leaves are also successful. For instance, the leaf labeled by $(<Ch>, s_0)$ is successful since there is no state s_i such that $s_0 \xrightarrow{Ch} s_i$. Therefore, the TS of Fig. 3.2 is accepted by the ABTA of Formula 1. Consequently, this TS satisfies Formula 1 and respects its decomposition semantics.

Soundness and completeness of our model checking method are stated by the following theorem.

Theorem 3.1 (Soundness and Completeness). *Let ψ be a ACTL* formula and B_{ψ} the ABTA obtained by the translation procedure described above, and let $T = \langle S, Lab, \wp, L, Act, \xrightarrow{Act}, s_0\rangle$ be a TS that represents a dialogue game protocol. Then, $s_0 \models_T \psi$ iff T is accepted by B_{ψ}.*

Proof. (**Direction** $\Rightarrow$). To prove that T is accepted by B_{ψ}, we have to prove that there exists a run r of B_{ψ} on T such that all leaves and all infinite paths in the run are successful. Let us assume that $s_0 \models_T \psi$. First, let us suppose that there exists a leaf $<q, s>$ in r such that $s \models_T \neg l(q)$. Since the application of tableau rules does not change the satisfaction of formulae, it follows from Definition 3.4 that $s_0 \models_T \neg\psi$ which contradicts our assumption.
Now, we will prove that all infinite paths are successful. The proof proceeds by contradiction. ψ is a state formula that we can write under the form $E\Phi$, where Φ is a set of path formulae. Let us assume that there exists an unsuccessful infinite path x_r in r and prove that $x_T \models_T \neg\Phi$ where x_T is the path in T that corresponds to x_r (x_r is the product of B_{ψ} and T). The fact that x_r is infinite implies that $R22$ occurs at infinitely many positions in x_r. Because x_r is unsuccessful, $\exists\phi_1, \phi_2, q_i$ such that $\phi_1 U \phi_2 \in q_i$ and $\forall j \geq i$ we have $\phi_2 \notin q_j$. When this formula appears in the ABTA at the position q_i, we have $l(q_i) = \vee$. Thus, according to Definition 3.4 and the form of $R22$, the current node φ_1 of r labeled by $<q_i, s>$ has one successor φ_1 labeled

```
DFS(v = (q, s)): boolean {
    if v marked visited {
      if (sign(v) =  "+" and not accepting(v)) or (sign(v) = "-" and accepting(v))
        return false
    } // end of if v marked visited
    else {
      mark v visited
      switch(l(q)) {
          case (p ∈ Γp):
              switch(sign(v)) {
                  case("+"): if s is a sub-state and l(q) ∉ L'(s) return false
                  case("-"): if s is a sub-state and l(q)) ∉ L'(s) return false
                  case("neutral"): return false
              } // end of switch(sign(v))
          case(∧):
              if s is a leaf return false
              else
                  switch(sign(v)) {
                      case(neutral): for all v'' ∈ {v' / v →B⊗ v'}
                                     if not DFS(v'') return false
                      case("+"): for all v'' ∈ {v' / v →B⊗ v'}
                                 if not DFS(v'') return false
                      case("-"): for all v'' ∈ {v' / v →B⊗ v'}
                                 if DFS(v'') return true else return false
                  } // end of switch(sign (v))
          case(∨):
              if s is a leaf return false
              else
                  switch(sign(v)) {
                      case(neutral): for all v'' ∈ {v' / v →B⊗ v'}
                                     if DFS(v'') return true else return false
                      case("+"): for all v'' ∈ {v' / v →B⊗ v'}
                                 if DFS(v'') return true else return false
                      case("-"): for all v''  {v' / v →B⊗ v'}
                                 if not DFS(v'') return false
                  } // end of switch(sign (v))
          case(<•>):
              if s is a leaf return true
              else for the v'' ∈ {v' / v →B⊗ v'} if not DFS(v'') return false
          case(X, SCAg, <≡>, ?):
              if s is a leaf return false
              else for the v'' ∈ {v' / v →B⊗ v'} if not DFS(v'') return false
      } // end of switch(l(q))
    } // end of else
    return true }
```

Fig. 3.5 The model checking algorithm

by $< q_{i+1}, s >$ with $\phi_1 U \phi_2 \in q_i$ and $\{\phi_1, X(\phi_1 U \phi_2)\} \subseteq q_{i+1}$. Therefore, $l(q_{i+1}) = \wedge$, and φ_2 has a successor φ_3 labeled by $< q_{i+2}, s >$ with $X(\phi_1 U \phi_2) \in q_{i+2}$. Using $R20$ and the fact that $l(q_{i+2}) = X$, the successor φ_4 of φ_3 is labeled by $< q_{i+3}, s' >$ with $\phi_1 U \phi_2 \in q_{i+3}$ and s' is a successor of s. This process will be repeated infinitely since the path is unsuccessful. It follows that there is no s in T such that $s \models_T \phi_2$. Thus, according to the semantics of U, there is no s in T such that $s \models_T \phi_1 U \phi_2$. Therefore, $x_T \models_T \neg \Phi$.
(**Direction** $\Leftarrow$). The proof proceeds by an inductive construction of x_r and an analysis of the different tableau rules. A detailed proof of this theorem is presented in [35].

3.7 Case Studies

In this section, we will exemplify the model checking technique presented in this chapter by means of two case studies: 1) the persuasion/negotiation protocol for agent-based web services ($\mathcal{PNAWS}$) [36]; and 2) the NetBill protocol, a system of micropayments for goods on the Internet [405]. We will also discuss their implementations using an extension of the Concurrency Workbench of New Century (CWB-NC) model checker[10] [107, 446], which has been used to check many large-scale protocols in communication networking and process control systems. As benchmark, we will show the simulation results of these two case studies using the MCMAS model checker [355].

3.7.1 Verifying $\mathcal{PNAWS}$

$\mathcal{PNAWS}$ is a dialogue game-based protocol allowing web services to interact in a negotiation setting via argumentative agents. Agents can negotiate their participation in composite web services and persuade each other to perform some actions such as joining some existing business communities. In this case, two agents are used: the Master agent that manages the community and the Slave agent that is invited to join the community. $\mathcal{PNAWS}$ is specified using two special moves: refusal and acceptance as well as five dialogue games: entry game (to open the interaction), defense game, challenge game, justification game, and attack game. The $\mathcal{PNAWS}$ protocol can be defined as follows using a BNF-like grammar where "|" is the choice symbol and ";" the sequence symbol:

$\mathcal{PNAWS}$ = entry game; defense game; WSDG

[10] The CWB-NC model checker can be downloaded from:
http://www.cs.sunysb.edu/ cwb/

WSDG = acceptance move | CH | ATT
CH = challenge game; justification game; (WSDG | refusal move)
ATT = attack game; (WSDG | refusal move)

Each game is specified by a set of moves using a set of logical rules. Fig. 3.6 illustrates the different actions of this protocol using a finite state machine. Many properties can be checked in this protocol, such as deadlock freedom (a safety property), and liveness (something good will eventually happen). Deadlock freedom means that there is always a possibility for an action and can be expressed as follows, where $Ag \in \{Ag_1, Ag_2\}$:

$$AG(Cr(Ag_1, SC(Ag_1, Ag_2, \phi)) \Rightarrow AF(Action(Ag, SC(Ag_1, Ag_2, \phi)) \vee Action^{+}(Ag, SC(Ag_1, Ag_2, \phi), \phi_1))) \quad (3.2)$$

An example of liveness can be expressed by the following formula stating that if there is a challenge, a justification will eventually follow:

$$AG(Ch(Ag_2, SC(Ag_1, Ag_2, \phi_1)) \Rightarrow F(Jus(Ag_1, SC(Ag_1, Ag_2, \phi_1), \phi_2))) \quad (3.3)$$

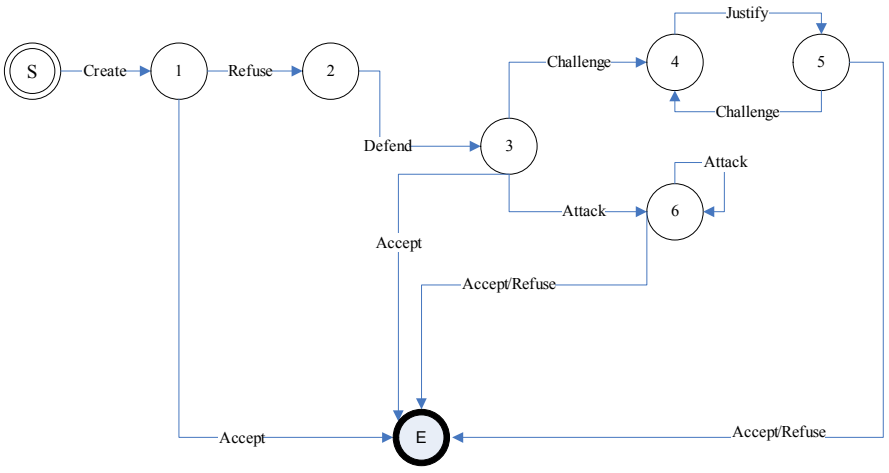

Fig. 3.6 Actions of the $\mathcal{PNAWS}$ protocol

We have extended the CWB-NC model checker by adding SC and argument operators and implemented this case study. CWB-NC supports GCTL*, which is close to our logic (without SC and argument operators) and allows modeling concurrent systems using Calculus of Communicating Systems (CCS) developed in [316]. CCS is a process algebra language, which is a prototype specification language for reactive systems. CCS can be used not only to describe implementations of processes, but also specifications of their expected behaviors. To implement this case study,

CCS is used to describe the model M to be checked by specifying the states and labeled transitions. ACTL* is used to specify the properties and the extended CWB-NC tool takes as input the CCS code and the ACTL* property and automatically builds the dialogue game protocol and checks the property by building the ABTA and executing the model checking algorithm presented in Fig. 3.5 (see the methodology in Fig. 3.1). To use CCS as the design language to describe the $\mathcal{PNAWS}$ protocol, we need first to introduce its syntax. Let A be the set of actions performed on SC we consider in ACTL* logic. For all $a \in A$, we associate a complementary action $'a$. An action a represents the receipt of an input action, while $'a$ represents the deposit of an output action. The syntax is given by the following BNF grammar:

$$P ::= nil|\alpha(\phi).P|(P+P)|(P|P)|proc\ C = P$$

"." represents the prefixing operator, "+" is the choice operator, "|" is the parallel operator and "$proc =$" is used for defining processes. The semantics can be defined using operational semantics in the usual way. $\alpha(\phi).P$ is the processes of performing the action α on the SC content ϕ and then evolves into process P. For representation reasons, we consider only the commitment content and we omit the other arguments. In addition, we abstract away from the internal states and we focus only on the global states. $P+Q$ is the process which non-deterministically makes the choice of evolving into either P or Q. $P|Q$ is the process which evolves in parallel into P and Q. To implement $\mathcal{PNAWS}$, we need to model the protocol and the agents using this protocol (the Master and Slave agents). For this reason, four particular processes should be defined: the states process describing the protocol dynamics; the two agents processes describing the agents legal decisions; and the communication synchronization process. The formulae to be checked are then encoded in CWB-NC input language. A simplified version of the states process is as follows:

```
proc Spec = create(φ).S1
proc Accept = accept(φ).Spec
proc Accept' = 'accept(φ).Spec
proc Refuse = refuse(φ).Spec
proc Refuse' = 'refuse(φ).Spec
proc S1 = 'refuse(φ).S2 + Accept'
proc S2 = defend(φ').S3
proc S3 = 'challenge(φ').S4 + 'attack(φ').S6 + 'accept(φ').Spec
proc S4 = justify(φ).S5
proc S5 = 'challenge(φ).S4 + 'Accept +'Refuse
proc S6 = attack(φ').S7 + Accept +Refuse
proc S7 = 'attack(φ).S6 + 'accept(φ').Spec + 'refuse(φ').Spec
set Internals = {create, challenge, justify, accept,
                 refuse, attack, defend}
```

The Master agent process has the form:

```
proc Master = create(φ).'accept(φ).master
            + create(φ).'refuse(φ).defend(φ).'accept(φ).master
            + create(φ).'refuse(φ).defend(φ).'refuse(φ).master
              ...
```

The Slave agent process has a similar form except the fact that it does not initiate the communication. The process describing the communication synchronization activity of an agent is as follows:

```
proc Ag = 'create(φ).Ag +
  create(φ).('refuse(φ ).Ag + 'accept(φ ).Ag) +
  refuse(φ).(Ag + 'defend( φ').Ag) +
  defend(φ').('challenge(φ').Ag + 'attack(φ).Ag + 'accept(φ').Ag)
+
  challenge(φ).'justify(φ').Ag +
  justify(φ').('challenge(φ').Ag + 'accept(φ').Ag +'refuse(φ').Ag)
+
  attack(φ').('attack(φ).Ag + 'accept(φ').Ag+'refuse(φ').Ag) +
  accept(φ).Ag
```

The model size is $|M| = |S| + |R|$, where $|S|$ is the state space and $|R|$ is the relation space. $|S| = |S_{Ag_1}| \times |S_{Ag_2}| \times |S_{PNAWS}|$, where $|S_{Ag_i}|$ is the number of states for Ag_i and $|S_{PNAWS}|$ is the number of states of the protocol. An agent state is described in terms of the possible actions and each action is described by a set of states. For example, create action needs 2 states, challenge needs 3 states, and justify needs 5 states (see Fig. 3.2). Thus, for each agent we have 35 states. The protocol is described by the legal actions (Fig. 3.6), so it needs 29 states. In total, the number of states needed for this case study is $|S| = 35525 \approx 3.5 \cdot 10^4$. To calculate $|R|$, we have to consider the operators of ACTL* and the actions, where the total number is $6 + 11 = 17$. We can then approximate $|R|$ by $17 \cdot |S|^2$. So we have $|M| \approx 17 \cdot |S|^2 \approx 2 \cdot 10^{10}$. This is a theoretically estimated size if all possible transitions are considered. However, in the implementation, not all these transitions are used. On the other hand, the system considers additional states for the internal coding of variable states and actions. Some simulation results on a laptop Intel Core 2 Duo CPU T6400 2.20 GHz with 3.00 GB of RAM running Windows Vista Home Premium are given in Table 3.3. Fig. 3.7 shows the results screenshot. In fact, CWB-NC does not search the whole model, but it proceeds by simplifying the ABTA, minimizing the sets of accepting states, and computing bisimulation before starting the model checking.

As benchmark, we use MCMAS [355] that supports agent specifications. As discussed in Section 3.2.2, MCMAS is a symbolic model checker based on OBDDS, where the model and formula to be checked are not represented as automata, but using boolean functions. in MCMAS, models are described into a modular language called Interpreted Systems Programming Language (ISPL). An ISPL program includes: 1) a list of agents' descriptions; 2) an evaluation function indicating the

Table 3.3 Statistics of verifying $\mathcal{PNAWS}$ using CWB-NC

Model size (states/transitions)	35709/77244
Time for building the model (sec)	1.763
Verification time (sec)	5.912
Total execution time (sec)	7.675

```
cwb-nc> load PNAWS.ccs
Execution time (user,system,gc,real):(0.016,0.000,0.016,0.016)
cwb-nc> size Spec
Building automaton...
.........1000.........2000.........3000.........4000.........5000
.........6000.........7000.........8000.........9000.........10000
.........11000........12000........13000........14000........15000
.........16000........17000........18000........19000........20000
.........21000........22000........23000........24000........25000
.........26000........27000........28000........29000........30000
.........31000........32000........33000........34000........35000
.....
States: 35709
Transitions: 77244
Done building automaton.
States: 35709
Transitions: 77244
Execution time (user,system,gc,real):(1.763,0.000,0.172,1.763)
cwb-nc> chk -L gctl Spec can_deadlock
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 states.
Simplifying ABTA:
Minimizing sets of accepting states...done
Performing constant propagation...done
Joining operations...done
Shrinking automaton...done
Computing bisimulation...
Done computing bisimulation.
Simplification completed.
Simplified ABTA has 4 states.
Starting ABTA model checker.........*...*.*..*..*.*.*..*.*.*..*.*.*.*.*.*.*.*..
*.*.*.*.*.*.*.*.*.**.*.*..*.*.*.*.*.*.*.*.*.**.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*
.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.**.*.*.*.**.*.*.**.*.**.*.**.***.***.***.****
*****.*
Model checking completed.
Expanded state-space 107127 times.
Stored 112953 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(5.912,0.000,1.393,5.912)
cwb-nc>
```

Fig. 3.7 $\mathcal{PNAWS}$ simulation results with CWB-NC

states where atomic propositions are true; 3) a set of initial states; and 4) a list of formulae. Each agent is composed by: a set of local states, a set of actions, a rule (protocol) describing which action can be performed by the agent, and evolution functions that describe how the local states of the agent evolve based on the current local states and agent's actions.

To implement the $\mathcal{PNAWS}$ protocol with ISPL, commitments are encoded as variables. The Master and Slave agents are specified in two Agent sections along with the Protocol and its Evolution. The atomic propositions are evaluated in Evaluation section. Formulae are then encoded in the same file in the Formulae section. As example, we show here the form of the Master agent:

```
Agent Master
   Vars:
     state : {M0, M1, M2, ...};
     ...
   end Vars
   Actions = {create, defend, ...};
```

```
    Protocol:
      -- initiate the contract by creating
      state = M0 : {create};
      ...
    end Protocol
    Evolution:
      state = M1 if state = M0 and Slave.Action = reject
      ...
    end Evolution
end Agent
```

Some simulation results using the same machine as for CWB-NC are given in Table 3.4. Fig. 3.8 shows the results screenshot. This simulation reveals that MCMAS uses greater number of reachable states, which are needed to encode commitments and agent local states. The execution time is very close to the previous experiment.

Table 3.4 Statistics of verifying $\mathcal{PNAWS}$ using MCMAS

Number of reachable states	39475
Number of BDD and ADD nodes	152093
Total execution time (sec)	8

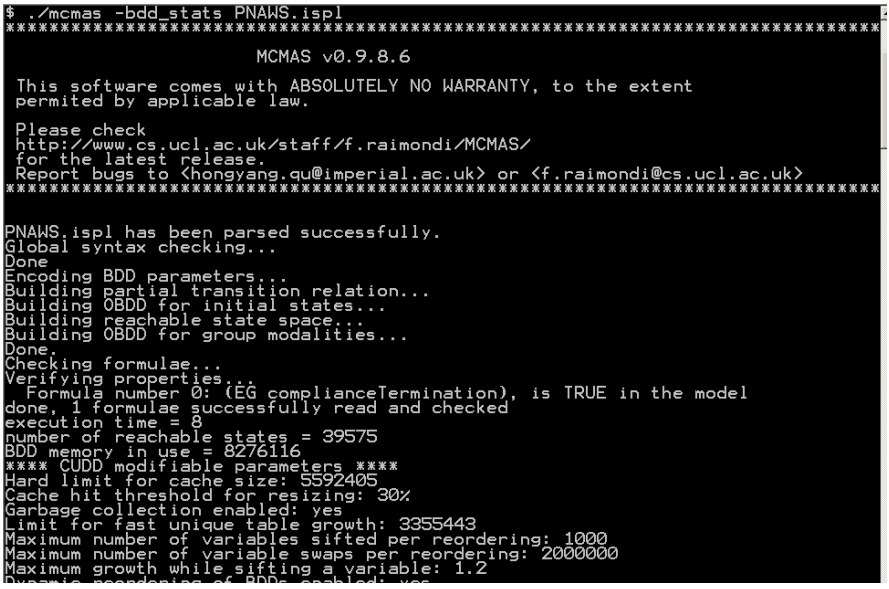

Fig. 3.8 $\mathcal{PNAWS}$ simulation results with MCMAS

3.7.2 Verifying NetBill

We consider a modified version of the NetBill protocol where two agents, Customer (Cus) and Merchant (Mer), are interacting about some goods. The protocol starts when the Customer requests a quote, which means creating a commitment about a content ϕ_1. The merchant can then either reject the request, which means refuse the commitment and the protocol will end, or accept the request (i.e. accept the commitment) and then make an offer (i.e. create another commitment about a content ϕ_2). The protocol is self-described in Fig.3.9. An example of liveness in this protocol can be expressed by the following formula stating that if a commitment is created, then there is a possibility of satisfying it.

$$AG(Cr(Ag_1, SC(Ag_1, Ag_2, \phi_1)) \Rightarrow EF(Sat(Ag_1, SC(Ag_1, Ag_2, \phi_1))) \tag{3.4}$$

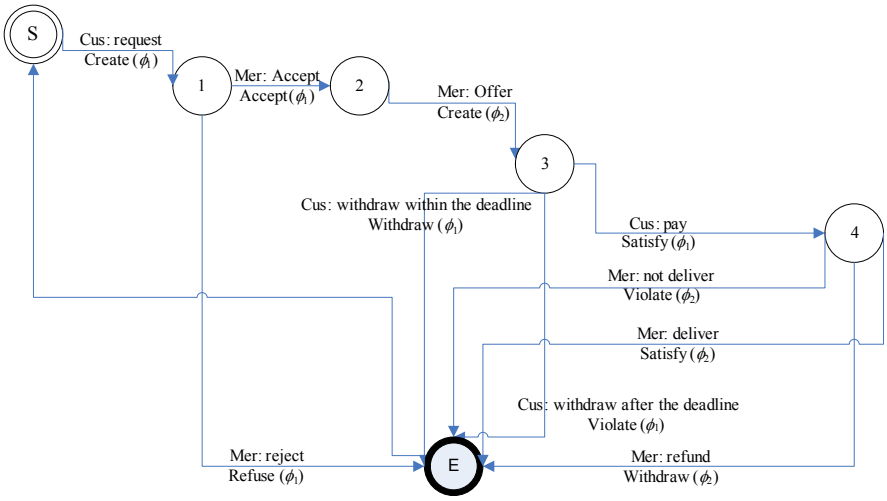

Fig. 3.9 Actions of the NetBill protocol

NetBill size is $|M| = (|S_{Ag_1}| \times |S_{Ag_2}| \times |S_{NetBill}|) + |R|$, where Ag_1 is the Customer and Ag_2 is the Merchant. According to the actions the Customer and Merchant are allowed to perform, we have $|S_{Ag_1}| = 9$ and $|S_{Ag_2}| = 13$. The NetBill protocol is described by the legal actions, and by considering the size of each action, we obtain $|S_{NetBill}| = 22$. In total, the number of states needed for this case study is $|S| = 2574 \approx 2.5 \cdot 10^3$. As we did in the previous case study, the theoretical estimation of $|R|$ if all possible transitions are considered is $|R| \approx 17 \cdot |S|^2$. So we have $|M| \approx 10^{10}$. As illustrated in Table 3.5, which shows the NetBill simulation results with CWB-NC using the same machine as in the previous case study, the number of transitions that are effectively considered is much more smaller. Table 3.6 shows the simulation

results with MCMAS. Fig. 3.10 shows the results screenshot with the two model checkers. Because NetBill is 14 times smaller than *PNAWS*, its execution time is shorter.

```
cwb-nc> load NetBillnew.ccs
Execution time (user,system,gc,real):(0.016,0.000,0.000,0.016)
cwb-nc> load formula.gctl
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.000)
cwb-nc>
cwb-nc>
cwb-nc> size Spec
Building automaton...
.........1000.........2000.....
States: 2593
Transitions: 5911
Done building automaton.
States: 2593
Transitions: 5911
Execution time (user,system,gc,real):(0.125,0.000,0.000,0.125)
cwb-nc>
cwb-nc>
cwb-nc> chk -L gctl Spec can_deadlock
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 states.
Simplifying ABTA:
Minimizing sets of accepting states...done
Performing constant propagation...done
Joining operations...done
Shrinking automaton...done
Computing bisimulation...
Done computing bisimulation.
Simplification completed.
Simplified ABTA has 4 states.
Starting ABTA model checker....*.*.**.*.***
Model checking completed.
Expanded state-space 7779 times.
Stored 8504 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.359,0.000,0.046,0.359)
cwb-nc>
```

```
$ ./mcmas -bdd_stats NetBill.ispl
********************************************************************************

                        MCMAS v0.9.8.6

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check
 http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/
 for the latest release.
 Report bugs to <hongyang.qu@imperial.ac.uk> or <f.raimondi@cs.ucl.ac.uk>
********************************************************************************


NetBill.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Done.
Checking formulae...
Verifying properties...
  Formula number 0: (AF purchase_violation), is FALSE in the model
  Formula number 1: (AF purchase_compliance), is FALSE in the model
done, 2 formulae successfully read and checked
execution time = 0
number of reachable states = 2851
BDD memory in use = 6537844
**** CUDD modifiable parameters ****
Hard limit for cache size: 5592405
Cache hit threshold for resizing: 30%
Garbage collection enabled: yes
Limit for fast unique table growth: 3355443
Maximum number of variables sifted per reordering: 1000
```

Fig. 3.10 Simulation results of NetBill

Table 3.5 Statistics of verifying NetBill using CWB-NC

Model size (states/transitions)	2593/5911
Time for building the model (sec)	0.125
Verification time (sec)	0.359
Total execution time (sec)	0.484

Table 3.6 Statistics of verifying NetBill using MCMAS

Number of reachable states	2851
Number of BDD and ADD nodes	9332
Total execution time (sec)	≈ 0.5

3.8 Discussion and Future Work

Model checking is an effective technique to verify finite state systems. Compared to classical software systems, model checking multi-agent systems raise new challenges related to the need of considering: 1) epistemic properties where the semantics is expressed in terms of accessibility relations; and 2) agent communication protocols that integrate agent properties and message meaning, which make them more complex than simple message exchanging mechanisms. These two fundamental issues need new and efficient verification techniques considering computational interpretations of accessibility relations and message meaning.

In this chapter we described a verification technique for dialogue game protocols. The proposed model checking algorithm allows us to verify both protocols' correctness and agents' compliance to the decomposition semantics of communicative acts. This technique uses a combination of automata and tableau-based algorithms to verify temporal and action specifications. The formal properties to be verified are expressed in ACTL* logic and translated to ABTA using tableau rules. Our model checking algorithm that works on a product graph is an efficient on-the-fly procedure.

The field of automatic verification of multi-agent systems has manifested significant advances in the past few years, as efficient algorithms and techniques have been proposed. However, many issues still need investigations. The most challenging among them are: 1) verifying the compliance of agents' joint actions to the norms and rules of the multi-agent system in which they operate; 2) integrating the verification of mental and social attitudes in the same framework; 3) allowing the use of expressive logical languages to specify agents and their communication and coordination, multi-agent environments, and requirements (i.e. desired properties); and 4) developing tools integrating the whole aforementioned issues.

We plan to extend this work to address some of these issues. In fact, we intend to use the proposed tableau-based technique to verify MAS specifications and the conformance of agents to these specifications. We also plan to extend the technique and logic in order to consider epistemic properties, so that we will have a same framework for private and social attitudes. We plan to use this technique to specify

and verify the compliance of agents' actions to the norms and policies of multi-agent systems. Although the technique discussed in this chapter is computationally efficient, it has the problem of state explosion. For this reason, we plan to consider symbolic and bounded model checking to verify agent commitments and their dialogue games. We are investigating the extension of the MCMAS model checker to integrate LTL logic with commitment modalities and action formulae, so it will be possible to symbolically model check dialogue games with ACTL* logic.

Acknowledgement

We would like to thank the reviewers for their valuable comments and suggestions. Jamal Bentahar would like to thank Natural Sciences and Engineering Research Council of Canada (NSERC: Project 341422-07), Fonds québécois de la recherche sur la nature et les technologies (FQRNT: Project 2008-NC-119348) and Fonds québécois de la recherche sur la société et la culture (FQRSC: Project 2007- 111881) for their financial support.

Chapter 4
Directions for Agent Model Checking*

R.H. Bordini, L.A. Dennis, B. Farwer, and M. Fisher

Abstract In this chapter we provide a perspective on current and future work in the area of *agent model-checking*. In particular, we describe our approach, which was the first to provide comprehensive verification of practical agent programming languages. It provides a library of general agent concepts that has been formally defined and implemented in Java, upon which interpreters for various agent programming languages can be succinctly programmed. The Java library has been prepared so that it can be efficiently used with an existing Java model checker, thus facilitating the verification of (heterogeneous) multi-agent programs. Besides giving an overview of our approach, in this chapter we identify its current shortfalls and discuss where we aim to target future development.

R.H. Bordini
Institute of Informatics, Federal University of Rio Grande do Sul, Brazil e-mail: r.bordini@inf.ufrgs.br

L.A. Dennis
Department of Computer Science, University of Liverpool, U.K. e-mail: l.a.dennis@liverpool.ac.uk

B. Farwer
School of Engineering and Computing Sciences, Durham University, U.K. e-mail: berndt.farwer@durham.ac.uk

M. Fisher
Department of Computer Science, University of Liverpool, U.K. e-mail: mfisher@liverpool.ac.uk

* This work was partially supported by EPSRC, through projects EP/D054788 and EP/D052548.

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_4,

4.1 Introduction

4.1.1 Agents and Rational Agents

While the key aspect of an object is *encapsulation* of state and (some) behaviour, agents are truly *autonomous*. Thus, an agent not only has control over its own state, but also can dynamically change its patterns of behaviour and communication as execution progresses (for example by "learning" or "adapting"), and so can choose what form of interactions it presents to its environment (including other agents). In particular, rational agents continuously choose the *goals* they will target and the *courses of action* they will take in order to achieve these. In this sense, the agent abstraction captures the core elements of *autonomy* and has been successfully used to model/develop autonomous systems at a high level.

We are specifically concerned with *rational agents* [86, 110, 361, 438]. Since an agent is autonomous, it must have some *motivation* for acting in the way it does. The key aspect of a rational agent is that the decisions the rational agent makes, based on these dynamic motivations, should be both "reasonable" and "justifiable". Just as the use of agents is now seen as an essential tool in representing, understanding, and implementing complex software systems, so the characterisation of complex components as rational agents allows the system designer to work at a much higher level of abstraction. Since we are particularly concerned with the deliberative aspects, we can also term the rational agents we examine as *deliberative* agents.

Agents of the above form, autonomously (and asynchronously) executing, reside in an environment consisting of other agents. Typically, the only direct interaction between such agents occurs through message-passing and, through this simple communication mechanism, agents can be organised into a variety of structures. As it is often the case that agents must work together, these structures typically support *cooperative activity* or *teamwork*.

One of the best known and most studied agent *architectures* is known as the BDI architecture [358, 359, 361], where BDI stands for "belief-desire-intention". In the BDI tradition, an agent's architecture is composed of three main parts: *beliefs* represent the information available to an agent about its environment and the other agents sharing such environment; *desires* (or goals) represent preferred states of affairs, and *intentions* represent either a subset of the desires that the agent is committed to bringing about or, more often, the particular choices of courses of action that the agent has made expecting that they will bring about (a subset of) the desired states of affairs. Such agent architectures will typically also contain (at least in practice) a *plan library* which represents the agent's *know-how*; it contains "recipes for action" associated with the particular goals they are meant to bring about and the particular circumstances in which that is expected to happen, and agents use such recipes to form intentions. There has also been considerable work on BDI logics [362], in particular multi-modal logics built on top of a temporal logic where different modalities

are used to represent an individual agent's *mental attitudes*: beliefs are *informational* attitudes, desires are *motivational* attitudes, and intentions are *deliberative* attitudes.

Practical BDI agents are implemented as "reactive planning systems"[2] [191]: they run continuously, reacting to events (such as perceived changes in the environment and new goals to achieve) by executing *plans* given by the programmer. As mentioned above, plans are *courses of action* that agents commit to executing in order to achieve their goals; agents have a repertoire of such 'actions' that they are able to perform in order to change the environment, such as a robot changing its location. The pro-active behaviour of agents is possible through the notion of *goals* (i.e., desired or preferred states of the world) that are also part of the language in which plans are written.

4.1.2 Logical Agent Descriptions

In representing an individual agent's behaviour as well as the properties we want to verify about these behaviours, we choose to utilise languages based on *formal logic*. The advantages of following such an approach is that:

- the notation has a well-defined, and usually well understood, semantics;
- it provides a high-level, yet concise, form of description consisting of a small range of powerful constructs;
- it allows us to model not only static aspects of an agent state but also the *dynamic* behaviour of agents;
- there is a uniformity of style between the description of the agent behaviour and the properties we want to verify;
- it imposes few operational constraints upon the system designer; and
- allows us to narrow the gap between the agent descriptions and agent theory in that the semantics of an agent is close to that of its logical description.

The use of formal logic thus allows for the possibility of uniformly employing both specification and verification techniques that are theoretically well founded to the development of agent-based systems.

As we are interested in *rational agents*, we impose additional requirements for describing, and reasoning about, rational behaviour at a high level. We note that the predominant rational agent theories all share similar elements, in particular:

- an *informational* component, such as being able to represent an agent's *beliefs* or *knowledge*;

[2] Many researchers do not like this term because in reactive planning systems no "planning" as such (i.e., reasoning from first principles) takes place.

- a *motivational* component, representing the agent's *desires* or *goals*;
- a *dynamic* component, allowing the representation of agent activity (e.g., in the form of *plans*); and
- a *deliberative* component, representing choices the agent has made (e.g., in the form of *intentions*).

Although a variety of different approaches exist, such aspects are typically represented logically by temporal or dynamic logics (dynamism), and modal logics of intentions, desires/goals, or wishes (motivation). Thus, the predominant approaches to rational agent theory use relevant combinations of such formalisms, for example the BDI model [362] uses branching-time temporal logic (CTL$*$) combined with modal logics of belief (KD45), desire (KD) and intention (KD), while the KARO framework [315] uses dynamic logic (PDL) combined with a modal logic of wishes (KD).

4.1.3 Formal Verification and Model Checking

By *formal verification* we mean carrying out a mathematical analysis in order to assess all the possible behaviours of a system. The first rigorous attempt at proving correctness of sequential programs dates back to Floyd [181]. A number of formal verification techniques were subsequently developed within Computer Science in order to assess the behaviour of complex, interacting, distributed, uncertain computational processes. These techniques are beginning to be used outside such areas. In formal verification, the *properties* or behavioural requirements we have of complex systems can be *specified* using formulæ from an appropriate formal logic. Importantly, the formal logic used can incorporate a wide range of concepts matching the view of the system being described, for example *time* for dynamic/evolving systems, *probability* for systems incorporating uncertainty, *goals* for autonomous systems, and so forth. This gives great flexibility in the property specification languages that can be used.

Given such a *specification*[3], we can check this against models/views of the system under consideration in a number of ways. The most popular of these is *model checking* [102, 247], where the specification is checked against all possible executions of the system; if a representation for all such executions can be achieved with a *finite* number of system *states*, then this check can often be completely carried out automatically. Indeed, the verification, via model checking, of both hardware systems (such as chip designs) and software systems (such as device drivers) has been very successful in industry as well as academia [26, 43], and has led to the recent application of verification techniques to autonomous agents [71, 73, 355, 429].

[3] Note that in the model checking literature, "specification" refers to the formulæ representing the properties we want the system to satisfy.

Formal Verification in a Nutshell.

Imagine that we have a logical formula, φ, that is used to specify some property that we wish to check of an agent. Now, we have to check this against a description of the behaviour of such an agent (and its environment, including other agents). One (*deductive*) approach is to have another logical formula, say Γ, that *exactly* specifies the agent system (e.g., derived via a formal semantics). Thus, Γ must characterise all the (possibly infinite number of) models (or executions) of the agent. To check that the property holds, we must prove

$$\vdash (\Gamma \Rightarrow \varphi)$$

This means establishing that

> the set of models/executions that satisfy Γ is a *subset* of the set of models/executions that satisfy φ.

If, on the other hand, we can represent all possible executions of the agent system using a relatively small (or, at least, finite) number of states that the system can be at in a given moment in time, then it makes sense to check all these individually. Thus, in this case, we only need to check whether $\Sigma \models \varphi$, where the structure Σ describes all possible system executions. Thus, the problem is to determine whether:

$$\forall \sigma \in \Sigma.\, \langle \sigma, 0 \rangle \models \varphi$$

where $\langle \sigma, 0 \rangle$ is the initial state of the execution sequence σ. This (*algorithmic*) approach to verification is called *model checking* [101], whereas the deductive approach is called *theorem proving* [332].

So, in order to carry out formal verification by model checking, formal descriptions of both the system being considered and the properties to be verified are required. We have been developing techniques for verifying multi-agent systems over a number of years [67, 71, 73]. As we mentioned earlier, in our work, we are interested in rational agents, particularly where 'intelligent' choices are required. In particular, we have previously looked at the verification of systems programmed in the BDI agent language AgentSpeak/*Jason* [76,357] where the properties to be verified were given in a simple BDI-like logic. We later worked on a new approach that aims at allowing the model checking of various (BDI) agent languages [141] using the JPF Java model checker [265], which is what this chapter focuses on. An advantage of using a Java model checker is that many agent platforms are already built on top of Java, and most realistic multi-agent systems will make significant use of legacy code (often in Java) so, with the successful use of abstraction techniques, it might be possible to (slowly) verify the complete system including all legacy code.

4.1.4 Program Verification

In our approach, one important aspect is that the system model is obtained directly from the program that is used when we run the system. This approach is called *program model checking* [426] and is a very active area of research within software verification. This is different from some other approaches in which a model of the system is (manually) created using a specific language, such as the Promela language used by the SPIN model checking tool [237]. The advantage of creating a model of the system (not necessarily a software system) from scratch, using a language such as Promela, is that the language itself facilitates the creation of a more abstract model of the system, which will often be required for full verification by model checking. The disadvantage is that if the model checker guarantees that the model of the system satisfies a given property, the question of whether the model of the system correctly describes all relevant behaviours of the system may still remain.

Whilst this is not a problem in program model checking (because we are, after all, model checking the same source that is used to run the system), the problem here is that the state space of any reasonably complex software will turn out too big for practical model checking. Fortunately, there is a vast literature on state-space reduction techniques that apply directly at the level of programming languages. We comment further on this in Section 4.4.

Another important advantage of model checking used in this way, is that model checkers are often used in the software industry as a more general development tool, even if full verification by model checking turns out not to be practical. For example, model checkers can be very useful for debugging and test case generation.

4.1.5 Agent Programming Languages

There are many ways in which logic can be used to specify (and so provide semantics for) agent-based systems. However, we are also interested in *building* agent-based systems using techniques based upon logic. There is a wide variety of such languages [64, 65], though we essentially concentrate on mechanisms for implementing *rational agents* in a logically clear and justifiable way.

An important aspect of agent programming languages is the explicit representation of *goals*. Agents deliberate on which goals to pursue, and use plans in order to achieve them. What an explicit representation of goals provides is the ability for agents to behave *rationally* in the sense that, if a plan fails to achieve the goal it was meant to achieve, for example because the environment changed in an unpredictable way, the agent will not simply carry on executing as if the plan had been successful. The agent will possibly try to use a different means to achieve the same goal, or may drop the goal if it believes that is the rational thing to do. That is an important part of agents' autonomy.

Another important aspect of most BDI languages is a data structure called the *set of intentions*. Intentions represent particular courses of action (i.e., instances of plans from the agent's plan library) that the agent has already committed itself to performing. Different intentions in this set typically represent different foci of attention for the agent. A partly executed plan will also typically contain further goals the agent should adopt in due course. The autonomous choice of which course of action to use in order to achieve those goals is left as late as possible, so that the agent can use its most up-to-date information about the state of the environment in order to choose the best course of action. Of course, in very dynamic environments, plans can still fail to achieve the goal they were meant to achieve, hence such environments require the type of rational deliberation discussed above.

In the area of autonomous agents and multi-agent systems, AgentSpeak is one of the best known agent-oriented programming languages based on the BDI architecture. It is a logic-based agent-oriented programming language introduced by Rao [357], and subsequently extended and formalised in a series of papers by Bordini and colleagues[4]. Various *ad hoc* implementations of BDI-based (or "goal-directed") systems exist, but one important characteristic of AgentSpeak is its theoretical foundation and formal semantics.

Our approach, outlined in the next section, evolved from our previous work on the development of model checking techniques for AgentSpeak programs [71, 73]. However, there is a variety of agent programming languages [64, 65, 175] and, in order not to preclude these, we have chosen to design a new approach so that it can be used for a large number of languages.

4.2 Our Approach

Our approach is described in more detail in [67]; a summary is given in Figure 4.1.

A multi-agent program, originally programmed in some agent programming language and interpreted in the *Agent Infrastructure Layer* (AIL) platform, is represented in Figure 4.1. It uses *data structures* to store its internal state comprising, for instance, a belief base, a plan library, a current intention, and a set of intentions (as well as other temporary state information). It also uses a specific interpreter for the agent programming language that is built using AIL classes and methods. The interpreter defines the reasoning cycle for the agent programming language which interacts with the model checker, essentially notifying it when a new state is reached that is relevant for verification.

The model checker used is a customised/extended version of the open-source Java model checker JPF (JavaPathFinder) [426] called AJPF (*Agent JPF*). The AIL interpreted program is paired with a *property* to form a product automaton in AJPF.

[4] *Jason* [76] is a Java-based platform for the development of multi-agent systems using a variant of AgentSpeak — `http://jason.sf.net`.

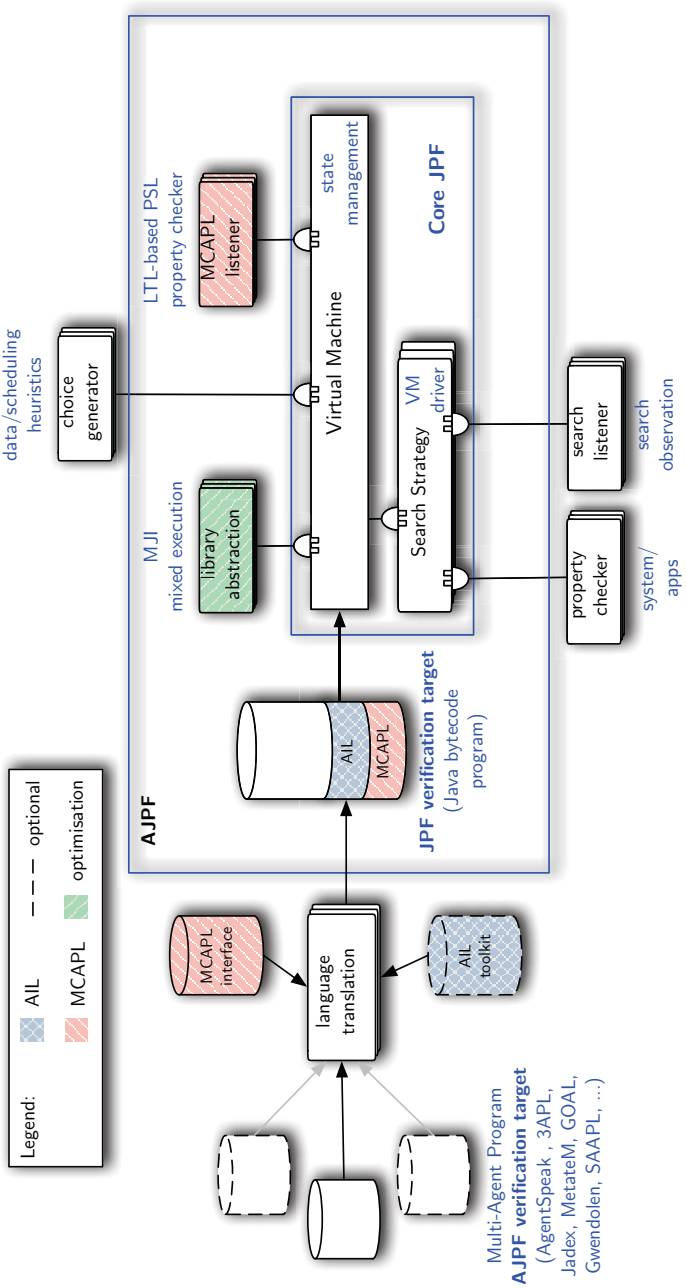

Fig. 4.1 Outline of Our Approach [67].

This product automaton is then executed by the virtual machine — a Java virtual machine specially designed to maintain backtrack points and explore, for example, all possible thread scheduling options (that can affect the result of the verification) [426]. An important feature of JPF for our work is that it can be extended with *listeners* that can be used to check for customised properties and prune the search space. AJPF provides a listener that checks for reachability of the product automaton and prunes branches which are guaranteed to succeed (and so cannot contribute to any error).

4.2.1 AIL: Mapping Agent Languages to a Common Basis

The *Agent Infrastructure Layer (AIL) toolkit* was introduced as a uniform framework for constructing interpreters for various agent programming languages [141], easing the integration of new languages into an existing execution and verification engine. It provides an effective, high-level basis for implementing the operational semantics of BDI-like programming languages. It is important to note that the AIL is *not* intended as a new language in its own right, but as an intermediate layer incorporating the main features of practical agent languages. The key *operations* of many (BDI-)languages, together with a set of standard rules, form the *AIL toolkit* that can be used to create interpreters for particular agent languages. It is also possible to add custom rules for specific languages built from the basic operations made available. These operations and rules have formal semantics and are implemented in Java.

The Agent Infrastructure Layer (AIL) [141] encompasses the main concepts from a wide range of agent programming languages. Technically speaking, it is a collection of Java classes that: (i) enables implementation of interpreters for various agent languages, (ii) contains adaptable, clear semantics, and (iii) can be verified through AJPF. AJPF is a customisation/extension of JPF for model checking agent programs with properties expressed in terms of temporal modalities, beliefs, goals, and intentions.

Common to all language interpreters implemented using AIL methods are the AIL-agent data structures for beliefs, intentions, goals, etc., which are accessed by the model checker and on which the modalities of the *property specification language* are defined. The implicit data structures of a target (BDI) language need to be translated into the AIL's data structures. In particular the initial state of an agent has to be translated into an AIL agent state.

We assume that agents programmed in an agent programming language all possess a reasoning cycle consisting of at least one, but typically several, stages (a reasoning cycle can often be broken down into various identifiable stages that help formalisation and understanding). Each stage is a disjunction of rules that define how an agent's state may change during the execution of that stage. The combined rules of the stages of the reasoning cycle define the operational semantics of that

language. The construction of an interpreter for a language involves the implementation of these rules (which in some cases might simply make reference to the pre-implemented rules) and a reasoning cycle. This means that the AIL can be viewed as a collection of Java classes/methods that provide the building blocks for custom programming of agent language interpreters, with the particular advantage of making the use of model checking techniques for verification directly available (and more efficient). In this way, we can implement an interpreter for an agent language following its operational semantics but using the high-level AIL operations on the agent data structures rather than using Java from scratch.

4.2.2 AJPF: Specialising the AIL and JPF to work together

Our previous approaches to model checking agent programs showed that encoding agent concepts, such as goals and beliefs, into the state machine of a model checker was a complex and time-consuming task. It was also necessary to adapt the property language of a model checker to express agent properties in these terms (the natural terminology for reasoning about an agent-based program). Our approach now is to encode the relevant concepts of the AIL into the model checker just once and then allow multiple languages to benefit from the encoding by utilising the AIL classes in their implementation. Each language will have its own interpreter, essentially a mapping of the language's operational semantics to the operational rules from the AIL toolkit. The AIL therefore consists of data structures for representing agents, beliefs, plans, etc., which can be adapted to the operational semantics of individual languages. Our first work in this direction was precisely to define (BDI) agent-related data structures that are general enough to suit a number of the best known agent programming languages [141]. A language interpreter implemented using the AIL then extends the AIL's agent class and specifies a *reasoning cycle* for the language. The reasoning cycle consists of a transition system which defines a number of stages and specifies the changes to the agent structure that occur as it passes from one stage to the next.

Our approach uses AIL as the basis and specialises a target model checker for use on AIL data structures. The AIL can be viewed as a platform on which agents programmed in different programming languages co-exist, and together with AJPF this provides uniform model checking techniques for various agent-oriented programming languages [67], also allowing the verification of heterogeneous systems — i.e., multi-agent systems comprising agents programmed in different languages [142].

The AIL has been implemented to exploit various techniques for improving the efficiency of model checking with JPF. For instance, it avoids the use of data structures, such as Java Stacks, that cause the JPF search space to branch when operated upon. It also makes use of more brute force methods (such as clustering statements together into uninterruptable blocks) to avoid search space branching in some particular contexts and eliminates certain fields from consideration as part of the state

in order to maximise the identification of states which are identical from the point of view of the multi-agent system, even if they have housekeeping differences at the Java level.

4.2.2.1 The Property Specification Language

AJPF provides a *Linear Temporal Logic* (LTL) based property specification language for defining properties to be verified against a program. This language is extended with simple modalities for belief, etc.

The same property specification language is used whenever AJPF is used but its semantics depends upon the language interpreter. All AIL based interpreters implement the same semantics for the property specification language.

4.2.2.2 AJPF Interfaces for Verification

AJPF provides a set of interface classes. These allow agents to be model checked using AJPF — even if no AIL-based interpreter for that language has been developed — by interfacing with the original interpreter for that language. AJPF can model check multi-agent systems comprising any agents and an environment that comply with these interfaces.

The interface primarily requires that an agent provides "hooks" indicating what in that particular agent language is considered to be a "turn" of the reasoning cycle. AJPF checks that the system meets its specification after each agent has completed such a turn. Each agent is executed in a separate thread and, in this way, AJPF examines all possible scheduling options among the turns of the agents' reasoning cycles. The environment can also run in a separate thread, if desired.

Implementations of the AJPF interfaces must define the required operators of the property specification language. For instance, agents implementing the AJPF agent interface must provide a method which succeeds when they believe the given parameter (represented as a "formula") is true. In this way, the implementation of such a method effectively implements the semantics for the *belief* modality in our supported property specification language. This needs to be done by the users of our verification framework wishing to model check programs in languages for which there is no AIL-based interpreter, and it is their responsibility to ensure that the interface implementations correspond, for their language, to the same semantics of those modalities for AIL. The AIL itself implements these interfaces and so defines an AIL-specific semantics for the property specification language; implementations of AIL interpreters must ensure that their use of the AIL makes the semantics of the properties consistent with their own semantics of those modalities.

The AJPF interface allows programming languages that do not have their own AIL-based interpreters to be used in a system to be model checked against specifications written in the same property specification language that is used for the AIL.

However, those systems will not benefit from the JPF specific efficiency improvements incorporated within the AIL.

4.2.3 Current Status

The overall approach has been designed and implemented. It has been tested on some small multi-agent programs: variations of the *contract net protocol* [407] and auction systems, but with five or fewer agents [431].

The efficacy and generality of the AIL toolkit has also been established by the implementation of a variety of different agent programming languages and the verification of multi-agent systems implemented in those languages. Interpreters have been implemented for Gwendolen [139], GOAL [60], ORWELL [131] and SAAPL [435]. Interpreters for AgentSpeak [76] and 3APL [132] are also being developed.

Multi-agent programs (albeit small ones) written in each of the above programming languages have been verified, although all those programs have had to be manually translated into Java code compatible with AIL/AJPF (see below), as automatic translators are currently being developed. Interestingly, a *heterogeneous* multi-agent systems has been verified as reported in [142]. There, a simple contract net comprising agents programmed in Gwendolen, GOAL, and SAAPL was formally verified using our approach.

The AIL and AJPF, together with interpreters for Gwendolen, GOAL, SAAPL, and ORWELL and a number of case studies and examples are available open source from Sourceforge (`http://mcapl.sourceforge.net`).

4.3 Obstacles

While the basic approach has been implemented and tested on some simple scenarios, a number of issues still remain. In this section, we discuss some of these.

4.3.1 Performance

A typical problem in model checking, particularly of concurrent systems (e.g., where various autonomous entities have independent, yet interacting, behaviour), is that of the *state-space explosion*. The model checker needs to build an in-memory model of the states of the system, and the number of such states grows exponentially, for example, on the number of different autonomous entities being modelled.

Even with refined representation techniques, such as OBDDs used in *symbolic model checking*, the formulæ/structures required to represent the state spaces of realistic systems are huge. JPF is an explicit-state, on-the-fly model checker, and a further problem is that the underlying JPF virtual machine is rather slow. On the other hand, JPF has various sophisticated mechanisms to allow for efficient program model checking, and it works directly on Java bytecodes as input, so the whole of the Java language can in principle be used. In the context of so many Java-based agent development frameworks, this is a very important aspect in favour of JPF.

Overall, our current verification system is relatively slow. For example, in [431], basic properties of simple auction examples comprising just 5 agents can take over 2 hours to verify. Although speed is the main problem, space required can also be problematic [67] (though note that the slow example above actually explores less than 10000 states in total). On the other hand, if we compare these to the results we obtained with JPF in our previous work [73], it is worth noting that JPF seems to be orders of magnitude more efficient than it used to be. We should also note that the success of program model checking relies a great deal on *state-space reduction techniques* that we have not yet developed (to work at the AIL level; see Sections 4.4.2 and 4.4.3).

4.3.2 Target Agent Languages

The AIL has been designed to make the development of interpreters for agent programming languages as simple as possible. However, it does not (and could not) make the task trivial. For example, the AIL assumes a reasoning cycle that passes through explicit stages, which is indicated by a flag in an agent's state. The operational semantics of agent languages are not necessarily expressed in this style, and so the semantics might need to be adjusted, raising questions about the fidelity of the translation and implementation.

Similarly, since JPF does not assume a fair thread scheduler, it is vital for the execution of multi-agent programs that agent threads are explicitly sent to sleep when they have nothing to do. This forces the inclusion of rules in the operational semantics which describe this behaviour. Such rules are commonly omitted from abstract language descriptions, so again this raises questions of fidelity to the language formalisation and opens the possibility that a program operating in the interpreter may behave differently from one in the original language implementation because its "sleeping behaviour" may vary. However, this is very unlikely given that, even though rules for such sleeping behaviour are omitted from formalisations, any practical agent platform would need to ensure those rules are implemented for obvious efficiency reasons (i.e., to avoid "busy waiting"). While the AJPF interface allows an original interpreter to be model checked directly, this is likely to increase the running time for model checking even further, unless the interpreter implementation is customised to JPF in a similar way that the AIL has been, which is considerable technical work (hence the usefulness of AIL).

4.3.3 Using Agent Model Checking

Given the current limitations of the model checking process we anticipate that it will not be used in its present form to check the final code of large multi-agent systems. However, we believe it will be possible to check scaled down versions of such systems, for instance, to check the behaviour of small systems containing just one or two representatives of each type of agent, or to test out particular parts of the code such as communication protocols. We should emphasise again that when combined with future work on state-space reduction techniques the chances of using the approach for model checking larger systems will be significantly increased.

Our approach is used for model checking the agent actions within a simulated environment, also written in Java which, again, will typically be an abstraction of the actual environment in which agents will be running. On the other hand, many multi-agent systems use a Java-based purpose-built simulated environment, in which case the same environment can potentially be used.

We have also found model checking useful in debugging the interpreter implementations and operational semantics of agent languages (besides the applications developed in those languages as originally intended). Particularly, our model checking tool has been useful in terms of locating potential deadlocks caused by the operational semantics of agent languages or their implementations.

Very much as it happens with model checking traditional software, we also expect our work to be useful for multi-agent systems developers to other purposes besides full verification. Model checkers are very useful tools for use during debugging, and automatic input generation [427] for software testing.

4.3.4 Applicability

The systems we have used so far as case studies for AJPF have been modest in scale and using traditional multi-agent scenarios (e.g., simplified contract-net and auction systems). We have not explored formalised environmental "artifacts" [367], nor how non-computational agents (such as humans) might be modelled in the system. The AIL has support for the formation of groups of agents in the style of MetateM [143], but this support is yet to be used by an interpreter and so is untested.

4.4 Directions

Given the current limitations outlined in the previous section, we can now highlight a number of areas for future development.

4.4.1 Applicability: Autonomous and Autonomic Systems

As well as expanding our applications to more sophisticated multi-agent systems, it is important that we tackle a large variety of scenarios. Autonomous software systems (including autonomic systems, self-∗ systems, agent-based systems, etc.) are increasingly being programmed for use in critical areas. Typical examples include:

- *self-managing e-Health* — where autonomous software must be produced ensuring that distributed (health monitoring) sensors work together to provide (and retain, if unexpected events occur) coverage;
- *autonomous space exploration* — where reliable, fault-tolerant software must be produced for autonomous rover vehicles;
- *ubiquitous computing* — where software on multiple devices must communicate and cooperate to ensure goals are achieved;
- *self-organising logistics/resource management* — where distributed control of logistic/routing systems must be developed to ensure reliability and efficiency;
- *dynamically reconfigurable systems* — where components (hardware or software) can self-configure to achieve/retain goals of the system (e.g., `http://www.dyscas.org`);
- *autonomic systems* [285]; and
- protocol standards such as *FIPA*, and *agent protocol certification*.

Yet, while real systems are being deployed, the key problem is not *whether* such systems can be implemented, but whether they can be implemented in a *clear*, *reliable* and *verifiable* way. Thus, our techniques can be of great value in these areas. (In addition, see Section 4.4.7.)

4.4.2 Efficiency: Potential for use of MJI

The *Model Java Interface* (MJI) is a feature of JPF that effectively allows code blocks to be treated as atomic/native methods (for which new states are *not* built by JPF). In fact, such code runs on the Java virtual machine where JPF — which is programmed in Java — itself runs (the "native" Java virtual machine), rather than the custom virtual machine that JPF creates for running the Java code that is being model checked. From our perspective, the key point here is that we might well be able to use this to improve efficiency. If we can identify code within the AIL (or AJPF) that does not need to be checked, then we can use MJI to cut out parts of the state space. However, we need to be careful that no property that can be checked depends on the code "hidden" in this way.

We have made some preliminary investigations in migrating the unification code present in the AIL to MJI. This would cause the generation and search for unifiers

to occur in the native Java virtual machine rather than in the (slow) JPF virtual machine. Since unification is a potentially computationally expensive process, and much used in agent languages (but at a level that does not affect the properties of agent systems), it seems to be a good candidate for the use of MJI. Even though the native virtual machine is quicker, there is a computational overhead in transporting information between the two virtual machines which might cancel out any efficiency gained. We intend to investigate this further before reporting the results; we hope to experiment with case studies that represent high, medium, and low use of unification, and report this on future work.

4.4.3 Efficiency: Potential for use of Program Slicing/Abstraction

Since model checking techniques notoriously suffer from the *state-space explosion problem*, it is vital to be able to reduce the state space required in the process of model checking wherever possible. A key technique used in simplifying the analysis of conventional programs is that of *program slicing* [416, 442]. The basic idea behind program slicing is to eliminate details of the program that are not relevant to the analysis in hand. Thus, in our case, since we wish to verify some property, the idea is to eliminate parts of the program that do not affect that property; this is called *property-based slicing*, and effectively represents a precise form of under-approximation. In property-based slicing, instead of finding a slice for a particular variable (e.g., in a particular literal in a logic program), we slice the multi-agent program based on the specified property (the one to be later model checked). Slicing should be such that the result of subsequent model checking of the sliced system for that property will give identical results to the checking of that property on the original system. The aim is, or course, to make model checking faster for the sliced system by reducing its state space.

Although slicing techniques have been successfully used in conventional programs to reduce the state-space required, these standard techniques are only partially successful when applied to multi-agent programs. This is typically because they are not sensitive to the semantics of the modalities of the agent-specific property specification language and how they relate to specific constructs of agent-oriented programming languages. What we require are slicing techniques tailored to the *agent-specific* aspects of (multi-)agent programs. In [72, 74] we developed a novel, *property-based slicing* algorithm for the model checking of AgentSpeak programs. The algorithm was designed, proved correct, and applied to a case study, which showed good improvements in terms of both space and time. Having carried out this work on the simpler AgentSpeak verification system, we need to carry out research on extending and adapting this to the AIL/AJPF framework.

4.4.4 Generality: Target Languages

Although the AIL toolkit has been shown to be useful for several agent programming languages, we clearly need to broaden our applicability and attempt to use the toolkit for even more languages. In particular we would like to provide implementations of both a version of AgentSpeak and a language from the 3APL/2APL family [132] since these are — together with Jadex [343] which does not have formal semantics — the most widely used BDI languages available. We would also like to explore languages which provide native support for groups and organisations.

In order to support the further use of the toolkit, we also need to look into mechanisms of support for parsers or translators from the the syntax of agent languages to the appropriate Java code for creating an initial agent state of an AIL interpreter. It is possible to provide such systems using a Java-based parser and, indeed, we have already developed ones for converting the GWENDOLEN, GOAL and ORWELL syntax into the underlying AIL representation. We intend to investigate ways to generalise this system in order to make the provision of such parsers and translators as straightforward as possible.

4.4.5 Engineering: Agent Development Approach

A key observation is that while there are several software design methods for autonomous systems, their evolution has not occurred in a coherent way. Novel work has been done in, for example, autonomic design [189], programming languages for autonomous behaviour [76], and formal modelling and verification of multi-agent systems [73], but no full methodology has been devised that:

1. tackles the full life cycle, from design through to implementation, verification, and testing;
2. tackles true autonomy — i.e., the ability of a software entity to effectively decide upon its own goals, and decide how to work towards them;
3. tackles both the individual and collective autonomous activity, including complex organisational aspects such as cooperation, teamwork, self-management, and artifacts;
4. tackles implementation in high-level programming languages, such as agent-oriented languages; and
5. provides the ability to carry out formal verification, synthesis, and automated test case generation.

It would be also important to do all this within a framework that is:

a) semantically transparent, so that a formal description of every autonomous entity is available at every point in its life-cycle, and
b) graphical, providing user-friendly tools for visualising and developing autonomous software.

This is something that we are working towards and, clearly, the agent model-checking techniques described in this chapter play a central role in that endeavour. This work brings together: agent design methodologies (e.g., Gaia), executable agent specifications (e.g., MetateM), organisational and artifact models, graphical design tools, agent programming languages (e.g., *Jason*), and agent verification tools such as ours, in order to develop an integrated design environment.

4.4.6 Extension: Verification of Groups and Organisations

Rather than just seeing a multi-agent system as a simple (flat) set of agents communicating with each other, we can also view the agents as belonging to organisational constructs such as *groups*, *teams*, and *organisations*. Many approaches to the organisation of rational agents have been proposed, and have been shown to not only simplify the design and implementation of complex multi-agent systems, but also to improve the efficiency of those systems in practice.

Cohen and Levesque produced a significant paper on "Teamwork" [113], extending their previous work [110, 112, 282], which is often cited as the starting point for multi-agent organisations. They argued that a team of agents should *not* be modelled as an aggregate agent but propose new (logical) concepts of *joint intentions*, *joint commitments*, and *joint persistent goals* to ensure that teamwork does not break down due to any divergence of individual team members' beliefs or intentions. The authors' proposals oblige agents working in a team to retain team goals until it is mutually agreed amongst team members that a goal has now been achieved, is no longer relevant, or is impossible to achieve. This level of commitment is stronger than an agent's commitment to its individual goals which are dropped the moment it (individually) believes they are satisfied. Joint intentions can be reduced to individual intentions if supplemented with mutual beliefs.

Yet, there are many other approaches building on those fundamental ideas, as follows.

- Tidhar [414] introduced the concept of *team-oriented programming* with social structure;
- Ferber *et al.* [171, 172] present a case for an organisational-centred approach to the design and engineering of complex multi-agent systems;
- Pynadath *et al.* [351] describe their interpretation of "team-oriented programming" that aims to organise groups of heterogeneous agents to achieve team goals;

- Fisher *et al.* produced a series of papers [174, 177, 179, 230] that developed the MetateM language into a generalised approach for expressing dynamic, organised, distributed computations;
- Hübner *et al.*, in [241], proposed a three-component approach to the specification of agent organisations that combines independent structural and functional specifications with a deontic specification, the latter defining, among other things, which agent *roles* (structural) have *permission* to carry out group tasks (functional).

And there is still further work on groups [177, 179, 302], teams [178, 271, 274, 350, 411] and organisations [171, 183, 423, 445].

As all this suggests, it is important to be able to fit such organisational structures within our verification framework. Initial steps in this direction have been reported in [143, 144, 213].

4.4.7 Applicability: Verifying Human-Agent Teamwork

Whereas agent languages help with the complexity of developing autonomous systems — for example by allowing the modelling of agents at the right level of abstraction — very little work exists in using them for modelling realistic scenarios of human activity (or *human-robot teamwork*). While agent verification has the potential to be used in analysing agents in critical applications, there is an increasing need to consider the verification of situations involving more human-agent teamwork. For example, NASA's new focus on doing human-robotic exploration of the Moon and Mars brings human "agents" into the picture — this might simplify some of the autonomy requirements, but clearly increases safety and certification concerns. Autonomous space software is hard enough to verify already: the environments in which it executes are uncertain and the systems involved tend to be very complex. Adding a human element to space exploration scenarios will surely make this even more complex. The emphasis here is not on individual behaviour, but on *team* aspects, such as coordinating activities or cooperating to achieve a common goal.

On the one hand we have complex, human-agent teamwork, intended to be used in realistic environments. On the other, we have the need to formally verify teamwork, cooperation, and coordination aspects of these human-agent teams. Clearly these two aims are very far apart. Our current approach is to utilise the Brahms modelling language [402, 403], which has been used for many years to model human-robot activities. Thus, Brahms modellers have significant experience in describing and modelling this type of system. Brahms describes teamwork at quite a high level of abstraction and characterises (simple forms of) human behaviour as agent behaviour. However, Brahms has no formal semantics, which makes it more difficult to use model checking for verification, in particular in defining how the modalities

of the logic language used to write the system specification (i.e., the expected behaviour of the system) is to be interpreted in regards to states of a Brahms model. This is essentially the main problem we are tackling at present. Initial steps in this direction are described in [70, 176].

4.4.8 Efficiency/Extension: Alternative Model Checkers

The current approach is tightly linked to Java and JPF. However, in principle there is no requirement to use JPF. If the AIL is abstract enough then we should be able to provide alternative "back-ends" (i.e., target model checker) not only to potentially improve efficiency (e.g., by using NuSMV [98]), but also to check different types of properties (e.g., probabilistic model checking with Prism).

There are various different possible directions to achieve this. At the moment, model checking programs running in AIL interpreters integrates naturally with JPF since the AIL is implemented in Java — the input language for JPF. Providing a new back-end would involve either converting the existing AIL implementation into the input language of those model checkers or re-implementing the AIL in those input languages. It may be that some halfway option is available, much like the use of MJI in JPF. For instance, we might continue to use the existing AIL Java code for manipulating and deriving the unifiers for first order terms (tasks which are complex to implement but assumed to occur atomically in BDI languages) but re-implement much of the higher level AIL data structures in the new language.

4.5 Concluding Remarks

In this chapter, we first reviewed a number of essential concepts on autonomous agents (and rational agents in particular), (modal) logics for rational agents, logic-based (multi-)agent programming languages, and formal verification — (program) model checking in particular. We then gave an overview of our own approach for model checking multi-agent systems programmed in various agent programming languages, which uses a Java model checker as verification tool. The general approach to verification is that of program model checking: the input to the model checker, besides the property to be verified, is the same program that is used to run the system. The properties to be checked are given in a language based on modal logics for rational agents. An advantage of using one of the various existing model checkers is that such model checking tools have been developed for many years, which not only makes them reliable, they also encompass an enormous variety of techniques that take long to develop robustly and that make a huge difference in the performance of the model checker. In fact, we have been taking advantage of sophisticated existing model-checking tools since our previous work on model checking for AgentSpeak [71, 73].

We have also discussed a number of current shortcomings of our framework, then discussed various possible future directions of research that would extend and apply our framework towards various promising but challenging directions. Many visions for the future of Computer Science, such as pervasive and autonomic computing, depend on the ability to program dependable large-scale distributed systems where each component displays both autonomous and rational behaviour. Our work aims at contributing one particular approach to the engineering of such systems.

Chapter 5

Model Checking Logics of Strategic Ability: Complexity*

N. Bulling, J. Dix, and W. Jamroga

Abstract This chapter is about model checking and its complexity in some of the main temporal and strategic logics, e.g. **LTL**, **CTL**, and **ATL**. We discuss several variants of **ATL** (perfect vs. imperfect *recall*, perfect vs. imperfect *information*) as well as two different measures for model checking with concurrent game structures (explicit vs. implicit *representation* of transitions). Finally, we summarize some results about higher order representations of the underlying models.

N. Bulling, J. Dix
Dept. of Informatics, Niedersächsische Technische Hochschule, Germany e-mail: {bulling, dix}@in.tu-clausthal.de

W. Jamroga
Computer Science and Communications, University of Luxembourg, Luxembourg and Dept. of Informatics, Niedersächsische Technische Hochschule, Germany e-mail: wojtek.jamroga@uni.lu

* This work was partly funded by the NTH School for IT Ecosystems. NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover.

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_5,

5.1 Introduction

Model checking is a powerful method used in verification. Given a model and a formula in a certain logic, model checking determines whether the formula is true in the model. Usually, it is used to check specifications of desirable properties for a system whose model is given. If the formula is true, then we know the property expressed by the formula is satisfied in the model. If not, it might lead us to change the system or give hints how to debug it.

Model checking was invented and pioneered by the work of Edward Melson Clarke, Ernest Allen Emerson, and by Joseph Sifakis and Jean Pierre Queille in the 80ies as a means for formal verification of finite-state concurrent systems. Specifications about the system were expressed as temporal logic formulae. It was especially suited for checking hardware designs, but also applied to checking software specifications. While it started as a new approach replacing the then common Floyd-Hoare style logic, it could only handle relatively small (though non-trivial) examples. Scalability was an important motivation right from the beginning. The last years have seen many industrial applications, and a number of powerful model checkers are available today. As founders of a new and flourishing area in computer science, Clarke, Emerson and Sifakis have been honored with the Turing award in 2007.

Logic-based verification of multi-agent systems has become an important subfield on its own. Some important model-checkers are:

- Mocha [11], available for download at http://www.cis.upenn.edu/~mocha/,
- VeriCS [137], available at http://pegaz.ipipan.waw.pl/verics/,
- MCMAS [352, 353], available at http://www-lai.doc.ic.ac.uk/mcmas/.

In this chapter, we do not deal with practical aspects of MAS verification. Instead, we offer a comprehensive survey of theoretical results concerning the computational complexity of model checking for relevant properties of agents and their teams. To this end, we focus on the class of properties that can be specified in Alternating-time Temporal Logic **ATL** (a logic that extends the classical branching time logic **CTL** with strategic modalities) and some of its extensions.

The aim of this chapter is twofold: (1) to give a comprehensive overview of the complexity of model checking in various strategic logics based on **ATL**, and (2) to discuss how the complexity can change when the models are not given explicitly but implicitly. Often, a model cannot be represented explicitly: It is given in a certain symbolic manner. Thus, the representation can be much smaller than the model itself, but it has to be (at least partially) *unfolded* when checking its properties.

While there are several chapters in this book that investigate model checking in multi-agent systems (cf. Chapter 3, *Model Checking Agent Communication*, Chapter 4, *Directions for Agent Model Checking*, Chapter 8, *Model Checking Goal-Oriented Agent Programming*), in this chapter we investigate mainly logics of *strategic ability* (variants of **ATL**). We determine the precise complexity of several variants of the logics and show when the problems become (probably) undecidable.

The plan of this chapter is as follows. In Section 5.2 we introduce the logics we are interested in: the temporal logics **LTL**, **CTL**, and **CTL*** and the strategic logics **ATL** and **ATL*** as well as their variants based on the assumption that agents have (im)perfect recall and (im)perfect information. We define syntax and semantics and introduce several running examples. Section 5.3 is devoted to standard complexity results for the logics. By standard, we mean that the input size is given by the number of transitions in the model and the length of the formula. In particular, we assume that the model and the formula are given explicitly. In Section 5.4 we consider the case when the transitions in the model are given in a more compact way, rather than by enumerating outcomes of all the possible combinations of agents' actions. Then, it makes more sense to measure complexity with respect to the number of states and the number of agents in the model. Finally, in Section 5.5, we investigate model checking for symbolic, very compact representations of multiagent systems: *concurrent programs* and *modular interpreted systems*. This results in surprising complexity results, that can only be understood when looking closely at the size of the underlying structures (representations, models). We conclude in Section 5.6 with a discussion of our results, put them in perspective and point out future challenges.

5.2 The Logics: Syntax and Semantics

We begin by introducing temporal and strategic logics. We start with the linear-time logic **LTL** (Linear-time Temporal Logic) and the branching-time logics **CTL*** and **CTL** (Computation Tree Logic). Then, we present one of the most popular logics of strategic ability in multi-agent systems: **ATL** and **ATL*** (Alternating-time Temporal Logic). The relations between perfect vs. imperfect information on one hand, and perfect vs. imperfect recall on the other are discussed, and we show how they give rise to different semantics for **ATL** and **ATL***, yielding an interesting class of logics.

In the rest of this chapter we assume that Π is a non-empty set of *propositional symbols* and St a non-empty and finite set of *states*.

Remark 5.1 (Language, Semantics and Logic). In the following we proceed as follows. We introduce a logical language, say $\mathcal{L}$, which is defined as a set of formulae. Elements of $\mathcal{L}$ are called $\mathcal{L}$-formulae. Then, we consider (possibly several) semantics for the language. We look at each tuple consisting of a language and a suitable semantics (over a class of models) as a *logic*. The logic **CTL**, for instance, is given by the language $\mathcal{L}_{CTL}$ using the standard Kripke semantics.

5.2.1 Linear- and Branching-Time Logics

We begin by recalling two well-known classes of temporal logics: the *linear-time* logic **LTL** (Linear-Time Temporal Logic) and the *branching-time logics* **CTL** and **CTL*** (Computation Tree Logic).

5.2.1.1 The Languages $\mathcal{L}_{LTL}$, $\mathcal{L}_{CTL}$, and $\mathcal{L}_{CTL^*}$

$\mathcal{L}_{LTL}$ [341] extends the language of propositional logic with operators that allow to express temporal patterns over an infinite sequences of states, called *paths*. The basic temporal operators are $\mathcal{U}$ (*until*) and (*in the next state*).

Definition 5.1 (Language $\mathcal{L}_{LTL}$ [341]). The *language* $\mathcal{L}_{LTL}$ is given by all formulae generated by the following grammar, where $\mathsf{p} \in \Pi$ is a proposition: $\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathcal{U} \varphi \mid \varphi$.

The $\mathcal{L}_{LTL}$-formula $(\varphi \wedge \psi)$, for instance, expresses that φ and ψ hold in the next moment; $\varphi \mathcal{U} \psi$ states that the property φ is true at least until ψ becomes true which will eventually be the case. The additional operators $\Diamond$ (*sometime from now on*) and $\Box$ (*always from now on*) can be defined as macros by $\Diamond\varphi \equiv \top \mathcal{U} \varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$, respectively. The standard Boolean connectives $\top, \bot, \vee, \rightarrow$, and $\leftrightarrow$ are defined in their usual way.

The logic is called *linear-time* since formulae are interpreted over infinite *linear* orders of states. The logic **CTL*** [158] explicitly refers to patterns of properties that can occur along a particular temporal path, *as well as* to the set of possible time series, and thus extends **LTL**. The latter dimension is handled by so called *path quantifiers*: E (*there is a path*) and A (*for all paths*) where the A quantifier is defined as macro: $\mathsf{A}\varphi \equiv \neg\mathsf{E}\neg\varphi$. Hence, the language of **CTL***, $\mathcal{L}_{CTL^*}$, extends $\mathcal{L}_{LTL}$ by adding the existential path quantifier E.

Definition 5.2 (Language $\mathcal{L}_{CTL^*}$ [158]). The *language* $\mathcal{L}_{CTL^*}$ is given by all formulae generated by the following grammar: $\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{E}\gamma$ where $\gamma ::= \varphi \mid \neg\gamma \mid \gamma \wedge \gamma \mid \gamma \mathcal{U} \gamma \mid \gamma$ and $\mathsf{p} \in \Pi$. Formulae φ (resp. γ) are called *state* (resp. *path*) formulae.

Additionally, the same abbreviations as for $\mathcal{L}_{LTL}$ are defined. The $\mathcal{L}_{CTL^*}$-formula $\mathsf{E}\Diamond\varphi$, for instance, ensures that there is at least one path on which φ holds at some (future) time moment. Thus, $\mathcal{L}_{CTL^*}$-formulae do not only talk about temporal patterns on a given path but also quantify (existentially or universally) over such paths.

Finally, we define a fragment of **CTL*** called **CTL** [99] which is strictly *less expressive* but has *better computational properties*. The language $\mathcal{L}_{CTL}$ restricts $\mathcal{L}_{CTL^*}$ in such a way that each temporal operator must be directly preceded by a path quantifier. For example, $\mathsf{A}\Box\mathsf{E}\ \ \mathsf{p}$ is a $\mathcal{L}_{CTL}$-formula whereas $\mathsf{A}\Box\Diamond\mathsf{p}$ is not. Although

this completely characterizes the language we also provide the original definition in which modalities are given by path quantifiers *coupled* with temporal operators. Note that, chronologically, **CTL** was proposed and studied before **CTL***.

Definition 5.3 (Language $\mathcal{L}_{CTL}$ [99]). The *language* $\mathcal{L}_{CTL}$ is given by all formulae generated by the following grammar, where $\mathsf{p} \in \Pi$ is a proposition: $\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{E}(\varphi \mathcal{U} \varphi) \mid \mathsf{E} \quad \varphi \mid \mathsf{E}\Box\varphi$.

Again, the Boolean connectives are given by their usual abbreviations. In addition to that, we define the following: $\Diamond\varphi \equiv \top \mathcal{U} \varphi$, $\mathsf{A} \quad \varphi \equiv \neg\mathsf{E} \quad \neg\varphi$, $\mathsf{A}\Box\varphi \equiv \neg\mathsf{E}\Diamond\neg\varphi$, and $\mathsf{A}\varphi \mathcal{U} \psi \equiv \neg\mathsf{E}((\neg\psi)\mathcal{U}(\neg\varphi \wedge \neg\psi)) \wedge \neg\mathsf{E}\Box\neg\psi$. We note that in the definition of the language the existential quantifier cannot be replaced by the universal one without losing expressiveness (cf. [277]).

5.2.1.2 Semantics for $\mathcal{L}_{LTL}$, $\mathcal{L}_{CTL^*}$, and $\mathcal{L}_{CTL}$

As mentioned above, the semantics of **LTL** is given over *paths* that are infinite sequences of states from St and a labeling function $\pi : \Pi \to \mathcal{P}(St)$ that determines which propositions are true at which states. Note that each path can be considered as a mapping $\mathbb{N} \to St$. We use $\lambda[i]$ to denote the ith position on path λ (starting from $i = 0$) and $\lambda[i,\infty]$ to denote the subpath of λ starting from i (i.e. $\lambda[i,\infty] = \lambda[i]\lambda[i+1]\ldots$).

Definition 5.4 (Semantics $\models^{\mathbf{LTL}}$). Let λ be a path and π be a valuation over St. The semantics of $\mathcal{L}_{LTL}$-formulae is defined by the satisfaction relation $\models^{\mathbf{LTL}}$ defined as follows:

$\lambda, \pi \models^{\mathbf{LTL}} \mathsf{p}$ iff $\lambda[0] \in \pi(\mathsf{p})$ and $\mathsf{p} \in \Pi$;
$\lambda, \pi \models^{\mathbf{LTL}} \neg\varphi$ iff not $\lambda, \pi \models^{\mathbf{LTL}} \varphi$ (we will also write $\lambda, \pi \not\models \mathbf{LTL}\varphi$);
$\lambda, \pi \models^{\mathbf{LTL}} \varphi \wedge \psi$ iff $\lambda, \pi \models^{\mathbf{LTL}} \varphi$ and $\lambda, \pi \models^{\mathbf{LTL}} \psi$;
$\lambda, \pi \models^{\mathbf{LTL}} \quad \varphi$ iff $\lambda[1,\infty], \pi \models^{\mathbf{LTL}} \varphi$; and
$\lambda, \pi \models^{\mathbf{LTL}} \varphi \mathcal{U} \psi$ iff there is an $i \in \mathbb{N}_0$ such that $\lambda[i,\infty], \pi \models \psi$ and $\lambda[j,\infty], \pi \models^{\mathbf{LTL}} \varphi$ for all $0 \leq j < i$;

Thus, according to Remark 5.1, the *logic* **LTL** is given by $(\mathcal{L}_{LTL}, \models^{\mathbf{LTL}})$. Paths are considered as (canonical) models for $\mathcal{L}_{LTL}$-formulae.

For model checking we require a finite representation of the input λ. To this end, we use a (pointed) Kripke model $\mathfrak{M}, q$ and consider the problem whether an $\mathcal{L}_{LTL}$-formula holds on *all* paths of $\mathfrak{M}$ starting in q.

A *Kripke model* (or *unlabeled transition system*) is given by $\mathfrak{M} = \langle St, \mathcal{R}, \Pi, \pi \rangle$ where St is a nonempty set of states (or possible worlds), $\mathcal{R} \subseteq St \times St$ is a *serial* transition relation on states, Π is a set of atomic propositions, and $\pi : \Pi \to \mathcal{P}(St)$ is a valuation of propositions. A *path* λ (or *computation*) in $\mathfrak{M}$ is an infinite sequence

of states that can result from subsequent transitions, and refers to a possible course of action. We use the same notation for these paths as introduced above. For $q \in St$ we use $\Lambda_{\mathfrak{M}}(q)$ to denote the set of all paths of $\mathfrak{M}$ starting in q and we define $\Lambda_{\mathfrak{M}}$ as $\bigcup_{q \in St} \Lambda_{\mathfrak{M}}(q)$. The subscript "$\mathfrak{M}$" is often omitted when clear from context.

$\mathcal{L}_{CTL^*}$- and $\mathcal{L}_{CTL}$-formulae are interpreted over Kripke models but in addition to $\mathcal{L}_{LTL}$-(path) formulae (which can only occur as subformulae) it must be specified how state formulae are evaluated.

Definition 5.5 (Semantics $\models^{\mathbf{CTL}^*}$). Let $\mathfrak{M}$ be a Kripke model, $q \in St$ and $\lambda \in \Lambda$. The semantics of $\mathcal{L}_{CTL^*}$- and $\mathcal{L}_{CTL}$-formulae are given by the satisfaction relation $\models^{\mathbf{CTL}^*}$ for state formulae by

$\mathfrak{M}, q \models^{\mathbf{CTL}^*} \mathsf{p}$ iff $\lambda[0] \in \pi(\mathsf{p})$ and $\mathsf{p} \in \Pi$;
$\mathfrak{M}, q \models^{\mathbf{CTL}^*} \neg\varphi$ iff $\mathfrak{M}, q \not\models \mathbf{CTL}^*\varphi$;
$\mathfrak{M}, q \models^{\mathbf{CTL}^*} \varphi \wedge \psi$ iff $\mathfrak{M}, q \models^{\mathbf{CTL}^*} \varphi$ and $\mathfrak{M}, q \models^{\mathbf{CTL}^*} \psi$;
$\mathfrak{M}, q \models^{\mathbf{CTL}^*} \mathsf{E}\varphi$ iff there is a path $\lambda \in \Lambda(q)$ such that $\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \varphi$;

and for path formulae by:

$\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \varphi$ iff $\mathfrak{M}, \lambda[0] \models^{\mathbf{CTL}^*} \varphi$;
$\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \neg\gamma$ iff $\mathfrak{M}, \lambda \not\models \mathbf{CTL}^*\gamma$;
$\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \gamma \wedge \delta$ iff $\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \gamma$ and $\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \delta$;
$\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \quad \gamma$ iff $\lambda[1, \infty], \pi \models^{\mathbf{CTL}^*} \gamma$; and
$\mathfrak{M}, \lambda \models^{\mathbf{CTL}^*} \gamma \mathcal{U} \delta$ iff there is an $i \in \mathbb{N}_0$ such that $\mathfrak{M}, \lambda[i, \infty] \models^{\mathbf{CTL}^*} \delta$ and $\mathfrak{M}, \lambda[j, \infty] \models^{\mathbf{CTL}^*} \gamma$ for all $0 \leq j < i$.

Alternatively, an equivalent *state-based* semantics for **CTL** can be given:

$\mathfrak{M}, q \models^{\mathbf{CTL}} p$ iff $q \in \pi(p)$;
$\mathfrak{M}, q \models^{\mathbf{CTL}} \neg\varphi$ iff $\mathfrak{M}, q \not\models \mathbf{CTL}\varphi$;
$\mathfrak{M}, q \models^{\mathbf{CTL}} \varphi \wedge \psi$ iff $\mathfrak{M}, q \models^{\mathbf{CTL}} \varphi$ and $\mathfrak{M}, q \models^{\mathbf{CTL}} \psi$;
$\mathfrak{M}, q \models^{\mathbf{CTL}} \mathsf{E} \quad \varphi$ iff there is a path $\lambda \in \Lambda(q)$ such that $\mathfrak{M}, \lambda[1] \models^{\mathbf{CTL}} \varphi$;
$\mathfrak{M}, q \models^{\mathbf{CTL}} \mathsf{E}\Box\varphi$ iff there is a path $\lambda \in \Lambda(q)$ such that $\mathfrak{M}, \lambda[i] \models^{\mathbf{CTL}} \varphi$ for every $i \geq 0$;
$\mathfrak{M}, q \models^{\mathbf{CTL}} \mathsf{E}\varphi\mathcal{U}\psi$ iff there is a path $\lambda \in \Lambda(q)$ such that $\mathfrak{M}, \lambda[i] \models^{\mathbf{CTL}} \psi$ for some $i \geq 0$, and $\mathfrak{M}, \lambda[j, \infty] \models^{\mathbf{CTL}} \varphi$ for all $0 \leq j < i$.

This equivalent semantics underlies the model checking algorithm for **CTL** which can be implemented in *P* rather than *PSPACE* which is the case for **CTL*** (cf. Section 5.3.1). Hence, the logics **CTL** and **CTL*** are given by $(\mathcal{L}_{CTL}, \models^{\mathbf{CTL}})$ and $(\mathcal{L}_{CTL^*}, \models^{\mathbf{CTL}^*})$, respectively.

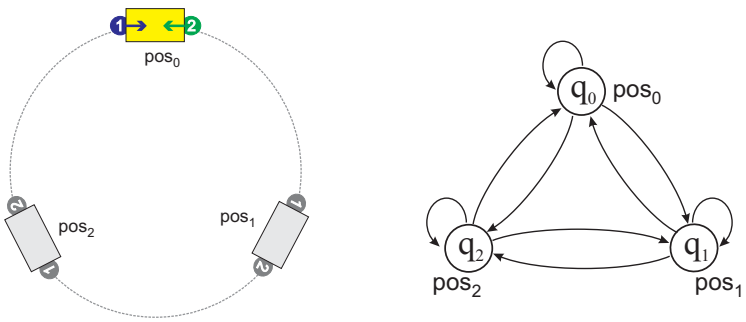

Fig. 5.1 Two robots and a carriage: a schematic view (left) and a transition system $\mathfrak{M}_0$ that models the scenario (right).

Remark 5.2. Note that model checking problem for an $\mathcal{L}_{LTL}$-formula φ with respect to a given Kripke model $\mathfrak{M}$ and a state q is equivalent to the **CTL*** model checking problem $\mathfrak{M}, q \models^{\mathbf{CTL}^*} \mathsf{A}\varphi$.

We end this section with an example.

Example 5.1 (Robots and Carriage). Consider the scenario depicted in Figure 5.1. Two robots push a carriage from opposite sides. As a result, the carriage can move clockwise or anticlockwise, or it can remain in the same place – depending on who pushes with more force (and, perhaps, who refrains from pushing). To make our model of the domain discrete, we identify 3 different positions of the carriage, and associate them with states q_0, q_1, and q_2. The arrows in transition system M_0 indicate how the state of the system can change in a single step. We label the states with propositions $\mathsf{pos}_0, \mathsf{pos}_1, \mathsf{pos}_2$, respectively, to allow for referring to the current position of the carriage in the object language.

For example, we have $\mathfrak{M}_0, q_0 \models^{\mathbf{CTL}} \mathsf{E}\Diamond\mathsf{pos}_1$: In state q_0, there is a path such that the carriage will reach position 1 sometime in the future. Of course, the same is not true for *all* paths, so we also have that $\mathfrak{M}_0, q_0 \models^{\mathbf{CTL}} \neg\mathsf{A}\Diamond\mathsf{pos}_1$.

5.2.2 Strategic Abilities under Perfect Information

In this section we introduce logics that can be used to model and to reason about strategic abilities of agents with perfect information. Here "perfect information" is understood in such a way that agents know the current state of the system: The agents are able to distinguish all states of the system. This is fundamentally different from the imperfect information setting presented in Section 5.2.3 where *different*

states possibly provide the same information to an agent and thus make them appear indistinguishable to it. This must be reflected in the agents' available strategies.

¿From now on, we assume that $\mathbb{A}\mathrm{gt} = \{1, \ldots, k\}$ is a non-empty and finite set of *agents*. Sometimes, in order to make the examples easier to read, we may also use symbolic names $(a, b, c, \ldots)$ when referring to agents.

5.2.2.1 The Languages $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$

The logics **ATL*** and **ATL** [13, 14] (Alternating-time Temporal Logic) are generalizations of **CTL*** and **CTL**, respectively. In $\mathcal{L}_{ATL^*}/\mathcal{L}_{ATL}$ the path quantifiers E, A are replaced by *cooperation modalities* $\langle\langle A\rangle\rangle$ where $A \subseteq \mathbb{A}\mathrm{gt}$ is a team of agents. Formula $\langle\langle A\rangle\rangle\gamma$ expresses that team A has a *collective strategy* to enforce γ. The recursive definition of the language is given below.

Definition 5.6 (Language $\mathcal{L}_{ATL^*}$ [13]). The *language* $\mathcal{L}_{ATL^*}$ is given by all formulae generated by the following grammar: $\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle A\rangle\rangle\gamma$ where $\gamma ::= \varphi \mid \neg\gamma \mid \gamma \wedge \gamma \mid \gamma\,\mathcal{U}\,\gamma \mid \quad \gamma$, $A \subseteq \mathbb{A}\mathrm{gt}$, and $\mathsf{p} \in \Pi$. Formulae φ (resp. γ) are called *state* (resp. *path*) formulae.

We use similar abbreviations to the ones introduced in Section 5.2.1.1. In the case of a single agent a we will also write $\langle\langle a\rangle\rangle$ instead $\langle\langle\{a\}\rangle\rangle$. An example $\mathcal{L}_{ATL^*}$-formula is $\langle\langle A\rangle\rangle\Box\Diamond\mathsf{p}$ which says that coalition A can guarantee that p is satisfied infinitely many times (ever and ever again in the future).

The language $\mathcal{L}_{ATL}$ restricts $\mathcal{L}_{ATL^*}$ in the same way as $\mathcal{L}_{CTL}$ restricts $\mathcal{L}_{CTL^*}$: Each temporal operator must be directly preceded by a cooperation modality.

Definition 5.7 (Language $\mathcal{L}_{ATL}$ [13]). The *language* $\mathcal{L}_{ATL}$ is given by all formulae generated by the following grammar: $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle A\rangle\rangle \quad \varphi \mid \langle\langle A\rangle\rangle\Box\varphi \mid \langle\langle A\rangle\rangle\varphi\,\mathcal{U}\,\varphi$ where $A \subseteq \mathbb{A}\mathrm{gt}$ and $\mathsf{p} \in \Pi$.

The $\mathcal{L}_{ATL^*}$-formula $\langle\langle A\rangle\rangle\Box\Diamond\mathsf{p}$ is obviously not a formula of $\mathcal{L}_{ATL}$ as it includes two consecutive temporal operators. In more general terms, $\mathcal{L}_{ATL}$ does not allow to express abilities related to, e.g., fairness properties. Still, many interesting properties are expressible. For instance, we can state that agent a has a strategy that permanently take away the ability to enforce p from coalition B: $\langle\langle a\rangle\rangle\Box\neg\langle\langle B\rangle\rangle \quad \mathsf{p}$. As for the two computation tree logics, the choice between $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$ reflects the tradeoff between expressiveness and practicality.

5.2.2.2 Perfect Information Semantics for $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$

The semantics for $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$ are defined over a variant of transition systems where transitions are labeled with combinations of actions, one per agent. Formally,

a *concurrent game structure* (CGS) is a tuple $\mathfrak{M} = \langle \mathbb{A}\mathrm{gt}, St, \Pi, \pi, Act, d, o\rangle$ which includes a nonempty finite set of all agents $\mathbb{A}\mathrm{gt} = \{1,\dots,k\}$, a nonempty set of states St, a set of atomic propositions Π and their valuation $\pi : \Pi \to \mathcal{P}(St)$, and a nonempty finite set of (atomic) actions Act. Function $d : \mathbb{A}\mathrm{gt} \times St \to \mathcal{P}(Act)$ defines nonempty sets of actions available to agents at each state, and o is a (deterministic) transition function that assigns the outcome state $q' = o(q, \alpha_1, \dots, \alpha_k)$ to state q and a tuple of actions $\langle \alpha_1, \dots, \alpha_k\rangle$ for $\alpha_i \in d(i,q)$ and $1 \le i \le k$, that can be executed by $\mathbb{A}\mathrm{gt}$ in q. We also write $d_a(q)$ instead of $d(a,q)$. So, it is assumed that all the agents execute their actions synchronously: The combination of the actions, together with the current state, determines the next transition of the system.

A *strategy* of agent a is a conditional plan that specifies what a is going to do in each situation. It makes sense, from a conceptual and computational point of view, to distinguish between two types of "situations" (and hence strategies): An agent might base his decision only on the current state or on the whole history of events that have happened. A *history* is considered as a finite sequence of states of the system.

A *perfect information perfect recall strategy* for agent a (*IR-strategy* for short)[2] is a function $s_a : St^+ \to Act$ such that $s_a(q_0 q_1 \dots q_n) \in d_a(q_n)$. The set of such strategies is denoted by Σ_a^{IR}. On the other hand, a *perfect information memoryless strategy* for agent a (*Ir-strategy* for short) is given by a function $s_a : St \to Act$ where $s_a(q) \in d_a(q)$. The set of such strategies is denoted by Σ_a^{Ir}. We will use the term *strategy* to refer to any of these two types.

A *collective strategy* for a group of agents $A = \{a_1, \dots, a_r\} \subseteq \mathbb{A}\mathrm{gt}$ is simply a tuple $s_A = \langle s_{a_1}, \dots, s_{a_r}\rangle$ of strategies, one per agent from A. By $s_A|_a$, we denote agent a's part s_a of the collective strategy s_A where $a \in A$. The set of A's collective perfect information strategies is given by $\Sigma_A^{IR} = \prod_{a\in A} \Sigma_a^{IR}$ (in the perfect recall case) and $\Sigma_A^{Ir} = \prod_{a\in A} \Sigma_a^{Ir}$ (in the memoryless case). The set of all *strategy profiles* is given by $\Sigma^{IR} = \Sigma^{IR}_{\mathbb{A}\mathrm{gt}}$ (resp. $\Sigma^{Ir} = \Sigma^{Ir}_{\mathbb{A}\mathrm{gt}}$).

Function $out(q, s_A)$ returns the set of all paths that may occur when agents A execute strategy s_A from state q onward. For an *IR*-strategy the set is given as follows:

$out(q, s_A) = \{\lambda = q_0 q_1 q_2 \dots \mid q_0 = q$ and for each $i = 1, 2, \dots$ there exists a tuple of agents' decisions $\langle \alpha^{i-1}_{a_1}, \dots, \alpha^{i-1}_{a_k}\rangle$ such that $\alpha^{i-1}_a \in d_a(q_{i-1})$ for every $a \in \mathbb{A}\mathrm{gt}$, and $\alpha^{i-1}_a = s_A|_a(q_0 q_1 \dots q_{i-1})$ for every $a \in A$, and $o(q_{i-1}, \alpha^{i-1}_{a_1}, \dots, \alpha^{i-1}_{a_k}) = q_i\}$.

For an *Ir*-strategy s_A the outcome is defined analogously: "$s_A|_a(q_0 q_1 \dots q_{i-1})$" is simply replaced by "$s_A|_a(q_{i-1})$"

The semantics for $\mathcal{L}_{ATL}$ and $\mathcal{L}_{ATL^*}$, one for each type of strategy, are shown below. Informally speaking, $\mathfrak{M}, q \models \langle\langle A\rangle\rangle\gamma$ if, and only if, there exists a collective strategy s_A such that γ holds for all computations from $out(q, s_A)$.

[2] The notation was introduced in [388] where i (resp. I) stands for *imperfect* (resp. *perfect*) *information* and r (resp. R) for *imperfect* (resp. *perfect*) *recall*. Also compare with Section 5.2.3.

Definition 5.8 (Perfect Information Semantics $\models_{IR}$ and $\models_{Ir}$). Let $\mathfrak{M}$ be a CGS. The *perfect information perfect recall semantics* for $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$, *IR-semantics* for short, is defined as $\models^{\mathbf{CTL}^*}$ from Definition 5.5, denoted by $\models_{IR}$, but the rule for $\mathsf{E}\varphi$ is replaced by the following clause:

$\mathfrak{M}, q \models_{IR} \langle\langle A\rangle\rangle\gamma$ iff there is an *IR*-strategy $s_A \in \Sigma_A^{IR}$ for A such that for every path $\lambda \in out(s_A, q)$, we have $\mathfrak{M}, \lambda \models_{IR} \gamma$.

The *perfect information memoryless semantics* for $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$, *Ir-semantics* for short, is given as above but "*IR*" is replaced by "*Ir*" everywhere.

Remark 5.3. Note that cooperation modalities are neither "diamonds" nor "boxes" in terms of classical modal logic. Rather, they are combinations of both as their structure can be described by "$\exists\forall$": we ask for the *existence* of a strategy of the proponents which is successful against *all* responses of the opponents.

In [89] it is shown how the cooperation modalities can be decomposed into two parts in the context of **STIT** logic. A similar decomposition is considered in [253] for the analysis of stochastic multi-agent systems.

The $\mathcal{L}_{CTL^*}$ path quantifiers A and E can be embedded in $\mathcal{L}_{ATL^*}$ using the *IR*-semantics in the following way: $\mathsf{A}\gamma \equiv \langle\langle\varnothing\rangle\rangle\gamma$ and $\mathsf{E}\gamma \equiv \langle\langle\mathbb{A}\mathrm{gt}\rangle\rangle\gamma$.

Analogously to **CTL**, it is possible to provide a state-based semantics for $\mathcal{L}_{ATL}$. We only present the clause for $\langle\langle A\rangle\rangle\Box\varphi$ (the cases for the other temporal operators are given in a similar way):

$\mathfrak{M}, q, \models_{Ix}^{\mathbf{ATL}} \langle\langle A\rangle\rangle\Box\varphi$ iff there is an *Ix*-strategy $s_A \in \Sigma_A^{Ix}$ such that for all $\lambda \in out(q, s_A)$ and $i \in \mathbb{N}_0$ it holds that $\mathfrak{M}, q, \models_{Ix}^{\mathbf{ATL}} \varphi$

where x is either R or r.

This already suggests that dealing with $\mathcal{L}_{ATL}$ is computationally less expensive than with $\mathcal{L}_{ATL^*}$. On the other hand, $\mathcal{L}_{ATL}$ lacks expressiveness: There is no formula which is true for the memoryless semantics and false for the perfect recall semantics, and vice versa.

Theorem 5.1. [3] *For $\mathcal{L}_{ATL}$, the perfect perfect recall semantics is equivalent to the memoryless semantics under perfect information, i.e., $\mathfrak{M}, q \models_{IR} \varphi$ iff $\mathfrak{M}, q \models_{Ir} \varphi$. Both semantics are different for $\mathcal{L}_{ATL^*}$.*

Thus, when referring to $\mathcal{L}_{ATL}$ using the perfect information semantics, we can omit the subscript in the satisfaction relation $\models$.

Definition 5.9 ($\mathbf{ATL}_{Ix}$, $\mathbf{ATL}^*_{Ix}$, ATL, $\mathbf{ATL}^*$). We define $\mathbf{ATL}_{Ix}$ and $\mathbf{ATL}^*_{Ix}$ as the logics $(\mathcal{L}_{ATL}, \models_{Ix})$ and $(\mathcal{L}_{ATL^*}, \models_{Ix})$ where $x \in \{r, R\}$, respectively. Moreover, we use **ATL** (resp. $\mathbf{ATL}^*$) as an abbreviation for $\mathbf{ATL}_{IR}$ (resp. $\mathbf{ATL}^*_{IR}$).

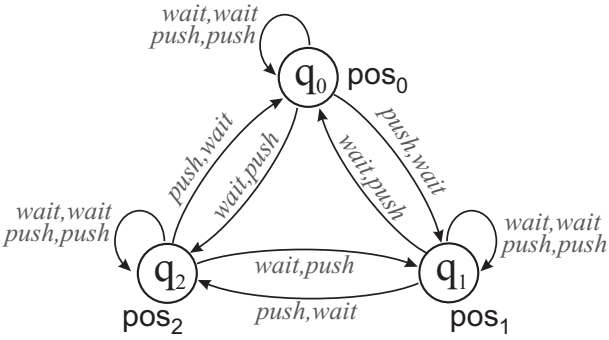

Fig. 5.2 The robots and the carriage: a concurrent game structure $\mathfrak{M}_1$.

Note again, that $\mathbf{ATL}_{IR}$ and $\mathbf{ATL}_{Ir}$ are equivalent logics. We end our presentation of the language and semantics with an example.

Example 5.2 (Robots and Carriage, ctd.). Transition system $\mathfrak{M}_0$ from Figure 5.1 enabled us to study the evolution of the system as a whole. However, it did not allow us to represent *who* can achieve *what*, and how the possible actions of the agents interact. Concurrent game structure $\mathfrak{M}_1$, presented in Figure 5.2, fills the gap. We assume that each robot can either push (action *push*) or refrain from pushing (action *wait*). Moreover, they both use the same force when pushing. Thus, if the robots push simultaneously or wait simultaneously, the carriage does not move. When only one of the robots is pushing, the carriage moves accordingly.

As the outcome of each robot's action depends on the current action of the other robot, no agent can make sure that the carriage moves to any particular position. So, we have for example that $\mathfrak{M}_1, q_0 \models \neg\langle\langle 1\rangle\rangle\Diamond \mathsf{pos}_1$. On the other hand, the agent can at least make sure that the carriage will *avoid* particular positions. For instance, it holds that $\mathfrak{M}_1, q_0 \models \langle\langle 1\rangle\rangle\Box\neg \mathsf{pos}_1$, the right strategy being $s_1(q_0) = wait, s_1(q_2) = push$ (the action that we specify for q_1 is irrelevant).

5.2.3 Strategic Abilities under Imperfect Information

ATL* and **ATL** include no way of addressing uncertainty that an agent or a process may have about the current situation. Several extensions capable of dealing with imperfect information have been proposed, e.g., in [14, 255, 388].

Here, we take Schobbens' version from [388] as the "core", minimal $\mathcal{L}_{ATL^*}$-based language for strategic ability under imperfect information. We take the already defined languages $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$ but here the cooperation modalities have an addi-

[3] The property has been first observed in [388] but it follows from [14] in a straightforward way.

tional *epistemic flavor* by means of a modified semantics as we will show below.[4] The models, *imperfect information concurrent game structures* (ICGS), can be seen as concurrent game structures augmented with a family of indistinguishability relations $\sim_a \subseteq St \times St$, one per agent $a \in \mathbb{A}\mathrm{gt}$. The relations describe agents' uncertainty: $q \sim_a q'$ means that agent a cannot distinguish between states q and q' of the system. Each $\sim_a$ is assumed to be an equivalence relation. It is also required that agents have the same choices in indistinguishable states: if $q \sim_a q'$ then $d(a,q) = d(a,q')$. Two *histories* $h = q_0 q_1 \dots q_n$ and $h' = q'_0 q'_1 \dots q'_{n'}$ are said to be *indistinguishable* for agent a, $h \sim_a h'$, if and only if, $n = n'$ and $q_i \sim_a q'_i$ for $i = 1, \dots, n$. This means that we deal with the synchronous notion of recall according to the classification in [169].

An *imperfect information strategy*[5] – memoryless or perfect recall – of agent a is a plan that takes into account a's epistemic limitations. An executable strategy must prescribe the *same choices for indistinguishable situations*. Therefore, we restrict the strategies that can be used by agents in the following way.

An *imperfect information perfect recall strategy* (*iR*-strategy for short) of agent a is an *IR*-strategy satisfying the following additional constraint: For all histories $h, h' \in St^+$, if $h \sim_a h'$ then $s_a(h) = s_a(h')$. That is, an *iR*-strategy is required to assign the same action to indistinguishable histories. Note that, as before, a perfect recall strategy (memoryless or not) assigns an action to each element from St^+.

An *imperfect information memoryless strategy* (*ir*-strategy for short) is an *Ir*-strategy satisfying the following constraint: if $q \sim_a q'$ then $s_a(q) = s_a(q')$. The set of a's *ir* (resp. *iR*) strategies is denoted by Σ_a^{ir} (resp. Σ_a^{iR}).

A *collective iR/ir-strategy* is a combination of individual *iR*/*ir*-strategies. The set of A's collective imperfect information strategies is given by $\Sigma_A^{iR} = \prod_{a \in A} \Sigma_a^{iR}$ (in the perfect recall case) and $\Sigma_A^{ir} = \prod_{a \in A} \Sigma_a^{ir}$ (in the memoryless case). The set of all strategy profiles is given by $\Sigma^{iR} = \Sigma_{\mathbb{A}\mathrm{gt}}^{iR}$ (resp. $\Sigma^{ir} = \Sigma_{\mathbb{A}\mathrm{gt}}^{ir}$). The outcome function $out(q, s_A)$ for the imperfect information cases is defined as before.

Definition 5.10 (Imperfect Information Semantics $\models_{iR}$ and $\models_{ir}$). Let $\mathfrak{M}$ be an ICGS, and let $\mathrm{img}(q,\rho) = \{q' \mid \rho(q,q')\}$ be the image of state q wrt a binary relation ρ. The *imperfect information perfect recall* semantics (*iR-semantics*) for $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$, denoted by $\models_{iR}$, is given as in Definition 5.8 with the rule for $\langle\langle A \rangle\rangle \gamma$ replaced by the following clause:

> $\mathfrak{M}, q \models_{iR} \langle\langle A \rangle\rangle \gamma$ iff there is an *iR*-strategy $s_A \in \Sigma_A^{iR}$ such that, for each $q' \in \mathrm{img}(q, \sim_A)$ and every $\lambda \in out(s_A, q')$, we have $\mathfrak{M}, \lambda \models_{iR} \gamma$ (where $\sim_A := \bigcup_{a \in A} \sim_a$).

The *imperfect information memoryless semantics* for $\mathcal{L}_{ATL^*}$ and $\mathcal{L}_{ATL}$, *ir-semantics for short*, is given as above but "*iR*" is replaced by "*ir*" everywhere.

---

[4] In [388] the cooperation modalities are presented with a subscript: $\langle\langle A \rangle\rangle_{ir}$ to indicate that they address agents with imperfect **i**nformation and imperfect **r**ecall. Here, we take on a rigorous semantic point of view and keep the syntax unchanged.

[5] Also called *uniform* strategy.

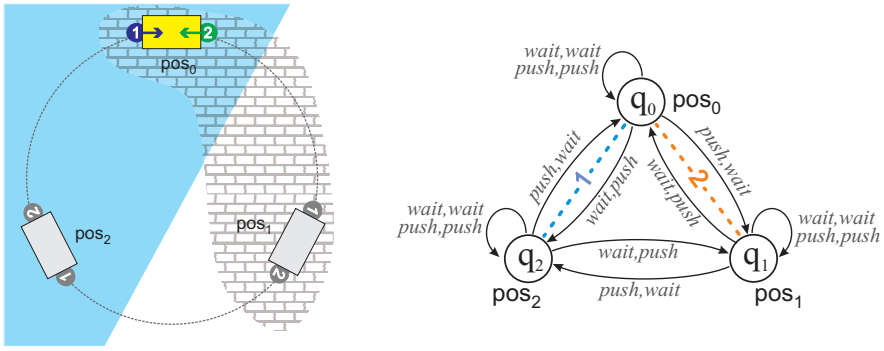

Fig. 5.3 Two robots and a carriage: a schematic view (left) and an imperfect information concurrent game structure $\mathfrak{M}_2$ that models the scenario (right).

Note that $\mathfrak{M}, q \models_{ix} \langle\langle A \rangle\rangle \gamma$ requires A to have a single strategy that is successful in *all* states indistinguishable from q.

Remark 5.4 (Implicit knowledge operators). Note that some knowledge operators are *implicitly* given by the cooperation modalities if the imperfect information semantics is used. In this setting a formula $\langle\langle A \rangle\rangle \gamma$ is read as follows: every agent in A *knows* that they (the agents in A) have a collective strategy to enforce γ. In particular, one can express $K_a\varphi$ ("a knows that φ") by $\langle\langle a \rangle\rangle \varphi \mathcal{U} \varphi$, and $E_A\varphi$ ("everybody in A knows that φ") by $\langle\langle A \rangle\rangle \varphi \mathcal{U} \varphi$. More sophisticated epistemic versions of **ATL** which contain explicit knowledge operators (including ones for common and distributed knowledge) are, for instance, considered in [201, 233, 255, 328].

Definition 5.11 ($\mathbf{ATL_{ix}}$, $\mathbf{ATL^*_{ix}}$).
We define $\mathbf{ATL_{ix}}$ and $\mathbf{ATL^*_{ix}}$ as the logics $(\mathcal{L}_{ATL}, \models_{ix})$ and $(\mathcal{L}_{ATL^*}, \models_{ix})$ where $x \in \{r, R\}$, respectively.

Example 5.3 (Robots and Carriage, ctd.). We refine the scenario from Examples 5.1 and 5.2 by restricting perception of the robots. Namely, we assume that robot 1 is only able to observe the color of the surface on which it is standing, and robot 2 perceives only the texture (cf. Figure 5.3). As a consequence, the first robot can distinguish between position 0 and position 1, but positions 0 and 2 look the same to it. Likewise, the second robot can distinguish between positions 0 and 2, but not 0 and 1. We also assume that the agents are memoryless, i.e., they cannot memorize their previous observations.

With their observational capabilities restricted in such way, no agent can make the carriage reach or avoid any selected states singlehandedly. E.g., we have that $\mathfrak{M}_2, q_0 \models_{ir} \neg\langle\langle 1 \rangle\rangle \Box \neg \mathsf{pos}_1$. Note in particular that strategy s_1 from Example 5.2 cannot be used here because it is not uniform (indeed, the strategy tells robot 1 to wait

in q_0 and push in q_2 but both states look the same to the robot). The robots cannot even be sure to achieve the task together: $\mathfrak{M}_2, q_0 \models_{ir} \neg\langle\langle 1,2\rangle\rangle\Box\mathsf{pos}_1$ (when in q_0, robot 2 considers it possible that the current state of the system is q_1, in which case all the hope is gone). So, do the robots know how to play to achieve anything? Yes, for example they know how to make the carriage *reach* a particular state eventually: $\mathfrak{M}_2, q_0 \models_{ir} \langle\langle 1,2\rangle\rangle\Diamond\mathsf{pos}_1$ etc. – it suffices that one of the robots pushes all the time and the other waits all the time. Still, $\mathfrak{M}_2, q_0 \models_{ir} \neg\langle\langle 1,2\rangle\rangle\Diamond\Box\mathsf{pos}_x$ (for $x = 0,1,2$): there is no memoryless strategy for the robots to bring the carriage to a particular position and keep it there forever.

Most of the above properties hold for the *iR* semantics as well. Note, however, that for robots with perfect recall we do have that $\mathfrak{M}_2, q_0 \models_{iR} \langle\langle 1,2\rangle\rangle\Diamond\Box\mathsf{pos}_x$. The right strategy is that one robot pushes and the other waits for the first 3 steps. After that, they know their current position exactly, and can go straight the specified position.

5.2.4 Other Subsets of $\mathcal{L}_{ATL^*}$

5.2.4.1 Coalition Logic

***C**oalition **L**ogic* (**CL**), introduced in [333], is another logic for modeling and reasoning about strategic abilities of agents. The main construct of **CL**, $[A]\varphi$, expresses that coalition A can bring about φ in a single-step game.

Definition 5.12 (Language $\mathcal{L}_{CL}$ [333]). The *language* $\mathcal{L}_{CL}$ is given by all formulae generated by the following grammar: $\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid [A]\varphi$, where $\mathsf{p} \in \Pi$ and $A \subseteq \mathbb{A}\mathrm{gt}$.

In [333], *coalitional models* were chosen as semantics for $\mathcal{L}_{CL}$. These models are given by (St, E, π) consisting of a set of states St, a *playable effectivity function* E, and a valuation function π. The effectivity function determines the outcomes that a coalition is effective for, i.e., given a set $X \subseteq St$ of states a coalition C is said to be effective for X iff it can enforce the next state to be in X. However, in [201] it was shown that CGS provide an equivalent semantics, and that **CL** can be seen as the *next-time fragment* of **ATL**. Hence, for this presentation we will interpret $\mathcal{L}_{CL}$-formulae over CGS's, and consider $[A]\varphi$ as an abbreviation for $\langle\langle A\rangle\rangle \quad \varphi$. The various logics $\mathbf{CL}_{xy}$ that we can obtain using the semantics $\models_{xy}$ for $x \in \{i, I\}$ and $y \in \{r, R\}$ are defined analogously to $\mathbf{ATL}_{xy}$.

5.2.4.2 ATL$^+$

The language $\mathcal{L}_{ATL^+}$ is the subset of $\mathcal{L}_{ATL^*}$ that requires each temporal operator to be followed by a state formula, but allows for Boolean combinations of path

subformulae. The formula $\langle\langle A\rangle\rangle(\Box \mathsf{p} \wedge \Diamond \mathsf{q})$, for instance, is an $\mathcal{L}_{ATL^+}$-formula but not an $\mathcal{L}_{ATL}$-formula. Formally, the language is given as follows:

Definition 5.13 (Language $\mathcal{L}_{ATL^+}$). The *language* $\mathcal{L}_{ATL^+}$ is given by all formulae generated by the following grammar: $\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle A\rangle\rangle\gamma$ where $\gamma ::= \neg\gamma \mid \gamma \wedge \gamma \mid \varphi \mathcal{U} \varphi \mid \quad \varphi$, $A \subseteq \mathbb{A}\mathrm{gt}$ and $\mathsf{p} \in \Pi$.

We define the various logics emerging from $\mathcal{L}_{ATL^+}$ and the different semantics analogously to the case of $\mathcal{L}_{ATL}$. The logic **ATL**$^+$ is strictly more expressive than **ATL** (contrary to common belief, each **ATL**$^+$ formula can only be translated to an equivalent **ATL** formula if the "release" or "weak until" operator is added to the language of $\mathcal{L}_{ATL}$ [92, 208, 279]) but it enables a more succinct encoding of properties (this follows from the results in [432]). In Section 5.3 we will see that the more succinct language has its price: The model checking problem becomes computationally more expensive.

5.2.5 Summary, Notation, and Related Work

We have recalled the linear-time temporal logic and two versions of the computation tree logics for reasoning about purely temporal systems. Then, we presented several variants of the alternating-time temporal logics: the richest underlying language $\mathcal{L}_{ATL^*}$, somewhat restricted variants $\mathcal{L}_{ATL^+}$ and $\mathcal{L}_{ATL}$, and $\mathcal{L}_{CL}$ which can be seen as a very limited fragment of $\mathcal{L}_{ATL}$. All these languages were coupled with four alternative semantics that result from combining perfect/imperfect information with perfect recall/memoryless strategies (the *IR*, *Ir*, *iR*, and *ir*-semantics).

The resulting logics were defined with respect to the notation introduced by Schobbens [388] to refer to a strategic logic using a specific semantics. For $ID \in \{\mathbf{CL}, \mathbf{ATL}, \mathbf{ATL}^+, \mathbf{ATL}^*\}$, $x \in \{I, i\}$, and $y \in \{R, r\}$, we used ID_{xy} to refer to the logic over the language $\mathcal{L}_{ID}$ using the xy-semantics $\models_{xy}$.

In this chapter we are concerned with model checking strategic logics and thus take on a semantic view. Naturally, there is more than that to be studied. In [200] a complete *axiomatization* for $\mathbf{ATL}_{IR}$ is presented. Also the *satisfiability problem* of $\mathbf{ATL}_{IR}$ and $\mathbf{ATL}^*_{IR}$ has been considered by researchers: The problem was proven *EXPTIME*-complete for $\mathbf{ATL}_{IR}$ [150, 428] and even *2EXPTIME*-complete for $\mathbf{ATL}^*_{IR}$ [384]. Axiomatization and satisfiability of other variants of alternating-time temporal logic still remains open.

5.3 Standard Model Checking Complexity Results

In this section we consider model checking for the logics introduced in Section 5.2. The process of model checking seeks to answer the question whether a given for-

mula φ is satisfied in a state q of model $\mathfrak{M}$. Formally, *local model checking* is the decision problem that determines membership in the set

$$\mathsf{MC}(\mathcal{L}, \mathsf{Struc}, \models) := \{(\mathfrak{M}, q, \varphi) \in \mathsf{Struc} \times \mathcal{L} \mid \mathfrak{M}, q \models \varphi\},$$

where $\mathcal{L}$ is a logical language, Struc is a class of (pointed) models for $\mathcal{L}$ (i.e. a tuple consisting of a model and a state), and $\models$ is a semantic satisfaction relation compatible with $\mathcal{L}$ and Struc. We omit parameters if they are clear from context, e.g., we use MC(**CTL**) to refer to model checking of **CTL** over the class of (pointed) Kripke models and the introduced semantics.

It is often useful to compute the set of states in $\mathfrak{M}$ that satisfy formula φ instead of checking if φ holds in a particular state. This variant of the problem is known as *global model checking*. It is easy to see that, for the settings we consider here, the complexities of local and global model checking coincide, and the algorithms for one variant of model checking can be adapted to the other variant in a simple way. As a consequence, we will use both notions of model checking interchangeably.

In the following, we are interested in the decidability and the computational complexity of determining whether an input instance $(\mathfrak{M}, q, \varphi)$ belongs to MC(...). The complexity is always relative to the *size* of the instance; in the case of model checking, it is the size of the representation of the model and the representation of the formula that we use. Thus, in order to establish the complexity, it is necessary to fix how we *represent* the input and how we *measure* its size. In this section, we consider explicit representation of models and formulae, together with the "standard" input measure, where the size of the model ($|\mathfrak{M}|$) is given by the *number of transitions* in $\mathfrak{M}$, and the size of the formula ($|\varphi|$) is given by its *length* (i.e., the number of elements it is composed of, apart from parentheses). For example, the model in Figure 5.2 includes 12 (labeled) transitions, and the formula $\langle\langle 1 \rangle\rangle \quad (\mathsf{pos}_0 \vee \mathsf{pos}_1)$ has length 5.

5.3.1 Model Checking Temporal Logics

An excellent survey on the model checking complexity of temporal logics has been presented in [387]. Here, we only recall the results relevant for the subsequent analysis of strategic logics.

Let $\mathfrak{M}$ be a Kripke model and q be a state in the model. Model checking a $\mathcal{L}_{CTL}/\mathcal{L}_{CTL^*}$-formula φ in $\mathfrak{M}, q$ means to determine whether $\mathfrak{M}, q \models \varphi$, i.e., whether φ holds in $\mathfrak{M}, q$. For **LTL**, checking $\mathfrak{M}, q \models \varphi$ means that we check the *validity* of φ in the pointed model $\mathfrak{M}, q$, i.e., whether φ holds *on all the paths* in $\mathfrak{M}$ that start from q (equivalent to **CTL*** model checking of formula $\mathsf{A}\varphi$ in $\mathfrak{M}, q$, cf. Remark 5.2).

It has been known for a long time that formulae of **CTL** can be model-checked in time linear with respect to the size of the model and the length of the formula [101], whereas formulae of **LTL** and **CTL*** are significantly harder to verify.

function $mcheck(\mathfrak{M}, \varphi)$**.**
Model checking formulae of CTL. Returns the exact subset of St for which formula φ holds.
case $\varphi \equiv p$ **:** return $\{q \in St \mid p \in \pi(q)\}$
case $\varphi \equiv \neg\psi$ **:** return $St \setminus mcheck(\mathfrak{M}, \psi)$
case $\varphi \equiv \psi_1 \wedge \psi_2$ **:** return $mcheck(\mathfrak{M}, \psi_1) \cap mcheck(\mathfrak{M}, \psi_2)$
case $\varphi \equiv \mathsf{E} \quad \psi$ **:** return $pre(mcheck(\mathfrak{M}, \psi))$
case $\varphi \equiv \mathsf{E}\Box\psi$ **:**
  $Q_1 := Q; \quad Q_2 := Q_3 := mcheck(\mathfrak{M}, \psi);$
  while $Q_1 \not\subseteq Q_2$ **do** $Q_1 := Q_1 \cap Q_2; \ Q_2 := pre(Q_1) \cap Q_3$ **od**;
  return Q_1
case $\varphi \equiv \mathsf{E}\psi_1 \mathcal{U} \psi_2$ **:**
  $Q_1 := \emptyset; \quad Q_2 := mcheck(\mathfrak{M}, \psi_2); \quad Q_3 := mcheck(\mathfrak{M}, \psi_1);$
  while $Q_2 \not\subseteq Q_1$ **do** $Q_1 := Q_1 \cup Q_2; \ Q_2 := pre(Q_1) \cap Q_3$ **od**;
  return Q_1
end case

Fig. 5.4 The **CTL** model checking algorithm from [99].

Theorem 5.2 (CTL [101, 387]). *Model checking* **CTL** *is P-complete, and can be done in time* $\mathbf{O}(|\mathfrak{M}| \cdot |\varphi|)$*, where* $|\mathfrak{M}|$ *is given by the number of transitions.*

Proof (Sketch). The algorithm determining the states in a model at which a given formula holds is presented in Figure 5.4. The lower bound (*P*-hardness) can be for instance proven by a reduction of the tiling problem [387]. □

Theorem 5.3 (LTL [286, 406, 422]). *Model checking* **LTL** *is PSPACE-complete, and can be done in time* $2^{\mathbf{O}(|\varphi|)}\mathbf{O}(|\mathfrak{M}|)$*, where* $|\mathfrak{M}|$ *is given by the number of transitions.*

Proof (Sketch). We sketch the approach given in [422]. Firstly, given an $\mathcal{L}_{LTL}$-formula φ, a Büchi automaton $\mathcal{A}_{\neg\varphi}$ of size $2^{\mathbf{O}(|\varphi|)}$ accepting exactly the paths satisfying $\neg\varphi$ is constructed. The pointed Kripke model $\mathfrak{M}, q$ can directly be interpreted as a Büchi automaton $\mathcal{A}_{\mathfrak{M},q}$ of size $\mathbf{O}(|\mathfrak{M}|)$ accepting all possible paths in the Kripke model starting in q. Then, the model checking problem reduces to the non-emptiness check of $L(\mathcal{A}_{\mathfrak{M},q}) \cap L(\mathcal{A}_{\neg\varphi})$ which can be done in time $\mathbf{O}(|\mathfrak{M}|) \cdot 2^{\mathbf{O}(|\varphi|)}$ by constructing the product automaton. (Emptiness can be checked in linear time wrt to the size of the automaton.) A *PSPACE*-hardness proof can for instance be found in [406]. □

The hardness of $\mathbf{CTL}^*$ model checking is immediate from Theorem 5.3 as $\mathcal{L}_{LTL}$ can be seen as a fragment of $\mathcal{L}_{CTL^*}$. For the proof of the upper bound one combines the **CTL** and **LTL** model checking techniques. Consider a $\mathcal{L}_{CTL^*}$-formula φ which contains a state subformula $\mathsf{E}\psi$ where ψ is a pure $\mathcal{L}_{LTL}$-formula. Firstly, we can use **LTL** model checking to determine all state which satisfy $\mathsf{E}\psi$ (these are all states q in which the $\mathcal{L}_{LTL}$-formula $\neg\psi$ is *not* true) and label them by a fresh propositional symbol , say p, and replace $\mathsf{E}\psi$ in φ by p as well. Applying this procedure recursively

yields a pure $\mathcal{L}_{CTL}$-formula which can be verified in polynomial time. Hence, the procedure can be implemented by an oracle machine of type $P^{PSPACE} = PSPACE$ (the **LTL** model checking algorithm might be employed polynomially many times). Thus, the complexity for **CTL*** is the same as for **LTL**.

Theorem 5.4 (CTL* [101, 161]). *Model checking* **CTL*** *is PSPACE-complete, and can be done in time* $2^{\mathbf{O}(|\varphi|)}\mathbf{O}(|\mathfrak{M}|)$*, where* $|\mathfrak{M}|$ *is given by the number of transitions.*

In Section 5.2.4 we introduced **ATL**$^+$, a variant of **ATL**. As the model checking algorithm for **ATL**$^+$ will rely on the complexity of **CTL**$^+$ model checking,[6] we mention the latter result here.

Theorem 5.5 (CTL$^+$ [280]). *Model checking* **CTL**$^+$ *is* Δ_2^P*-complete in the number of transitions in the model and the length of the formula.*

5.3.2 *Model Checking* ATL *and* CL*: Perfect Information*

One of the main results concerning **ATL** states that its formulae can also be model-checked in deterministic linear time, analogously to **CTL**. It is important to emphasize, however, that the result is relative to the number of transitions in the model and the length of the formula. In Section 5.4 we will discuss an alternative input measure in terms of agents, states, and the length of the formula, and show that this causes a substantial increase in complexity.

The **ATL** model checking algorithm from [14] is presented in Figure 5.5. The algorithm employs the well-known fixpoint characterizations of strategic-temporal modalities:

$$\langle\langle A\rangle\rangle\Box\varphi \leftrightarrow \varphi \wedge \langle\langle A\rangle\rangle \quad \langle\langle A\rangle\rangle\Box\varphi$$
$$\langle\langle A\rangle\rangle\varphi_1 \mathcal{U} \varphi_2 \leftrightarrow \varphi_2 \vee \varphi_1 \wedge \langle\langle A\rangle\rangle \quad \langle\langle A\rangle\rangle\varphi_1 \mathcal{U} \varphi_2,$$

and computes a winning strategy step by step (if it exists). That is, it starts with the appropriate candidate set of states ($\emptyset$ for $\mathcal{U}$ and the whole set St for $\Box$), and iterates backwards over A's one-step abilities until the set gets stable. It is easy to see that the algorithm needs to traverse each transition at most once per subformula of φ. Note that it does not matter whether perfect recall or memoryless strategies are used: The algorithm is correct for the *IR*-semantics, but it always finds an *Ir*-strategy. Thus, for an $\mathcal{L}_{ATL}$-formula $\langle\langle A\rangle\rangle\gamma$, if A have an *IR*-strategy to enforce γ, they also have an *Ir*-strategy to obtain it.

[6] **CTL**$^+$ is defined analogously to **ATL**$^+$: Boolean combinations of path formulae are allowed in the scope of path quantifiers.

function $mcheck(M,\varphi)$**.**

ATL model checking. Returns the set of states in model $M = \langle \mathbb{A}\text{gt}, St, \Pi, \pi, o\rangle$ for which formula φ holds.

case $\varphi \in \Pi$ **:** return $\pi(p)$
case $\varphi = \neg\psi$ **:** return $St \setminus mcheck(M,\psi)$
case $\varphi = \psi_1 \vee \psi_2$ **:** return $mcheck(M,\psi_1) \cup mcheck(M,\psi_2)$
case $\varphi = \langle\langle A\rangle\rangle \bigcirc \psi$ **:** return $pre(M,A,mcheck(M,\psi))$
case $\varphi = \langle\langle A\rangle\rangle \Box\psi$ **:**
  $Q_1 := St$; $Q_2 := mcheck(M,\psi)$; $Q_3 := Q_2$;
  while $Q_1 \not\subseteq Q_2$
  do $Q_1 := Q_2$; $Q_2 := pre(M,A,Q_1) \cap Q_3$ **od**;
  return Q_1
case $\varphi = \langle\langle A\rangle\rangle \psi_1 \mathcal{U} \psi_2$ **:**
  $Q_1 := \varnothing$; $Q_2 := mcheck(M,\psi_1)$;
  $Q_3 := mcheck(M,\psi_2)$;
  while $Q_3 \not\subseteq Q_1$
  do $Q_1 := Q_1 \cup Q_3$; $Q_3 := pre(M,A,Q_1) \cap Q_2$ **od**;
  return Q_1
end case

function $pre(M,A,Q)$**.**

Auxiliary function; returns the exact set of states Q' such that, when the system is in a state $q \in Q'$, agents A can cooperate and enforce the next state to be in Q.

return $\{q \mid \exists\alpha_A \forall\alpha_{\mathbb{A}\text{gt}\setminus A}\ o(q,\alpha_A,\alpha_{\mathbb{A}\text{gt}\setminus A}) \in Q\}$

Fig. 5.5 The **ATL** model checking algorithm from [14]

Theorem 5.6 ($\mathbf{ATL}_{Ir}$ and $\mathbf{ATL}_{IR}$ [14]). *Model checking* $\mathbf{ATL}_{Ir}$ *and* $\mathbf{ATL}_{IR}$ *is P-complete, and can be done in time* $\mathbf{O}(|\mathfrak{M}| \cdot |\varphi|)$*, where* $|\mathfrak{M}|$ *is given by the number of transitions in* $\mathfrak{M}$*.*

Proof (Sketch). Each case of the algorithm is called at most $\mathbf{O}(|\varphi|)$ times and terminates after $\mathbf{O}(|\mathfrak{M}|)$ steps [14]. The latter is shown by translating the model to a two-player game [14], and then solving the "invariance game" on it in polynomial time [30]. Hardness is shown by a reduction of reachability in And-Or-Graphs, which was shown to be *P*-complete in [248], to model checking the (constant) $\mathcal{L}_{ATL}$-formula $\langle\langle 1\rangle\rangle \Diamond \mathsf{p}$ in a two player game. In each Or-state it is the turn of player 1 and in each And-state it is player 2's turn [14]. □

In the next theorem, we show that the model checking of coalition logic is as hard as for **ATL**. To our knowledge, this is a new result; the proof is done by a slight variation of the hardness proof for **ATL** in [14] (cf. the proof of Theorem 5.6).

Theorem 5.7 ($\mathbf{CL}_{Ir}$ and $\mathbf{CL}_{IR}$). *Model checking* $\mathbf{CL}_{Ir}$ *and* $\mathbf{CL}_{IR}$ *is P-complete, and can be done in time* $\mathbf{O}(|\mathfrak{M}| \cdot |\varphi|)$*, where* $|\mathfrak{M}|$ *is given by the number of transitions in* $\mathfrak{M}$*.*

Proof. The upper bound follows from the fact that $\mathcal{L}_{CL}$ is a sublanguage of $\mathcal{L}_{ATL}$. We show *P*-hardness by the following adaption of the reduction of And-Or-Graph reachability from [14]. Firstly, we observe that if a state y is reachable from x in graph G then it is also reachable via a path whose length is bounded by the number n of states in the graph. Like in the proof of Theorem 5.6, we take G to be a turned-based CGS in which player 1 "owns" all the Or-states and player 2 "owns" all the And-states. We also label node y with a special proposition y, and replace all the transitions outgoing from y with a deterministic loop. Now, we have that y is reachable from x in G iff $G, x \models \underbrace{\langle\langle 1\rangle\rangle \quad \ldots \langle\langle 1\rangle\rangle}_{n\text{-times}} \mathsf{y}$. The reduction uses only logarithmic space. □

It is worth pointing out, however, that checking strategic properties in one-step games is somewhat easier. We recall that AC^0 is the class corresponding to constant-depth, unbounded-fanin, polynomial-size Boolean circuits with AND, OR, and NOT gates [185]. We call a formula *flat* if it contains no nested cooperation modalities. Moreover, a formula is *simple* if it is flat and does not include Boolean connectives. For example, the language of "simple **CL**" consists only of formulae p and $\langle\langle A\rangle\rangle \quad p$, for $p \in \Pi$ and $A \subseteq \mathbb{A}\mathrm{gt}$.

Theorem 5.8 (Simple $\mathbf{CL}_{Ir}$ and $\mathbf{CL}_{IR}$ [279]). *Model checking "Simple $\mathbf{CL}_{Ir}$" and "Simple $\mathbf{CL}_{IR}$" with respect to the number of transitions in the model and the length of the formula is in AC^0.*

Proof (Sketch). For $\mathfrak{M}, q \models \langle\langle A\rangle\rangle \quad \mathsf{p}$, we construct a 3-level circuit [279]. On the first level, we assign one AND gate for every possible coalition B and B's collective choice α_B; the output of the gate is "true" iff α_B leads to a state satisfying p for every response of $\mathbb{A}\mathrm{gt} \setminus B$. On the second level, there is one OR gate per possible coalition B that connects all the B's gates from the first level and outputs "true" iff there is any successful strategy for B. On the third level, there is a single AND gate that selects the right output (i.e., the one for coalition A). □

5.3.3 *Model Checking* ATL *and* CL*: Imperfect Information*

In contrast to the perfect information setting, analogous fixpoint characterizations do *not* hold for the incomplete information semantics over $\mathcal{L}_{ATL}$ because the choice of a particular action at a state q has non-local consequences: It automatically fixes choices at all states q' indistinguishable from q for the coalition A. Moreover, the agents' ability to *identify* a strategy as winning also varies throughout the game in an arbitrary way (agents can learn as well as forget). This suggests that winning strategies cannot be synthesized incrementally. Note that, in order to check $\mathfrak{M}, q \models \langle\langle A\rangle\rangle\gamma$ (where γ includes no nested cooperation modalities), the following procedure suffices. Firstly, we guess a uniform strategy s_A of team A (by calling an *NP* oracle),

and then verify the strategy by pruning $\mathfrak{M}$ accordingly (removing all the transitions that are not going to be executed according to s_A) and model-checking the $\mathcal{L}_{CTL}$-formula $\mathsf{A}\gamma$ in the resulting model. For nested cooperation modalities, we proceed recursively (bottom up). Since model checking **CTL** can be done in polynomial deterministic time, the procedure runs in polynomial deterministic time with calls to an NP oracle, which demonstrates the inclusion in $\Delta_2^P = P^{NP}$ [388]. As it turns out, a more efficient procedure does not exist, which is confirmed by the following result.

Theorem 5.9 ($\mathbf{ATL}_{ir}$ [259,388]). *Model checking $\mathbf{ATL}_{ir}$ is Δ_2^P-complete in the number of transitions in the model and the length of the formula.*

Proof (Sketch). The discussion above proves the membership in Δ_2^P. Δ_2^P-hardness was shown in [259] through a reduction of *sequential satisfiability* ($\mathbf{SNSAT}_2$), a standard Δ_2^P-complete problem [280]. The idea is that there are two agents where one agent tries to verify a (nested) propositional formula and a second agent tries to refute it. A winning strategy of the "verifier agent" corresponds to a satisfying valuation of the formula. Uniformity of the verifier's strategy is needed to ensure that identical proposition symbols, occurring at different places in the formula, are assigned the same truth values. □

Now we consider the incomplete information setting for Coalition Logic. It is easy to see that the *iR*- and *ir*-sematics are equivalent for $\mathcal{L}_{CL}$ since is the only temporal operator, and thus only the first action in a strategy matters. As a consequence, whenever there is a successful *iR*-strategy for agents A to enforce φ, then there is also an *ir*-strategy for A to obtain the same. Perfect recall of the history does not matter in one-step games.

Theorem 5.10 ($\mathbf{CL}_{ir}$ and $\mathbf{CL}_{iR}$). *Model checking $\mathbf{CL}_{ir}$ and $\mathbf{CL}_{iR}$ is P-complete wrt the number of transitions in the model and the length of the formula, and can be done in time $\mathbf{O}(|\mathfrak{M}| \cdot |\varphi|)$.*

Proof. The P-hardness follows from Theorem 5.7 (perfect information CGS's can be seen as a special kind of ICGSwhere the indistinguishability relations contain only the reflexive loops). For the upper bound, we use the following algorithm. For $\mathfrak{M}, q \models \langle\langle A \rangle\rangle$ p, we check if there is a collective action α_A such that for all responses $\alpha_{\mathbb{A}\mathrm{gt}\setminus A}$ we have that $\bigcup_{\{q' | q \sim_A q'\}} \{o(q', \alpha_A, \alpha_{\mathbb{A}\mathrm{gt}\setminus A})\} \subseteq \pi(\mathsf{p})$. For $\langle\langle A \rangle\rangle$ φ with nested cooperation modalities, we proceed recursively (bottom up). □

Theorem 5.11 (Simple $\mathbf{CL}_{ir}$ and $\mathbf{CL}_{iR}$). *Model checking "simple" formulae of $\mathbf{CL}_{ir}$ and $\mathbf{CL}_{iR}$ with respect to the number of transitions in the model and the length of the formula is in AC^0.*

Proof. For $\mathfrak{M}, q \models \langle\langle A \rangle\rangle$ p, we extend the procedure from [279] by creating one copy of the circuit per $q' \in \mathrm{img}(q, \sim_A)$. Then, we add a single AND gate on the fourth level of the circuit, that takes the output of those copies and returns "true" iff A have a strategy that is successful from all states indistinguishable from q. □

That leaves us with the issue of $\mathcal{L}_{ATL}$ with the semantics assuming imperfect information and perfect recall. To our knowledge, there is no formal proof in the literature regarding the complexity of model checking $\mathcal{L}_{ATL}$ with *iR*-strategies. However, the problem is commonly believed to be undecidable.

*Conjecture 5.1 (***ATL$_{iR}$** *[14]).* Model checking **ATL$_{iR}$** is undecidable.

5.3.4 Model Checking ATL* *and* ATL$^+$

We now turn to model checking logics over broader subsets of $\mathcal{L}_{ATL^*}$. In the first case we consider perfect recall strategies in the perfect information setting. The complexity results established here are based on an automata-theoretic approach which is explained below.

Let $\mathfrak{M}$ be a CGS and $\langle\langle A\rangle\rangle\psi$ be an $\mathcal{L}_{ATL^*}$-formula (where we assume that ψ is an $\mathcal{L}_{LTL}$-formula). Given a strategy s_A of A and a state q in $\mathfrak{M}$ the model can be unfolded into a q-rooted tree representing all possible behaviors with agents A following their strategy s_A. This structure can be seen as the tree *induced* by $out(q, s_A)$ and we will refer to it as a (q,A)-*execution tree*. Note that every strategy profile for A may result in a different execution tree. Now, a Büchi tree automaton $\mathcal{A}_{\mathfrak{M},q,A}$ can be constructed that accepts exactly the (q,A)-execution trees [14].

Secondly, it was shown that one can construct a Rabin tree automaton which accepts all trees that satisfy the $\mathcal{L}_{CTL^*}$-formula $\mathsf{A}\psi$ [162]. Hence, the $\mathcal{L}_{ATL^*}$-formula $\langle\langle A\rangle\rangle\psi$ is satisfied in $\mathfrak{M},q$ if there is a tree accepted by $\mathcal{A}_{\mathfrak{M},q,A}$ (i.e., it is a (q,A)-execution tree) and by $\mathcal{A}_\psi$ (i.e., it is a model of $\mathsf{A}\psi$).

Theorem 5.12 (ATL$^*_{IR}$ [14]). *Model checking* **ATL$^*_{IR}$** *is 2EXPTIME-complete in the number of transitions in the model and the length of the formula.*

Proof (Sketch). We briefly analyze the complexity for the procedure described above. Firstly, the Büchi tree automaton $\mathcal{A}_{\mathfrak{M},q,A}$ is built by considering the states A is effective for [14]. That is, in a state of the automaton corresponding to a state $q \in St$ of $\mathfrak{M}$ the automaton nondeterministically chooses a sequence $(q'_1, q'_2, \ldots, q'_n)$ of successors of q such that A has a common action to guarantee that the system will end up in one of the states $\{q'_1, q'_2, \ldots, q'_n\}$ in the next step. It is assumed that the sequence is minimal. Incrementally, this models any s_A strategy of A and thus accepts all (q,A)-execution trees. The transition function of the automaton is constructed in the described way. As the number of transitions in each state of the automaton is bounded by the move combinations of agents A the size of the automaton, $|\mathcal{A}_{\mathfrak{M},q,A}|$, is bounded by $\mathbf{O}(|\mathfrak{M}|)$. All states are defined as acceptance states, such that $\mathcal{A}_{\mathfrak{M},q,A}$ accepts all possible execution trees of A.

Following the construction of [162], the automaton $\mathcal{A}_\psi$ is a Rabin tree automaton with $2^{2^{\mathbf{O}(|\psi|)}}$ states and $2^{\mathbf{O}(|\psi|)}$ Rabin pairs.

The product automaton $\mathcal{A}_\psi \times \mathcal{A}_{\mathfrak{M},q,A}$, accepting the trees accepted by both automata, is a Rabin tree automaton with $n := \mathbf{O}(|\mathcal{A}_\psi| \cdot |\mathcal{A}_{\mathfrak{M},q,A}|)$ many states and $r := 2^{\mathbf{O}(|\psi|)}$ many Rabin pairs (note that $\mathcal{A}_{\mathfrak{M},q,A}$ can be seen as a Rabin tree automaton with one Rabin pair composed of the states of the automaton and the empty set). Finally, to determine whether the language accepted by the product automaton is empty can be done in time $\mathbf{O}(n \cdot r)^{3r}$ [160, 342]; hence, the algorithm runs in time $|\mathfrak{M}|^{2^{\mathbf{O}(|\psi|)}}$ (it might be employed at each state of the model and for each subformula).

The lower bound is shown by a reduction of the *2EXPTIME*-complete problem of the realizability of **LTL**-formulae [14, 342, 373]. □

The next result shows that model checking $\mathcal{L}_{ATL^*}$ *with memoryless strategies* is no worse than for **LTL** and $\mathbf{CTL}^*$ for both perfect and imperfect information.

Theorem 5.13 ($\mathbf{ATL}^*_{ir}$ and $\mathbf{ATL}^*_{Ir}$ [388]). *Model checking $\mathbf{ATL}^*_{ir}$ and $\mathbf{ATL}^*_{Ir}$ is PSPACE-complete in the number of transitions in the model and the length of the formula.*

Proof (Sketch). $\mathcal{L}_{LTL}$ is contained in $\mathcal{L}_{ATL^*}$ which renders $\mathcal{L}_{ATL^*}$ with the perfect information memoryless semantics to be at least *PSPACE*-hard.

On the other hand, there is a *PSPACE* algorithm for model checking $\mathcal{L}_{ATL^*}$ with the imperfect information memoryless semantics. Consider the formula $\langle\langle A\rangle\rangle\psi$ where ψ is an $\mathcal{L}_{LTL}$-formula. Then, an *ir*-strategy s_A for A is guessed and the model is "trimmed" according to s_A, i.e. all transitions which cannot occur by following s_A are removed. Note that a memoryless strategy can be guessed in polynomially many steps, and hence also using only polynomially many memory cells. In the new model the $\mathcal{L}_{CTL^*}$-formula $\mathsf{A}\psi$ is checked. This procedure can be performed in NP^{PSPACE}, which renders the complexity of the whole language to be in $P^{NP^{PSPACE}} = PSPACE$. □

We consider the more limited language $\mathcal{L}_{ATL^+}$. Boolean combinations of path formulae prevent us from using the fixed-point characterizations for model checking. Instead, given a formula $\langle\langle A\rangle\rangle\psi$ with no nested cooperation modalities, we can guess a (memoryless) strategy of A, "trim" the model accordingly, and model-check the $\mathcal{L}_{CTL^+}$-formula $\mathsf{A}\psi$ in the resulting model. Since the model checking problem for $\mathbf{CTL}^+$ is Δ_2^P-complete, we get that the overall procedure runs in time $\Delta_2^{P\Delta_2^P} = \Delta_3^P$ [388].

Theorem 5.14 ($\mathbf{ATL}^+_{ir}$ and$\mathbf{ATL}^+_{Ir}$ [388]). *Model checking $\mathbf{ATL}^+_{ir}$ and $\mathbf{ATL}^+_{Ir}$ is Δ_3^P-complete in the number of transitions in the model and the length of the formula.*

Proof (Sketch). The above procedure shows the membership. Note that in the incomplete information case one has to guess a *uniform* strategy. Again, it is essential that a strategy can be guessed in *polynomially many steps*, which is indeed the case for *Ir*- and *ir*-strategies. The hardness proof can be obtained by a reduction of the standard Δ_3^P-complete problem $\mathbf{SNSAT}_3$, cf. [388] for the construction. □

What about $\mathbf{ATL}^{+}_{\mathit{IR}}$? It has been believed that verification with $\mathcal{L}_{ATL^{+}}$ is Δ^{P}_{3}-complete for perfect recall strategies, too. However, it turns out that the complexity of $\mathbf{ATL}^{+}_{\mathit{IR}}$ model checking is much harder, namely *PSPACE* [92]. Since the Δ^{P}_{3}-completeness for memoryless semantics *is* correct, we get that memory makes verification harder already for $\mathcal{L}_{ATL^{+}}$, and not just for $\mathcal{L}_{ATL^{*}}$ as it was believed before.

Theorem 5.15 ($\mathbf{ATL}^{+}_{\mathit{IR}}$ [92]). *Model checking $\mathbf{ATL}^{+}_{\mathit{IR}}$ is PSPACE-complete with respect to the number of transitions in the model and the length of the formula. It is PSPACE-complete even for turn-based models with two agents and "flat" $\mathbf{ATL}^{+}$ formulae.*

Proof (Sketch). Consider the $\mathcal{L}_{ATL^{+}}$-formula $\langle\langle A\rangle\rangle\gamma$ where γ does not contain any further cooperation modalities. The upper bound can be proven by constructing an alternating Turing Machine that first produces (by alternatingly guessing the "best" choices of the proponents and the "most damaging" responses of the opponents) the relevant part of a path (whose length is asymptotically bounded by the product of the length of the formula and the number of states in the model) that suffices to determine the truth of an $\mathcal{L}_{ATL^{+}}$-formula. Then, we implement the game-theoretical semantics of propositional logic [229] as a game between the verifier (who controls disjunction) and the refuter (controlling conjunction). The machine runs in time $\mathbf{O}(nkl)$ where n (resp. k and l) denotes the number of states (resp. number of agents and length of the formula), cf. [92] for details.

Hardness is proved by a reduction of QSAT. A perfect recall strategy of the proponents is used to assign consistent valuations (step-by-step) to propositional variables that they control; analogously for the opponents. Thereby, the proponents control the existentially quantified variables and the opponents the universally quantified ones. An $\mathcal{L}_{ATL^{+}}$-formula is used to describe such valid assignments; i.e. truth values must be ascribed to variables in a uniform way. For the complete construction we refer to [92] again. □

Note that the input size only depends on the number of *states and agents* in the model and length of the formula which is important for the complexity result given in Theorem 5.25 about non-standard input measures.

The following conjectures are immediate consequences of Conjecture 5.1 as $\mathcal{L}_{ATL}$ is a fragment of $\mathcal{L}_{ATL^{*}}$ as well as $\mathcal{L}_{ATL^{+}}$.

Conjecture 5.2 ($\mathbf{ATL}^{*}_{\mathit{iR}}$). Model checking $\mathbf{ATL}^{*}_{\mathit{iR}}$ is undecidable.

Conjecture 5.3 ($\mathbf{ATL}^{+}_{\mathit{iR}}$). Model checking $\mathbf{ATL}^{+}_{\mathit{iR}}$ is undecidable.

Figure 5.3.4 presents an overview of the model checking complexity results for explicit models.

	Ir	*IR*	*ir*	*iR*
Simple $\mathcal{L}_{CL}$	AC^0	AC^0	AC^0	AC^0
$\mathcal{L}_{CL}$	*P*	*P*	*P*	*P*
$\mathcal{L}_{ATL}$	*P*	*P*	Δ_2^P	Undecidable†
$\mathcal{L}_{ATL^+}$	Δ_3^P	*PSPACE*	Δ_3^P	Undecidable†
$\mathcal{L}_{ATL^*}$	*PSPACE*	*2EXPTIME*	*PSPACE*	Undecidable†

Fig. 5.6 Overview of the model checking complexity results for explicit models. All results except for "Simple **CL**" are completeness results. Each cell represents the logic over the language given in the row using the semantics given in the column. † These problems are believed to be undecidable, though no formal proof has been proposed yet (cf. Conjectures 5.1, 5.2, and 5.3).

5.4 Complexity for Implicit Models: States and Agents

We have seen several complexity results for the model checking problem in logics like **LTL**, **CTL**, and **ATL**. Some of these results are quite attractive: one usually cannot hope to achieve verification with complexity better than *linear*.

However, it is important to remember that these results measure the complexity with respect to the *size of the underlying model*. Often, these models are so big, that an *explicit* representation is not possible and we have to represent the model in a "compressed" way. To give a simple illustration, consider the famed primality problem: checking whether a given natural number n is prime. The well-known algorithm uses $\sqrt{n}$-many divisions and thus runs in polynomial time *when the input is represented in unary*. But a symbolic representation of n needs only $\log(n)$ bits and thus the above algorithm runs in *exponential* time with respect to its size. This does not necessarily imply that the problem itself is of exponential complexity. In fact, the famous and deep result of Agrawal, Kayal and Saxena shows that the primality problem *can* be solved in polynomial time.

We will consider model checking of temporal and strategic logics for such highly compressed representations (in terms of *state space compression* and *modularization*) in Section 5.5. Such a rigorous compressed representation is not the only way in which the model checking complexity can be influenced. Another important factor is how we encode the transition function. So far, we assumed that the size of a model is measured with respect to the number of transitions in the model.

In this section we consider the complexity of the model checking problem *with respect to the number of states, agents, and an implicitly encoded transition function* rather than the (explicit) number of transitions. It is easy to see that, for CGS's, the number of transitions can be exponential in the number of states and agents. Therefore, all the algorithms presented in Section 5.3 give us only exponential time bounds provided that the transition function is encoded sufficiently small.

Observation 1 ([14,257]) *Let n be the number of states in a concurrent game structure $\mathfrak{M}$, let k denote the number of agents, and d the maximal number of available decisions (moves) per agent per state. Then, $m = \mathbf{O}(nd^k)$. Therefore the* $\mathbf{ATL}_{IR}$

model checking algorithm from [14] runs in time $\mathbf{O}(nd^k l)$*, and hence its complexity is exponential if the number of agents is a parameter of the problem.*

In comparison, for an *unlabeled* transition system with n states and m transitions, we have that $m = \mathbf{O}(n^2)$. This means that **CTL** model checking is in P also with respect to the number of states in the model and the length of the formula. The following theorem is an immediate corollary of the fact (and Theorem 5.2).

Theorem 5.16. CTL *model checking over unlabeled transition systems is P-complete in the number of states and the length of the formula, and can be done in time* $\mathbf{O}(n^2 l)$.

For **ATL** and concurrent game structures, however, the situation is different. In the following we make precise what we mean by a compressed transition function.

Implicit concurrent game structures (called this way first in [278], but already present in the ISPL modeling language behind MCMAS [352, 353]) are defined similarly to a CGS but the transition function is encoded in a particular way often allowing for a more compact representation than the explicit transition table. Formally, an *implicit* CGS is given by $\mathfrak{M} = \langle \mathbb{A}\mathrm{gt}, St, \Pi, \pi, Act, d, \hat{o} \rangle$ where $\hat{o}$, the *encoded transition function*, is given by a sequence

$$((\varphi_0^r, q_0^r), \ldots, (\varphi_{t_r}^r, q_{t_r}^r))_{r=1,\ldots,|Q|}$$

where $t_r \in \mathbb{N}_0$, $q_i^r \in St$ and each φ_i^r is a Boolean combination of propositions exec_α^j where $j \in \mathbb{A}\mathrm{gt}$, $\alpha \in Act$, $i = 1, \ldots, t$ and $r = 1, \ldots, |Q|$. It is required that $\varphi_{t_r}^r = \top$. The term exec_α^j stands for "agent j executes action α". We use $\varphi[\alpha_1, \ldots, \alpha_k]$ to refer to the Boolean formula over $\{\top, \bot\}$ obtained by replacing $\mathsf{exec}_\alpha^{a_j}$ with $\top$ (resp. $\bot$) if $\alpha_j = \alpha$ (resp. $\alpha_j \neq \alpha$). The encoded transition function induces a standard transition function $o_{\hat{o}}$ as follows:

$$o_{\hat{o}}(q_i, \alpha_1, \ldots, \alpha_k) = q_j^i \text{ where } j = \min\{\kappa \mid \varphi_\kappa^i[\alpha_1, \ldots, \alpha_k] \equiv \top\}$$

That is, $o_{\hat{o}}(q_i, \alpha_1, \ldots, \alpha_k)$ returns the state belonging to the formula φ_κ^i (associated with state q_i) with the minimal index κ that evaluates to "true" given the actions $\alpha_1, \ldots, \alpha_k$. We use $\hat{o}(q_i, \alpha_1, \ldots, \alpha_k)$ to refer to $o_{\hat{o}}(q_i, \alpha_1, \ldots, \alpha_k)$. Note that the function is well defined as the last formula in each sequence is given by $\top$: no deadlock can occur. The size of $\hat{o}$ is defined as $|\hat{o}| = \sum_{r=1,\ldots,|Q|} \sum_{j=1,\ldots,t_r} |\varphi_j^r|$, that is, the sum of the sizes of all formulae. Hence, the size of an implicit CGS is given by $|St| + |\mathbb{A}\mathrm{gt}| + |\hat{o}|$. Recall, that the size of an explicit CGS is $|St| + |\mathbb{A}\mathrm{gt}| + m$ where m is the number of transitions. Finally, we require that the encoding of the transition function is reasonably compact, that is, $|\hat{o}| \leq \mathbf{O}(|o_{\hat{o}}|)$.

Now, why should the model checking complexity change for implicit CGS's? Firstly, one can observe that we can take the trivial encoding of an explicit transition function yielding an implicit CGS that has the same size as the explicit CGS. This implies that all the *lower bounds* proven before are still valid.

Proposition 5.1. *Model checking with respect to implicit* CGS*'s is at least as hard as model checking over explicit* CGS*'s for any logic discussed here.*

Therefore, we focus on the question whether model checking can become more difficult for implicit CGS's. Unfortunately, the answer is *yes*: Model checking can indeed become more difficult.

We illustrate this by considering the presented algorithm for solving the $\mathbf{ATL}_{IR}$ model checking problem. It traverses all transitions and since transitions are considered explicitly in the input, the algorithm runs in polynomial time. But if we choose an encoding $\hat{o}$ that is significantly smaller than the explicit number of transitions, the algorithm still has to check all transitions, yet now the number of transitions can be *exponential* with respect to the input of size $|St| + |\mathbb{A}\mathrm{gt}| + |\hat{o}|$.

Henceforth, we are interested in the cases in which the size of the encoded transition function is much smaller, in particular, when the size of the encoding is polynomial with respect to the *number of states and agents*. This is the reason why we will often write that we measure the input in terms of states (n) and agents (k), neglecting the size of $\hat{o}$ when it is supposed to be polynomial in n, k.

Remark 5.5. An alternative view is to assume that the transition function is provided by an external procedure (a "black box") that runs in polynomial time, similar to an oracle [257]. This view comes along with some technical disadvantages, and we will not discuss it here.

5.4.1 *Model Checking* ATL *and* CL *in Terms of States and Agents*

As argued above the complexity of $\mathbf{O}(ml)$ may (but does not have to) include potential intractability if the transition function is represented more succinctly. The following result supports this observation.

Theorem 5.17 ([257, 259, 279]). *Model checking* $\mathbf{ATL}_{IR}$ *and* $\mathbf{ATL}_{Ir}$ *over implicit* CGS*'s is* Δ_3^P*-complete with respect to the size of the model and the length of the formula (l).*

Proof (Sketch). The idea of the proof for the lower bound is clear if we reformulate the model checking of $M, q \models \langle\langle a_1, \ldots, a_r \rangle\rangle \quad \varphi$ as

$$\exists(\alpha_1, \ldots, \alpha_r) \forall(\alpha_{r+1}, \ldots, \alpha_k)\ M, o(q, \alpha_1, \ldots, \alpha_k) \models \varphi,$$

which closely resembles QSAT_2, a typical Σ_2^P-complete problem. A reduction of this problem to our model checking problem is straightforward: For each instance of QSAT_2, we create a model where the values of propositional variables $p_1, \ldots, p_r$ are "declared" by agents A and the values of $p_{r+1}, \ldots, p_k$ by $\mathbb{A}\mathrm{gt} \setminus A$. The subsequent transition leads to a state labeled by proposition yes iff the given Boolean

formula holds for the underlying valuation of $p_1, \ldots, p_k$. Then, QSAT_2 reduces to model checking formula $\langle\langle a_1, \ldots, a_r \rangle\rangle$ yes [257]. In order to obtain Δ_3^P-hardness, the above schema is combined with nested cooperation modalities, which yields a rather technical reduction of the SNSAT_3 problem that can be found in [279].

For the upper bound, we consider the following algorithm for checking $\mathfrak{M}, q \models \langle\langle A \rangle\rangle \gamma$ with no nested cooperation modalities. Firstly, guess a strategy s_A of the proponents and fix A's actions to the ones described by s_A. Then check if $\mathsf{A}\gamma$ is true in state q of the resulting model by asking an oracle about the existence of a counterstrategy $s_{\bar{A}}$ for $\mathbb{A}\mathrm{gt} \setminus A$ that falsifies γ and reverting the oracle's answer. The evaluation takes place by calculating $\hat{o}$ (which takes polynomially many steps) regarding the actions prescribed by $(s_A, s_{\bar{A}})$ at most $|St|$ times. For nested cooperation modalities, we proceed recursively (bottom-up). □

Surprisingly, the imperfect information variant of **ATL** is no harder than the perfect information one under this measure:

Theorem 5.18 ([259]). *Model checking* $\mathbf{ATL}_{ir}$ *over implicit* CGS*'s is* Δ_3^P*-complete with respect to the size of the model and the length of the formula. This is the same complexity as for model checking* $\mathbf{ATL}_{Ir}$ *and* $\mathbf{ATL}_{IR}$.

Proof (Sketch). For the upper bound, we use the same algorithm as in checking $\mathbf{ATL}_{Ir}$. For the lower bound, we observe that $\mathbf{ATL}_{Ir}$ can be embedded in $\mathbf{ATL}_{ir}$ by explicitly assuming perfect information of agents (through the minimal reflexive indistinguishability relations). □

The Δ_3^P-hardness proof in Theorem 5.17 uses the "nexttime" and "until" temporal operators in the construction of an **ATL** formula that simulates SNSAT_3 [279]. However, the proof can be modified so that only the "nexttime" sublanguage of $\mathcal{L}_{ATL}$ is used. We obtain thus an analogous result for coalition logic. Details of the new construction can be found in the technical report [91].

Theorem 5.19. *Model checking* $\mathbf{CL}_{IR}$, $\mathbf{CL}_{Ir}$, $\mathbf{CL}_{ir}$, *and* $\mathbf{CL}_{iR}$ *over implicit* CGS*'s is* Δ_3^P*-complete with respect to the size of the model and the length of the formula. Moreover, it is* Σ_2^P*-complete for the "simple" variants of* **CL**.

It is worth mentioning that model checking "Positive **ATL**" (i.e., the fragment of $\mathcal{L}_{ATL}$ where negation is allowed only on the level of literals) is Σ_2^P-complete with respect to the size of implicit CGS's, and the length of formulae for the *IR*, *Ir*, and *ir*-semantics [259]. The same applies to "Positive **CL**", the analogous variant of coalition logic.

5.4.2 CTL *and* CTL$^+$ *Revisited*

At the beginning of Section 5.4, we mentioned that the complexity of model checking computation tree logic is still polynomial even if we measure the size of models with the number of states rather than transitions. That is certainly true for unlabeled transition systems (i.e., the original models of **CTL**). For concurrent game structures, however, this is no longer the case.

Theorem 5.20. *Model checking* **CTL** *over implicit* CGS*'s is* Δ_2^P*-complete with respect to the size of the model and the length of the formula.*

Proof (Sketch). For the upper bound, we observe that $\mathfrak{M}, q \models^{\mathbf{CTL}} \mathsf{E}\gamma$ iff $\mathfrak{M}, q \models_{IR} \langle\langle \mathbb{A}\mathrm{gt} \rangle\rangle \gamma$ which is in turn equivalent to $\mathfrak{M}, q \models_{Ir} \langle\langle \mathbb{A}\mathrm{gt} \rangle\rangle \gamma$. In other words, $\mathsf{E}\gamma$ holds iff the grand coalition has a *memoryless* strategy to achieve γ. Thus, we can verify $\mathfrak{M}, q \models \mathsf{E}\gamma$ (with no nested path quantifiers) as follows: we guess a strategy $s_{\mathbb{A}\mathrm{gt}}$ for $\mathbb{A}\mathrm{gt}$ (in polynomially many steps), then we construct the resulting model $\mathfrak{M}'$ by asking $\hat{o}$ which transitions are enabled by following the strategy s_A and check if $\mathfrak{M}', q \models \mathsf{E}\gamma$ and return the answer. Note that $\mathfrak{M}'$ is an *unlabeled transition system*, so constructing $\mathfrak{M}'$ and checking $\mathfrak{M}', q \models \mathsf{E}\gamma$ can be done in polynomial time. For nested modalities, we proceed recursively.

For the lower bound, we sketch the reduction of SAT to model checking $\mathcal{L}_{CTL}$-formulae with only one path quantifier. For propositional variables $p_1, \ldots, p_k$ and boolean formula φ, we construct an implicit CGS where the values of $p_1, \ldots, p_k$ are "declared" by agents $\mathbb{A}\mathrm{gt} = \{a_1, \ldots, a_k\}$ (in parallel). The subsequent transition leads to a state labeled by proposition yes iff φ holds for the underlying valuation of $p_1, \ldots, p_k$. Then, SAT reduces to model checking formula $\langle\langle \mathbb{A}\mathrm{gt} \rangle\rangle \quad \mathsf{yes}$. The reduction of SNSAT$_2$ (to model checking $\mathcal{L}_{CTL}$-formulae with nested path quantifiers) is an extension of the SAT reduction, analogous to the one in [258, 259]. □

As it turns out, the complexity of **CTL**$^+$ does not increase when we change the models to implicit concurrent game structures: It is still Δ_2^P.

Theorem 5.21. *Model checking* **CTL**$^+$ *over implicit* CGS*'s is* Δ_2^P*-complete with respect to the size of the model and the length of the formula.*

Proof (Sketch). The lower bound follows from Theorem 5.5 and Proposition 5.1.

For the upper bound, we observe that the **CTL**$^+$ model checking algorithm in [280] verifies $\mathfrak{M}, q \models \mathsf{E}\gamma$ by guessing a finite history h with length $|St_M| \cdot |\gamma|$, and then checking γ on h. We recall that $\mathsf{E}\gamma \equiv \langle\langle \mathbb{A}\mathrm{gt} \rangle\rangle \gamma$. Thus, for a concurrent game structure, each transition in h can be determined by guessing an action profile in $O(|\mathbb{A}\mathrm{gt}|)$ steps, calculating $\hat{o}$ wrt the guessed profile, and the final verification whether γ holds on the *finite* sequence h which can be done in deterministic polynomial time (cf. [92]). Consequently, we can implement this procedure by a nondeterministic Turing machine that runs in polynomial time. For nested path quantifiers, we proceed recursively which shows that the model checking problem can be solved by a polynomial time Turing machine with calls to an NP-oracle. □

We will use the last result in the analysis of **ATL**$^+$ in Section 5.4.3.

5.4.3 ATL* and ATL+

Theorem 5.22. *Model checking* $\mathbf{ATL}^*_{Ir}$ *and* $\mathbf{ATL}^*_{ir}$ *over implicit* CGS*'s is PSPACE-complete with respect to the size of the model and the length of the formula.*

Proof. The lower bound follows from Theorem 5.13 and Proposition 5.1.

For the upper bound, we model-check $\mathfrak{M}, q \models \langle\langle A \rangle\rangle \gamma$ by guessing a memoryless strategy s_A for coalition A, then we guess a counterstrategy $s_{\bar{A}}$ of the opponents. Having a complete strategy profile, we proceed as in the proof of Theorem 5.20 and check the **LTL** path formula γ on the resulting (polynomial model) $\mathfrak{M}'$ which can be done in polynomial space (Theorem 5.13). For nested cooperation modalities, we proceed recursively. □

Theorem 5.23 ([279]). *Model checking* $\mathbf{ATL}^*_{IR}$ *over implicit* CGS*'s is 2EXPTIME-complete with respect to the size of the model and the length of the formula.*

Proof. The lower bound follows from Theorem 5.12 and Proposition 5.1. For the upper bound, we have to modify the algorithm given in the proof of Theorem 5.12 such that it is capable of dealing with implicit models. More precisely, we need to modify the construction of the Büchi automaton $\mathcal{A}_{\mathfrak{M},q,A}$ that is used to accept the (q,A)-execution trees. Before, we simply checked all the moves of A in polynomial time and calculated the set of states A is effective for (as the moves are bounded by the number of transitions). Here, we have to incrementally generate all these moves from A using $\hat{o}$. This may take exponential time (as there can be exponentially many moves in terms of the number of states and agents). However, as this can be done independently of the non-emptiness check, the overall runtime of the algorithm is still double exponential. □

Theorem 5.24 ([279]). *Model checking* $\mathbf{ATL}^+_{Ir}$*, and* $\mathbf{ATL}^+_{ir}$ *over implicit* CGS*'s is* Δ^P_3*-complete with respect to the size of the model and the length of the formula.*

Proof. The lower bounds follow from Theorem 5.14 and Proposition 5.1. For the upper bound we model-check $\mathfrak{M}, q \models \langle\langle A \rangle\rangle \gamma$ by guessing a memoryless strategy s_A for coalition A, and constructing an *unlabeled transition system* $\mathfrak{M}'$ as follows. For each state q_i we evaluate formulae contained in $((\varphi^i_0, q^i_0), \ldots, (\varphi^i_{t_i}, q^i_{t_i}))$ according to the guessed strategy. Then, we introduce a transition from q_i to q^i_j if $(\bigwedge_{k=0,\ldots,j-1} \neg\varphi^i_k) \wedge \varphi^i_j$ is satisfiable (i.e., there is a countermove of the opponents such that φ^j_i is true and j is the minimal index) . This is the case iff the opponents have a strategy to enforce the next state to be q^i_j. These polynomially many tests can be done by *independent* calls of an NP-oracle. The resulting model $\mathfrak{M}'$ is an explicit CGS of polynomial size regarding the number of states and agents. Finally, we apply $\mathbf{CTL}^+$ model checking to $\mathsf{A}\gamma$ which can be done in time Δ^P_2. □

	Ir	*IR*	*ir*	*iR*
Simple $\mathcal{L}_{CL}$	Σ_2^P	Σ_2^P	Σ_2^P	Σ_2^P
$\mathcal{L}_{CL}$	Δ_3^P	Δ_3^P	Δ_3^P	Δ_3^P
$\mathcal{L}_{ATL}$	Δ_3^P	Δ_3^P	Δ_3^P	Undecidable†
$\mathcal{L}_{ATL^+}$	Δ_3^P	*PSPACE*	Δ_3^P	Undecidable†
$\mathcal{L}_{ATL^*}$	*PSPACE*	*2EXPTIME*	*PSPACE*	Undecidable†

Fig. 5.7 Overview of the model checking complexity results for implicit CGS. All results are completeness results. Each cell represents the logic over the language given in the row using the semantics given in the column. † These problems are believed to be undecidable, though no formal proof has been proposed yet.

Finally, we consider the case for perfect recall strategies. The lower and upper bound directly follow from the proof of Theorem 5.15.

Theorem 5.25 ([92]). *Model checking* $\mathbf{ATL}^+_{IR}$ *over implicit* CGS*'s is PSPACE-complete with respect to the size of the model and the length of the formula.*

A summary of complexity results for the alternative representation/measure of the input is presented in Figure 5.7. It turns out that, when considering the finer-grained representation that comes along with a measure based on the number of states, agents, and an encoded transition function rather than just the number of transitions, the complexity of model checking $\mathcal{L}_{ATL}$ seems distinctly harder than before for games with perfect information, and only somewhat harder for imperfect information. In particular, the problem falls into the same complexity classes for imperfect and perfect information analysis, which is rather surprising, considering the results from Section 5.3. Finally, the change of perspective does not influence the complexity of model checking of $\mathcal{L}_{ATL^*}$ as well as $\mathcal{L}_{ATL^+}$ at all.

5.5 Higher-Order Representations of Models

In this section, we summarize very briefly the results for higher-level representations of multi-agent systems (e.g., concurrent programs, reactive modules, modular interpreted systems etc.). Sections 5.3 and 5.4 presented complexity results for model checking with respect to models where global states of the system were represented *explicitly*. Most multi-agent systems, however, are characterized by an immensely huge state space. In such cases, one would like to define the model in terms of a *compact high-level representation*, plus an unfolding procedure that defines the relationship between representations and actual models of the logic (and hence also the semantics of the logic with respect to the compact representation). Of course, unfolding a higher-level description to an explicit model involves usually an exponential blowup in its size.

Consider, for example, a system whose state space is defined by r boolean variables (binary attributes). Obviously, the number of global states in the system is $n = 2^r$. A more general approach is presented in [275], where the "high-level description" is defined in terms of *concurrent programs*, that can be used for simulating Boolean variables, but also for processes or agents acting in parallel.

A concurrent program P is composed of k concurrent processes, each described by a labeled transition system $P_i = \langle St_i, Act_i, \mathcal{R}_i, \Pi_i, \pi_i \rangle$, where St_i is the set of local states of process i, Act_i is the set of local actions, $\mathcal{R}_i \subseteq St_i \times Act_i \times St_i$ is a transition relation, and Π_i, π_i are the set of local propositions and their valuation. The behavior of program P is given by the product automaton of $P_1, \ldots, P_k$ under the assumption that processes work asynchronously, actions are interleaved, and synchronization is obtained through common action names.

Theorem 5.26 ([275]). *Model checking* **CTL** *in concurrent programs is PSPACE-complete with respect to the number of local states and agents (processes), and the length of the formula.*

Concurrent programs seem to be sufficient to reason about purely temporal properties of systems, but not quite so for reasoning about agents' strategies and abilities. For the latter kind of analysis, we need to allow for more sophisticated interference between agents' actions (and enable modeling agents that play synchronously). **ATL** model checking for higher-order representations was first analyzed in [231] over a class of *simple reactive modules*, based on synchronous product of local models. However, even simple reactive modules do not allow to model interference between agents' actions. Because of that, we use *modular interpreted systems* [254,256], that draw inspiration from interpreted systems [169], reactive modules [12], and are in many respects similar to ISPL specifications [352].

A modular interpreted system (MIS) is defined as a tuple $\mathfrak{M} = \langle \mathbb{A}\mathrm{gt}, env, Act, \mathcal{I}n \rangle$, where $\mathbb{A}\mathrm{gt} = \{a_1, \ldots, a_k\}$ is a set of agents, env is the environment, Act is a set of actions, and $\mathcal{I}n$ is a set of symbols called *interaction alphabet*. Each agent has the following internal structure: $a_i = \langle St_i, d_i, out_i, in_i, o_i, \Pi_i, \pi_i \rangle$, where:

- St_i is a set of local states,
- $d_i : St_i \to \mathcal{P}(Act)$ defines local availability of actions; for convenience we additionally define the set of *situated actions* as $D_i = \{\langle q_i, \alpha \rangle \mid q_i \in St_i, \alpha \in d_i(q_i)\}$,
- out_i, in_i are *interaction functions*; $out_i : D_i \to \mathcal{I}n$ refers to the influence that a given situated action (of agent a_i) may possibly have on the external world, and $in_i : St_i \times \mathcal{I}n^k \to \mathcal{I}n$ translates external manifestations of the other agents (and the environment) into the "impression" that they make on a_i's transition function depending on the local state of a_i,
- $o_i : D_i \times \mathcal{I}n \to St_i$ is a (deterministic) local transition function,
- Π_i is a set of local propositions of agent a_i where we require that Π_i and Π_j are disjunct when $i \neq j$, and

- $\pi_i : \Pi_i \to \mathcal{P}(St_i)$ is a valuation of these propositions.

The environment *env* has the same structure as an agent except that it does not perform actions.

The unfolding of a MIS $\mathfrak{M}$ to a concurrent game structure follows by the synchronous product of the agents (and the environment) in $\mathfrak{M}$, with interaction symbols being passed between local transition functions at every step. The unfolding can also determine indistinguishability relations as follows $\langle q_1, \ldots, q_k, q_{env}\rangle \sim_i \langle q'_1, \ldots, q'_k, q'_{env}\rangle$ iff $q_i = q'_i$, thus yielding a full *i*CGS. This way the semantics of both $\mathbf{ATL}_{\boldsymbol{IR}}$/$\mathbf{ATL}_{\boldsymbol{Ir}}$ and $\mathbf{ATL}_{\boldsymbol{ir}}$ is extended to MIS.

Theorem 5.27 ([231]). *Model checking* $\mathbf{ATL}_{\boldsymbol{Ir}}$ *and* $\mathbf{ATL}_{\boldsymbol{IR}}$ *in simple reactive modules is EXPTIME-complete with respect to the number of local states and agents, and the length of the formula.*

Since simple reactive modules can be embedded in modular interpreted systems, and the model checking algorithm from [231] can be extended to MIS, we get the following.

Theorem 5.28 ([254]). *Model checking* $\mathbf{ATL}_{\boldsymbol{Ir}}$ *and* $\mathbf{ATL}_{\boldsymbol{IR}}$ *in modular interpreted systems is EXPTIME-complete with respect to the number of local states and agents, and the length of the formula.*

Note that this means that systems with no interference between agents are *not easier to handle* than the general case.

The real surprise, however, comes to light when we study the model checking complexity for imperfect information agents.

Theorem 5.29 ([254, 256]). *Model checking* $\mathbf{ATL}_{\boldsymbol{ir}}$ *in modular interpreted systems is PSPACE-complete with respect to the number of local states and agents, and the length of the formula.*

Thus, model checking in modular interpreted systems seems to be easier for imperfect rather than perfect information strategies (while it appears to be distinctly harder for explicit models, cf. Section 5.3). There are two reasons for that. The more immediate is that agents with limited information have *fewer* available strategies than if they had perfect information about the current (global) state of the game. Generally, the difference is exponential in the number of agents. More precisely, the number of perfect information strategies is *double exponential* with respect to the number of agents and their local states, while there are "only" exponentially many uniform strategies – and that settles the results in favor of imperfect information.

The other reason is more methodological. While model checking imperfect information is easier when we are given a particular MIS, modular interpreted systems may provide more compact representation to systems where all the agents have perfect information by definition. In particular, the most compact MIS representation of a

given ICGS $\mathfrak{M}$ can be exponentially larger than the most compact MIS representation of $\mathfrak{M}$ with the epistemic relations removed. In the former case, the MIS must encode the epistemic relations explicitly. In the latter case, the epistemic aspect is ignored, which gives some extra room for encoding the transition relation more efficiently.

On the other hand, it should be noted that for systems of agents with "reasonably imperfect information", i.e., ones where the number of each agent's local states is logarithmic in the number of global states of the system, the optimal MIS encodings for perfect and imperfect information are the same. Still, model checking $\mathbf{ATL}_{\boldsymbol{IR}}$ is *EXPTIME*-complete and model checking $\mathbf{ATL}_{\boldsymbol{ir}}$ is *PSPACE*-complete, which suggests that imperfect information can be beneficial in practical verification.

Finally, we report two results that are straightforward extensions of Theorem 5.19 and Theorem 5.29, respectively.

Theorem 5.30. *Model checking* $\mathbf{CL}_{\boldsymbol{IR}}$, $\mathbf{CL}_{\boldsymbol{Ir}}$, $\mathbf{CL}_{\boldsymbol{ir}}$, *and* $\mathbf{CL}_{\boldsymbol{iR}}$ *is* Δ_3^P*-complete with respect to the number of local states and agents in the* MIS *and the length of the formula. Moreover, it is* Σ_2^P*-complete for the "simple" variants of* **CL**.

Theorem 5.31. *Model checking* $\mathbf{ATL}^{+}_{\boldsymbol{ir}}$ *and* $\mathbf{ATL}^{*}_{\boldsymbol{ir}}$ *in modular interpreted systems is PSPACE-complete with respect to the number of local states and agents, and the length of the formula.*

5.6 Summary

Figure 5.8 gives a summary of the results. The results for $\mathbf{ATL}_{\boldsymbol{Ir}}$ and $\mathbf{ATL}_{\boldsymbol{ir}}$ form an intriguing pattern. When we compare model checking agents with *perfect vs. imperfect* information, the first problem appears to be much easier against explicit models measured with the number of transitions. Then, we get the same complexity class against explicit models measured with the number of states and agents. Finally, model checking imperfect information turns out to be *easier* than model checking perfect information for modular interpreted systems. Why is that so?

The *number of available strategies* (relative to the size of input parameters) is the crucial factor here. It is *exponential in the number of global states*. For uniform strategies, there are usually much less of them but still exponentially many in general. Thus, the fact that perfect information strategies can be synthesized incrementally has a substantial impact on the complexity of the problem. However, measured in terms of local states and agents, the number of all strategies is *double exponential*, while there are "only" exponentially many uniform strategies– which settles the results in favor of imperfect information. It should be also noted that the *representation* of a concurrent game structure by a MIS can be in general more compact than that of an *i*CGS. In the latter case, the MIS is assumed to encode the epistemic relations explicitly. In the case of CGS, the epistemic aspect is ignored, which gives some extra room for encoding the transition relation more efficiently.

Logic \ Input	m,l	n,k,l	n_{local},k,l
Simple $\mathbf{CL}_{iR,ir,IR,Ir}$	AC^0 [279]	Σ_2^P-complete	Σ_2^P-complete
$\mathbf{CL}_{iR,ir,IR,Ir}$	P-complete	Δ_3^P-complete	Δ_3^P-complete
$\mathbf{ATL}_{Ir,IR}$	P-complete [14]	Δ_3^P-compl. [259,279]	*EXPTIME*-compl. [231]
$\mathbf{ATL}_{ir}$	Δ_2^P-compl. [259,388]	Δ_3^P-compl. [259]	*PSPACE*-compl. [254]
$\mathbf{ATL}_{iR}$	Undecidable†	Undecidable†	Undecidable†
$\mathbf{ATL}^+_{Ir}$	Δ_3^P-complete [388]	Δ_3^P-complete	*EXPTIME*-hard
$\mathbf{ATL}^+_{ir}$	Δ_3^P-complete [388]	Δ_3^P-complete	*PSPACE*-complete
$\mathbf{ATL}^+_{IR}$	*PSPACE*-complete [92]	*PSPACE*-complete [92]	*EXPTIME*-hard
$\mathbf{ATL}^+_{iR}$	Undecidable†	Undecidable†	Undecidable†
$\mathbf{ATL}^*_{Ir}$	*PSPACE*-complete [388]	*PSPACE*-complete	*EXPTIME*-hard
$\mathbf{ATL}^*_{ir}$	*PSPACE*-compl. [388]	*PSPACE*-complete	*PSPACE*-complete
$\mathbf{ATL}^*_{IR}$	*2EXPTIME*-compl. [14]	*2EXPTIME*-compl.	*EXPTIME*-hard
$\mathbf{ATL}^*_{iR}$	Undecidable†	Undecidable†	Undecidable†

Fig. 5.8 Overview of the complexity results. All results except for "Simple **CL**" are completeness results. The results with no given reference have been established in this chapter for the first time (usually by a simple extension of existing proofs). The fields that report only hardness results correspond to problems which are still open. Symbols n and m stand for the number of states and transitions, respectively, and k is the number of agents in the model, l is the length of the formula, and n_{local} is the number of local states in a concurrent program, simple reactive module, or modular interpreted system. † These problems are believed to be undecidable, though no formal proof has been proposed yet (cf. Conjectures 5.1, 5.2, and 5.3).

What have we learned and what are the challenges ahead? An important outcome of theoretical research on verification is to determine the precise boundary between model checking problems that are decidable and those that are not. As we have shown, decidability depends very much on the underlying language, the chosen logic and whether we consider perfect or imperfect recall. But even if the problem is decidable, the precise complexity of the problem depends on the chosen representation and ranges from *P*- to *2EXPTIME*-completeness.

It must be noted that Figure 5.8 is filled mostly with complexity classes that are generally considered intractable. Of these, the undecidability hypotheses for $\mathbf{ATL}_{iR}$ are obviously the most pessimistic. But what does an undecidability result tell us? It shows that there is no general algorithm solving the problem at hand. Yet one is often not interested in model checking *all* possible specifications that can be expressed in the underlying logic. For most practical purposes, the set of interesting formulas to be model checked is quite small. This raises the question: *Which subsets of the logics are decidable?* A similar question can be stated for the complexity results reported here: *2EXPTIME* completeness concerns all formulas of $\mathcal{L}_{ATL^*}$, but suitable fragments can have much lower complexity. These are interesting questions to be investigated in the future.

Chapter 6

Correctness of Multi-Agent Programs: A Hybrid Approach

M. Dastani and J.-J. Ch. Meyer

Abstract This chapter proposes a twofold approach for ensuring the correctness of BDI-based agent programs. On the one hand, we advocate the alignment of the semantics of agent programming languages with agent specification languages such that for an agent programming language it can be shown that it obeys specific desirable properties expressed in the corresponding agent specification language. In this way, one can guarantee that specific properties expressed in the specification language are satisfied by any program implemented in the programming language. On the other hand, we introduce a debugging framework to find and resolve possible defects in such agent programs. The debugging approach consists of a specification language and a set of debugging tools. The specification language allows a developer to express cognitive and temporal properties of multi-agent program executions. The debugging tools allow a developer to verify if a specific multi-agent program execution satisfies a desirable property.

M. Dastani, J.-J. Ch. Meyer
Utrecht University, The Netherlands e-mail: {mehdi,jj}@cs.uu.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_6,

6.1 Introduction

A promising approach to develop computer programs for complex and concurrent applications are multi-agent systems. In order to implement multi-agent systems, various agent-oriented programming languages and development tools have been proposed [64, 65]. These agent-oriented programming languages facilitate the implementation of individual agents and their interactions. A special class of these programming languages aims at programming *BDI-based* multi-agent systems, i.e., multi-agent systems in which individual agents are programmed in terms of cognitive concepts such as beliefs, events, goals, plans, and reasoning rules [63, 122, 224, 343, 433].

There is a whole range of approaches to the correctness of multi-agent programs. While some of them are abstract verification frameworks for agent programs [60, 372], a majority of these approaches fall under the heading of either model checking [21, 61, 371] or theorem proving [6–8] methods. This chapter concerns the correctness of *BDI-based* multi-agent programs by proposing two techniques that are complementary to those of model-checking and theorem proving. These two approaches are somewhat dual to each other, and are the following: (1) we show how to prove properties that are *general* in the sense that they hold for *any* execution of *any* program written in the agent programming language at hand, and (2) we show how to verify the correctness of *specific* execution of *specific* program written in the agent programming language at hand by proposing a *debugging* approach with which it is possible to check whether run-time executions of programs are still in line with the specification, so still according to certain desirable properties (what we may call a 'so far so good?' approach of verification). Moreover, as we will see, both approaches are particularly targeted at programs written in a BDI-style programming language.

So let us first turn to the issue of proving certain properties that do not depend on the particular program but rather on the agent programming language and its interpretation. To this end we define the semantics of the programming language as well as an agent specification language such that we can express these general properties and prove them with respect to the semantics of the programming language. We will establish this such that the semantics of the specification language is aligned with that of the programming language in a systematic and natural way! We show that an agent programming language obeys some desirable properties expressed in an agent specification language, i.e., that any individual agent implemented by the programming language satisfies the desirable property expressed in the specification language. Note that this is an important issue raised in the literature of agent program correctness. As many other approaches to the correctness of agents are based on rather abstract agent (BDI-like) logics [110, 360, 385], it is not always clear how the abstract BDI notions appearing in these logics, often treated by means of modal operators, relate to notions rooted in actual computation. This is referred to by Wooldridge as the problem of *ungrounded semantics* for agent specification languages [439].

There is a number of proposals in the literature to ground agent logics in the actual computation of agents (e.g., [130, 219]). In these approaches, it is attempted to ground agent logics by rendering the notions that occur in these logics such as beliefs and goals less abstract and more computational, so that reasoning about these notions becomes relevant for reasoning about agent programs. In the current paper, we follow the line of [130] by connecting the semantics of agent logics directly to the operational semantics of agent programming languages. More precisely, the modal accessibility relations in Kripke models of an agent logic are associated with the (operational) semantics of the programming language at hand. Furthermore, it is important to realize that this problem is different from the model checking problem. In model checking, the problem is to verify if a certain property, expressed in a specification language such as the language of a BDI logic, holds for *all* executions of one *specific* agent program (and not for every agent program written in the same programming language). Thus, in contrast to model checking, where one verifies that all executions of a particular agent program satisfy a property expressed in a specification language, we are here interested only in certain general properties of individual agents such as such as commitment strategies, e.g., *an agent will not drop its goals until it believes it has achieved them*.

Secondly in this chapter we propose a debugging approach to check the correctness of a *specific* execution of a *specific* agent program. Debugging is the art of finding and resolving errors or possible defects in a computer program. Here, we will focus on the semantic bugs in BDI-based multi-agent programs and propose a generic approach for debugging such programs [63, 84, 114, 121, 122, 343, 345, 346, 425]. In particular, we propose a specification language to express execution properties of multi-agent programs and a set of debugging actions/tools to verify if a specific program execution satisfies a specific property. The expressions of the specification language are related to the proposed debugging actions/tools by means of special programming constructs. Using these constructs in a multi-agent program ensures that the debugging actions/tools are performed/activated as soon as their associated properties hold for the multi-agent program execution at hand. In order to illustrate how a developer can debug both *cognitive* and *temporal* aspects of the (finite) executions of multi-agent programs we discuss a number of examples. They show how the debugging constructs allow a developer to log specific parts of the cognitive state of individual agent programs (e.g., log the beliefs, events, goals, or plans) from the moment that specific condition holds, stop the execution of multi-agent programs whenever a specific cognitive condition holds, or check whether an execution trace of a multi-agent program (a finite sequence of cognitive states) satisfies a specific (cognitive/temporal) property.

Interestingly, if we compare the two approaches in this chapter we will see that although we use temporal logic for both, there are notable differences. The two most important ones being: (1) In the 'general' program-independent approach we use branching-time temporal logic, since we have to reason about behaviour in general, i.e. all possible executions of all programs, including explicit representations of agents' choices, while in the debugging approach we use linear-time temporal logic, since we are only interested in a particular execution, viz. the execution at hand, of

one particular program. (2) In the 'general' approach we use temporal models with infinite branches/paths, as usual in the temporal reasoning about programs, while in the debugging approach we use temporal models with only finite paths, reflecting the fact that we are looking at run-time (mostly unfinished) executions.

The structure of this chapter is as follows. First we present a simple but extendable BDI-based agent-oriented programming language that provides constructs to implement concepts such as beliefs, goals, plans, and reasoning rules. The syntax and semantics of this programming language is presented in section 6.2. In the rest of the chapter we focus on the correctness of programs of this programming language. We start by treating the 'general' approach of verifying program-independent (but semantics-dependent) properties, i.e., properties of the semantics of the programming language, and not of any specific program. To this end, we present in section 6.3 an agent specification language which is closely related to well-known BDI specification languages. This specification language consists of (modal) operators for beliefs, goals, and time such that properties such as commitment strategies can be expressed. In section 6.4 we show that all agents that are implemented in the proposed agent programming language satisfy the desirable properties that are specified in the specification language. Then we turn to the 'specific' approach of verifying execution-dependent properties by presenting a debugging approach for the same simple programming language. We first define a temporal specification language to express execution properties that we aim to verify. Special attention is given to the (non-standard) semantics of the specification language since its formula are evaluated on (finite) execution paths of multi-agent programs. Next we present our proposal of a tool set for debugging multi-agent programs. Finally, we discuss some related works on debugging multi-agent programs and conclude the chapter.

6.2 An agent-oriented Programming Language *APL*

In this section, we propose the syntax and operational semantics of a simple but prototypical logic-based agent-oriented programming language that provides programming constructs to implement agents in terms of cognitive concepts such as beliefs, goals, and plans. In order to focus on the relation between such a programming language and its related specification language and without loss of generality, we ignore some aspects that may be relevant or even necessary for the practicality and effectiveness of the programming language. The presented programming language is thus not meant to be practically motivated, but rather to illustrate how such a logic- and BDI-based programming language can be connected to logical specification languages and how to examine the correctness of the related programs. This aim is accomplished by defining the syntax and semantics of the programming language similar to those of the existing practical agent-oriented programming languages (e.g., 2APL, 3APL, GOAL, and Jason [64, 122, 129, 224]). This makes our approach applicable to the existing logic- and BDI-based agent programming lan-

guages. A concrete extension of this agent-oriented programming language is studied in [130].

6.2.1 Syntax of *APL*

A multi-agent program comprises a set of individual agent programs, each of which is implemented in terms of concepts such as beliefs, goals, and plans. In this section, we focus on a programming language for implementing individual agents and assume that a multi-agent program is a set of programs, each of which implements an individual agent. The presented agent-oriented programming language provides programming constructs for beliefs, goals, plans, and planning rules. We use a propositional language to represent an agent's beliefs and goals while plans are assumed to consists of actions that can update the agent's beliefs base. It is important to note that these simplifications do not limit the applicability of the proposed model.

The idea of an agent's belief is to represent the agent's information about the current state of affairs. The idea of an (achievement) goal is to reach a state that satisfies it. An agent is then expected to generate and execute plans to achieve its goal. The emphasis here is that the goal will not be dropped until a state is reached that satisfies it. An example of an achievement goal is to have fuel in car (`fuel`) for which the agent can generate either a plan to fuel at gas station 1 (`gs1`) or a plan to fuel at gas station 2 (`gs2`). The achievement goal will be dropped as soon as the agent believes that it has fuel in its car, i.e., as soon as it believes `fuel`. In our running example, a car driving agent believes he is in position 1 (`pos1`) and has the goal to fuel (`fuel`).

Definition 6.1. (belief and goal languages) Let L_p be a propositional language. The belief language is denoted by $L_\sigma \subseteq L_p$ and the goal language is denoted by $L_\gamma \subseteq L_p$.

For the purpose of this chapter, we assume a set of plans Plan each of which is executed atomically, i.e., a plan can be executed in one computation step. Moreover, agents are assumed to generate their plans based on their beliefs and goals. The planning rules indicate which plans are appropriate to be selected when the agent has a certain goal and certain beliefs. A planning rule is of the form $\beta, \kappa \Rightarrow \pi$ and represents the possibility to select plan π for the goal κ, if the agent believes β. In order to be able to check whether an agent has a certain belief or goal, we use propositional formulas from L_p to represent belief and goal query expressions.

Definition 6.2. (planning rule) Let Plan be the set of plans that an agent can use. The set of planning rules $\mathcal{R}_{\mathsf{PL}}$ is defined as:

$$\mathcal{R}_{\mathsf{PL}} = \{\beta, \kappa \Rightarrow \pi \mid \beta \in L_\sigma, \kappa \in L_\gamma, \pi \in \mathsf{Plan}\}$$

In the rest of the paper, **Goal**(r) and **Bel**(r) are used to indicate, respectively, the goal condition κ and the belief condition β of the planning rule $r = (\beta, \kappa \Rightarrow \pi)$. In the running example, the agent has two planning rules `pos1,fuel⇒gs1` and `pos2,fuel⇒gs2`. The first (second) planning rule indicates that the agent should fuel at gas station 1 (2) if he wants to fuel and believes that he is in position `pos1` (`pos2`). Given these languages, an agent can be implemented by programming two sets of propositional formulas (representing the agent's beliefs and goals), and one set of planning rules. For the purpose of this chapter, we assume that agents cannot have initial plans, but generate them during their execution.

Definition 6.3. (agent program) Let Id be the set of agent names. An individual agent program is a tuple $(\iota, \sigma, \gamma, \mathsf{PL})$, where $\iota \in Id$, $\sigma \subseteq L_\sigma, \gamma \subseteq L_\gamma$ and $\mathsf{PL} \subseteq \mathcal{R}_{\mathsf{PL}}$. A multi-agent program is a set of such tuples, i.e., $\{(\iota, \sigma, \gamma, \mathsf{PL}) \mid \iota \in Id, \sigma \subseteq L_\sigma, \gamma \subseteq L_\gamma, \mathsf{PL} \subseteq \mathcal{R}_{\mathsf{PL}}\}$.

The individual agent program for our running example is the tuple $(c, \sigma, \gamma, \mathsf{PL})$, where c is the name of the car agent, $\sigma = \{$`pos1`$\}$, $\gamma = \{$`fuel`$\}$, and $\mathsf{PL} = \{$`pos1, fuel⇒gs1, pos2,fuel⇒gs2`$\}$. Note that the agent believes it is in position 1 and has the goal to fuel.

6.2.2 Semantics of APL

The operational semantics of the multi-agent programming language is presented in terms of a transition system. A transition system is a set of derivation rules for deriving transitions. A transition is a transformation of one state into another and it corresponds to a single computation step. For the semantics of the multi-agent programming language a transition is a transformation of one multi-agent configuration (state) into another. The operational semantics of the multi-agent programming language is directly defined in terms of the operational semantics of individual agent programming language.

6.2.2.1 Agent Configuration

An agent's configuration denotes the state of the agent at one moment in time. It is determined by its mental attitudes, i.e., by its beliefs, goals, plans, and reasoning rules. A multi-agent configuration denotes the state of all agents at one moment in time.

Definition 6.4. (configuration) Let $\models_p$ be the classical propositional entailment relation (used also in the rest of the chapter). Let $\Sigma = \{\sigma \mid \sigma \subseteq L_\sigma, \sigma \not\models_p \bot\}$ be the set of possible consistent belief bases and $\Gamma = \{\phi \mid \phi \in L_\gamma, \phi \not\models_p \bot\}$ be the set of goals.

A configuration of an agent is a tuple $\langle \iota, \sigma, \gamma, \Pi, \mathsf{PL} \rangle$, where ι is the agent's identifier, $\sigma \in \Sigma$ is its belief base, $\gamma \subseteq \Gamma$ is its goal base, $\Pi \subseteq (L_\gamma \times L_\gamma \times \mathsf{Plan})$ is its plan base, and $\mathsf{PL} \subseteq \mathcal{R}_{\mathsf{PL}}$ includes its planning rules. The set of all agent configurations is denoted by $\mathcal{A}$. A multi-agent configuration is a subset of $\mathcal{A}$, i.e., a multi-agent configuration is a set of individual agent configurations.

In the sequel, we use $\langle \sigma_\iota, \gamma_\iota, \Pi_\iota, \mathsf{PL}_\iota \rangle$ instead of $\langle \iota, \sigma, \gamma, \Pi, \mathsf{PL} \rangle$ to keep the representation of agent configurations simple. In the above definition, it is assumed that the belief base of an agent is consistent since otherwise the agent can believe everything which is not a desirable property. Also, each goal is assumed to be consistent since otherwise an agent should achieve an impossible state. Finally, the elements of the plan base are defined as 3-tuples $(L_p \times L_p \times \mathsf{Plan})$ consisting of a plan and two goals (propositional formulas) that indicate the reasons for generating the plan. More specifically, (ϕ, κ, π) is added to an agent's plan base if the planning rule $\beta, \kappa \Rightarrow \pi$ is applied because κ is a subgoal of the agent's goal ϕ. Note that the planning rule can be applied only if κ is a subgoal of an agent's goal ϕ. This means that we have $\forall (\phi, \kappa, \pi) \in \Pi : \ \phi \models_p \kappa$. This information will be used to avoid applying a planning rule if it is already applied and the generated plan is not fully executed. In our running example, the agent can apply both planning rules (depending on the agent's beliefs) since the goals of these rules (i.e., `fuel`) is a subgoal of the agent's goal `fuel`, i.e., since `fuel`$\models$`fuel`. Note that these rules could also be applied if the agent had more complex goals such as `fuel`$\wedge$`cw`; `cw` stands for car wash.

The initial configuration of an individual agent is based on the individual agent program (definition 6.3) that specifies the initial beliefs, goals, and planning rules. As noted, for the purpose of this chapter, we assume that an agent does not have initial plans, i.e., the initial plan base is empty. The initial multi-agent configuration is the set of initial configurations of individual agents.

Definition 6.5. (initial configuration) Let $\{(1, \sigma_1, \gamma_1, \mathsf{PL}_1), \ldots, (n, \sigma_n, \gamma_n, \mathsf{PL}_n)\}$ be a multi-agent program. Then, the initial configuration of the multi-agent program is $\{\langle \sigma_1, \gamma_1, \Pi_1, \mathsf{PL}_1 \rangle, \ldots, \langle \sigma_n, \gamma_n, \Pi_n, \mathsf{PL}_n \rangle\}$, where $\Pi_i = \emptyset$ for $i = 1, \ldots, n$.

In the following, we assume that all agents use one and the same set of planning rules PL (i.e., all agents use the same planning rule library) and that the set of planning rules does not change during the agent executions. For this reason, we do not include the set PL in the individual agent configurations and use $\langle \sigma, \gamma, \Pi \rangle$ instead of $\langle \sigma, \gamma, \Pi, \mathsf{PL} \rangle$. This means that an *APL* program specifies only the initial beliefs and goals of an agent.

6.2.2.2 Transition System $\mathcal{T}$

This subsection presents the transition system $\mathcal{T}$ which consists of transition rules (also called derivation rules) for deriving transitions between configurations. Each

transition rule has the following form indicating that the configuration C can be transformed to configuration C' if the condition of the rule holds.

$$\frac{condition}{C \rightarrow C'}$$

We first present three transition rules that transform the configurations of individual agent programs, followed by a transition rule that transform multi-agent configurations. The first three transition rules capture the successful execution of plans, the failed execution of plans, and the application of planning rules, respectively.

In order to define the transition rule for the application of planning rules, we define the notions of *relevant* and *applicable* planning rules w.r.t. an agent's goal and its configuration. Intuitively, a planning rule is relevant for an agent's goal if it can contribute to the agent's goal, i.e., if the goal that occurs in the head of the planning rule is a subgoal of the agent's goal. A planning rule is applicable to an agent's goal if it is relevant for that goal and its belief condition is entailed by the agent's configuration.

Definition 6.6. (relevant, applicable) Let $C = \langle \sigma, \gamma, \Pi, \mathsf{PL} \rangle$ be an agent configuration. Given configuration C containing goal $\phi \in \gamma$, the set of relevant and applicable planning rules are defined as follows:
- $rel(\phi, C) = \{r \in \mathsf{PL} \mid \phi \models_p \mathbf{Goal}(r)\}$
- $app(\phi, C) = \{r \in rel(\phi, C) \mid \sigma \models_p \mathbf{Bel}(r)\}$

In the following transition rules we write $\frac{app(\phi)}{C \rightarrow C'}$ instead of $\frac{app(\phi, C)}{C \rightarrow C'}$.

When executing an agent, planning rules will be selected and applied based on its beliefs, goals and plans. The application of planning rules generates plans which can subsequently be selected and executed. Before introducing the transition rules to specify possible agent execution steps, we need to define what it means to execute a plan. The execution of a plan affects the belief and goal bases. The effect of plan execution on the belief base is captured by an update operator `update`, which takes the belief base and a plan and generates the updated belief base. This update operator can be as simple as adding/deleting atoms to/from the belief base. We assume a partial function $\texttt{update} : (\mathsf{Plan} \times \Sigma) \rightarrow \Sigma$ that takes a plan and a belief base, and yields the belief base resulting from the execution of the plan on the input belief base (if the update is not successful, the update operation is undefined).

The first transition rule (R_1) captures the case where the plan π is successfully executed. The resulting configuration contains a belief base that is updated based on the executed plan, a goal base from which achieved goals are removed, and a plan base from which plans with associated achieved goal are removed.

Rule R_1 (Plan execution 1)

$$\frac{(\phi, \kappa, \pi) \in \Pi \ \&\ \texttt{update}(\sigma, \pi) = \sigma'}{\langle \sigma, \gamma, \Pi \rangle \rightarrow \langle \sigma', \gamma', \Pi' \rangle}$$

where
$\gamma' = \gamma \setminus \{\psi \mid \sigma' \models_p \psi\}$ and $\Pi' = \Pi \setminus (\{(\phi,\kappa,\pi)\} \cup \{(\phi',\kappa',\pi') \in \Pi \mid \sigma' \models_p \phi'\})$.

The second transition rule (R_2) captures the case that the performance of the plan has failed, i.e., the update operation is undefined. In this case, the failed plan (ϕ,ψ,π) will be removed from the plan base.

Rule R_2 (Plan execution 2)

$$\frac{(\phi,\kappa,\pi) \in \Pi \;\&\; \texttt{update}(\sigma,\pi) = undefined}{\langle \sigma,\gamma,\Pi\rangle \rightarrow \langle \sigma,\gamma,\Pi \setminus \{(\phi,\kappa,\pi)\}\rangle}$$

Plans should be generated to reach the state denoted by goals. If the generated and performed plans do not achieve the desired state, then the corresponding goal remains in the goal base. The first transition rule below (called R_3) is designed to apply planning rules in order to generate plans the execution of which may achieve the subgoals of the goals. A planning rule can be applied if the goal in the head of the rule is not achieved yet, if there is no plan for the same subgoal in the plan base (in order to avoid applying rules if it is already applied), and if the subgoal is not achieved yet. The application of a planning rule will add the plan of the planning rule to the plan base.

Rule R_3 (apply planning rules)

$$\frac{\phi \in \gamma \;\&\; (\beta,\kappa \Rightarrow \pi) \in app(\phi) \;\&\; \nexists \pi' \in \mathsf{Plan} : (\phi,\kappa,\pi') \in \Pi \;\&\; \sigma \not\models_p \kappa}{\langle \sigma,\gamma,\Pi\rangle \rightarrow \langle \sigma,\gamma,\Pi \cup \{(\phi,\kappa,\pi)\}\rangle}$$

We consider an execution of a multi-agent program as an interleaved execution of the involved individual agent programs. The following transition rule captures the parallel execution

Rule R_4 (multi-agent execution)

$$\frac{A_i \rightarrow A_i'}{\{A_1,\ldots,A_i,\ldots,A_n\} \rightarrow \{A_1,\ldots,A_i',\ldots,A_n\}}$$

In this and next sections, we focus on the semantics of individual agent programs as the semantics of multi-agent programs is a composition of the semantics of individual agent programs.

6.2.2.3 Agent Execution

In order to define all possible behaviours of an agent program and compare them with each other, we need to define what it means to execute an agent program. Given

a transition system consisting of a set of transition rules, the execution of an agent program is a set of transitions generated by applying the transition rules to the initial configuration of the program (i.e., initial beliefs and goals). Thus, the execution of an agent program starts with its initial configuration and generates subsequent configurations that can be reached from the initial configuration by applying transition rules. The execution of an agent program forms a graph in which the nodes are the configurations and the edges indicate the application of a transition rule (i.e., execution of a plan, or the application of a planning rule). In the following, we define the execution of an agent program A by first defining the set of all possible transitions $\mathcal{R}_{\mathcal{T}}$ for all possible agents given a transition system $\mathcal{T}$, and then take the subset of those transitions that can be reached from the initial configuration of agent A.

Definition 6.7. (agent execution) Recall that $\mathcal{A}$ be the set of all agent configurations. Then, the set of transitions that are derivable from a transition system $\mathcal{T}$, denoted as $\mathcal{R}_{\mathcal{T}}$, is defined as follows:

$$\mathcal{R}_{\mathcal{T}} = \{(c_i, c_j) \mid c_i \rightarrow c_j \text{ is a transition derivable from } \mathcal{T} \ \& \ c_i, c_j \in \mathcal{A}\}$$

Given an agent program A with corresponding initial configuration c_0, the execution of A is the smallest set $\mathcal{E}_{\mathcal{T}}(A)$ of transitions derivable from $\mathcal{T}$ starting from c_0, i.e., it is the smallest subset $\mathcal{E}_{\mathcal{T}}(A) \subseteq \mathcal{R}_{\mathcal{T}}$ such that:

- if $(c_0, c_1) \in \mathcal{R}_{\mathcal{T}}$, then $(c_0, c_1) \in \mathcal{E}_{\mathcal{T}}(A)$, for $c_1 \in \mathcal{A}$
- if $(c_i, c_j) \in \mathcal{E}_{\mathcal{T}}(A)$ and $(c_j, c_k) \in \mathcal{R}_{\mathcal{T}}$, then $(c_j, c_k) \in \mathcal{E}_{\mathcal{T}}(A)$, for $c_i, c_j, c_k \in \mathcal{A}$

6.3 CTL_{apl}: A Specification Language for Agent Programs

In the area of agent theory and agent-oriented software systems, various logics have been proposed to characterize the behavior of rational agents. The most cited logics to specify agents' behavior are the BDI logics [110, 315, 360, 385]. These logics are multi-modal logics consisting of temporal and epistemic modal operators. In the BDI logics, the behaviour of an agent is specified in terms of the temporal evolution of its mental attitudes (i.e., beliefs, desires, and intentions) and their interactions. These logics are characterized by means of axioms and inference rules to capture the desired static and dynamic properties of agents' behaviour. In particular, the axioms establish the desired properties of the epistemic and temporal operators as well as the rational balance between them. For example, the axioms to capture the desired static properties of beliefs are $KD45$ (the standard weak $S5$ system), for desires and intentions are KD, and for the rational balance between beliefs and desires are various versions of *realism*. Moreover, some desired dynamic properties of agents' behaviour are captured through axioms that implement various versions of the *commitment strategies*. These axioms are defined using temporal operators expressing when and under which conditions the goals and intentions of agents can

be dropped. For example, an agent can be specified to either hold its goals until it has achieved it (blindly-committed agent type), or drop the goal if it believes that it can *never* achieve it (single-minded agent type) [110, 360].

A main concern in designing and developing agent-oriented programming languages is to provide programming constructs in such a way that their executions generate the agent behaviours having the same desirable properties as in their specifications. This implies that the semantics of the programming languages should be defined in such a way to satisfy the desirable properties captured by means of the axioms in the BDI logics. The main issue addressed in this part of the chapter is how agent specification logics, which are used to specify the agents' behaviour, can be related to agent-oriented programming languages, which are used to implement agents. We study this relation by proposing an instantiation of the BDI_{CTL} logic [360, 385] with a declarative semantics that is aligned with the operational semantics of the programming language *APL* as proposed in section 6.2. We then show that this alignment enables us to prove that certain properties expressed in the specification language are satisfied by the programming language. The specification language is a multi-modal logic consisting of temporal modal operators to specify the evolution of agents' configurations (the agents' execution) through time and epistemic modal operators to specify agents' mental state (beliefs and goals) in each configuration. In order to relate the specification and programming languages, we do not allow the nesting of epistemic operators. This is because the beliefs and goals in the agent programming language *APL*, presented in section 6.2, are propositional rather than modal formulas. This is, however, not a principle limitation as the representation of beliefs and goals in agent programming languages can be extended to modal formulas [437].

In the rest of this section and in section 6.4, we only consider the specification and properties of single agent programs. The proposed specification language can be extended for multi-agent programs in an obvious way because individual agent programs do not interact. The individual agent programs do not communicate or share an environment as their actions are limited to local belief and goal changes.

6.3.1 CTL_{apl} Syntax

The behaviour of an agent, generated by the execution of the agent, is a temporal structure over its mental states. In order to specify the mental state of agents, we will define the language L consisting of non-nested belief and goal formulas.

Definition 6.8 (specification language L). The language L for the specification of agents' mental attitudes consists of non-nested belief and goal formulas, defined as follows: if $\phi \in L_p$, then $B(\phi)$, $G(\phi) \in L$.

We then use the standard CTL^* logic [158] in which the primitive propositions are formulas from the language L. The resulting language will be called CTL_{apl} defined as follows.

Definition 6.9 (specification language CTL_{apl}). The state and path formulas are defined by the following S and P clauses, respectively.

- (S1) Each formula from L is a state formula.
- (S2) If ϕ and ψ are state formulas, then $\phi \wedge \psi$ and $\neg\phi$ are also state formulas.
- (S3) If ϕ is a path formula, then $E\phi$ and $A\phi$ are state formulas.
- (P1) Any state formula is a path formula.
- (P2) If ϕ and ψ are path formulas, then $\neg\phi$, $\phi \wedge \psi$, $X\phi$, $\diamond\phi$, and $\phi\ U\ \psi$ are path formulas.

Using the CTL_{apl} language, one can for example express that if an agent has a goal, then it will not drop the goal until it believes the goal is achieved, i.e., $G(\phi) \rightarrow A(G(\phi)\ U\ B(\phi))$.

6.3.2 CTL_{apl} Semantics

The semantics of the specification language CTL_{apl} is defined on a Kripke structure $M_{\mathcal{T}} = \langle C, R, V\rangle$, where the set of states C is the set of configurations of agents implemented in the agent programming language APL (definition 6.4), and the temporal relation R is specified by the transition system $\mathcal{T}$ of the agent programming language APL (definition 6.7). In particular, there exists a temporal relation between two configurations in the Kripke structure if and only if a transition between these two agent configurations is derivable from the transition system $\mathcal{T}$. Finally, the valuation function $V = (V_b, V_g)$ of the Kripke structure is defined on agent configurations and consists of different valuation functions each with respect to a specific mental attitude of agents' configurations. More specifically, we define a valuation function V_b that valuates the belief formulas in terms of agents' beliefs and a valuation function V_g that valuates the goal formulas in terms of the agents' goals. The belief valuation function V_b maps an agent configuration c to a set of propositions $V_b(c)$ that are derivable from the agent's belief base. An agent believes a proposition if and only if the proposition is included in $V_b(c)$. The valuation function V_g for goals is defined in such a way that all subgoals of an agent's goal are also considered as a goal. The valuation function V_g maps an agent's configuration to a set of sets of propositions. Each set contains all subgoals of a goal. An agent wants to achieve a proposition if and only if the proposition is included in a set. The semantics of the CTL_{apl} expressions are defined as follows.

Definition 6.10. Let $M_{\mathcal{T}} = \langle C, R, V\rangle$ be a Kripke structure specified by the execution of the transition system $\mathcal{T}$, where:

- C is a set of configurations (states) of the form $\langle \sigma, \gamma, \Pi\rangle$.

- $R \subseteq C \times C$ is a serial binary relation such that for each $(c, c') \in R$ we have $(c, c') \in \mathcal{R}_{\mathcal{T}}$ or $c = c'$.
- $V = (V_b, V_g)$ are the belief and goal evaluation functions, i.e.,
  - $V_b : C \rightarrow 2^{L_p}$ s.t. $V_b(\langle \sigma, \gamma, \Pi \rangle) = \{\phi \mid \sigma \models_p \phi\}$,
  - $V_g : C \rightarrow 2^{2^{L_p}}$ s.t. $V_g(\langle \sigma, \gamma, \Pi \rangle) = \{\{\phi' \mid \phi \models_p \phi'\} \mid \phi \in \gamma\}$.

A fullpath is an infinite sequence $x = c_0, c_1, c_2, \ldots$ of configurations such that $\forall i : (c_i, c_{i+1}) \in R$. We use x^i to indicate the i-th state of the path x.

- (S1) $M_{\mathcal{T}}, c \models B(\phi) \Leftrightarrow \phi \in V_b(c)$
- (S1) $M_{\mathcal{T}}, c \models G(\phi) \Leftrightarrow \exists s \in V_g(c) : \phi \in s$
- (S2) $M_{\mathcal{T}}, c \models \phi \wedge \psi \Leftrightarrow M_{\mathcal{T}}, c \models \phi$ and $M_{\mathcal{T}}, c \models \psi$
- (S2) $M_{\mathcal{T}}, c \models \neg\phi \Leftrightarrow M_{\mathcal{T}}, c \not\models \phi$
- (S3) $M_{\mathcal{T}}, c \models E\phi \Leftrightarrow \exists$ fullpath $x = c, c_1, c_2, \ldots \in M_{\mathcal{T}} : M_{\mathcal{T}}, x \models \phi$
- (S3) $M_{\mathcal{T}}, c \models A\phi \Leftrightarrow \forall$ fullpath $x = c, c_1, c_2, \ldots \in M_{\mathcal{T}} : M_{\mathcal{T}}, x \models \phi$
- (P1) $M_{\mathcal{T}}, x \models \phi \Leftrightarrow M_{\mathcal{T}}, x^0 \models \phi$ for ϕ is a state formula
- (P2) $M_{\mathcal{T}}, x \models X\phi \Leftrightarrow M_{\mathcal{T}}, x^1 \models \phi$
- (P2) $M_{\mathcal{T}}, x \models \diamond\phi \Leftrightarrow M_{\mathcal{T}}, x^n \models \phi$ for some $n \geq 0$
- (P2) $M_{\mathcal{T}}, x \models \phi \; U \; \psi \Leftrightarrow$
  a) $\exists k \geq 0$ such that $M_{\mathcal{T}}, x^k \models \psi$ and for all $0 \leq j < k : \; M_{\mathcal{T}}, x^j \models \phi$, or,
  b) $\forall j \geq 0 : \; M_{\mathcal{T}}, x^j \models \phi$

Note that the two options in the last clause capture two interpretations of the until operator. The first (strong) interpretation is captured by the option a and requires that the condition ψ of the until expression should hold at once. The second (weak) interpretation is captured by the option b and requires the formula ϕ can hold forever.

In the above definition, the CTL_{apl} state formulas are evaluated in the Kripke model $M_{\mathcal{T}}$ with respect to an arbitrary configuration c consisting of beliefs, goals, and plans. In the following, we model the execution of a particular agent program A (i.e., the execution of an agent with the initial configuration A[1]) based on the transition system $\mathcal{T}$ as the Kripke model $M^A_{\mathcal{T}} = \langle C^A_{\mathcal{T}}, R^A_{\mathcal{T}}, V \rangle$ on which the CTL_{apl} expressions (i.e., properties to be checked) can be evaluated. The accessibility relation $R^A_{\mathcal{T}}$ is defined as the set of executions (based on transition system $\mathcal{T}$) of the agent program A (i.e., traces that can be generated by applying planning rules and executing plans starting at the configuration specified by A) extended with a reflexive accessibility for all end configurations. This is to guarantee the seriality property of the accessibility relation $R^A_{\mathcal{T}}$. Moreover, the set $C^A_{\mathcal{T}}$ of configurations will be defined in terms of configurations that occur in the execution of the agent A.

[1] An agent's initial configuration is determined by the corresponding agent program which specifies the initial beliefs and goals. It is assumed that there are no initial plans.

Definition 6.11. (agent model) Let A be an agent program and let $\mathcal{E}_{\mathcal{T}}(A)$ be the execution of A. Then the model corresponding with agent program A, which we call an *agent model*, is defined as $M^A_{\mathcal{T}} = \langle C^A_{\mathcal{T}}, R^A_{\mathcal{T}}, V\rangle$, where the accessibility relation $R^A_{\mathcal{T}}$ and the set of configurations $C^A_{\mathcal{T}}$ are defined as follows:

$$R^A_{\mathcal{T}} = \mathcal{E}_{\mathcal{T}}(A) \cup \{(c_n, c_n) \mid \exists (c_{n-1}, c_n) \in \mathcal{E}_{\mathcal{T}}(A) \& \neg \exists (c_n, c_{n+1}) \in \mathcal{E}_{\mathcal{T}}(A)\}$$
$$C^A_{\mathcal{T}} = \{c \mid (c, c') \in R^A_{\mathcal{T}}\}$$

Note that agent models are Kripke structures in the sense of Definition 6.10.

As we are interested in expressing that a certain property holds for all executions of a particular agent program A, we will define the notion of satisfaction in an agent model.

Definition 6.12. (satisfaction in model) A formula ϕ is satisfied in the model $M^A_{\mathcal{T}} = \langle C^A_{\mathcal{T}}, R^A_{\mathcal{T}}, V\rangle$ if and only if ϕ holds in $M^A_{\mathcal{T}}$ with respect to all configurations $c \in C^A_{\mathcal{T}}$, i.e.,

$$M^A_{\mathcal{T}} \models \phi \quad \Leftrightarrow_{def} \quad \forall c \in C^A_{\mathcal{T}} \ : \ M^A_{\mathcal{T}}, c \models \phi$$

In section 6.4, we prove that certain properties hold for any agent program that is implemented in the *APL* programming language. As the above definition of model $M^A_{\mathcal{T}}$ is based on one specific agent program A, we need to quantify over all agent programs. Since the binary relation $R^A_{\mathcal{T}}$ (derived from the transition system $\mathcal{T}$, which is the semantics of the agent programs) has to be the same in all Kripke models, a quantification over agent programs means a quantification over models $M^A_{\mathcal{T}}$. This implies that we need to define the notion of validity of a property as being true for all agent programs and thus for all models $M^A_{\mathcal{T}}$.

Definition 6.13. (validity) A property $\phi \in CTL_{apl}$ holds for the execution of an arbitrary agent A based on the transition system $\mathcal{T}$, expressed as $\models_{\mathcal{T}}$, if and only if ϕ holds in all agent models $M^A_{\mathcal{T}}$, i.e.,

$$\models_{\mathcal{T}} \phi \quad \Leftrightarrow_{def} \quad \forall A \ : \ M^A_{\mathcal{T}} \models \phi$$

Note that this notion of validity is the same as the notion of validity in modal logic since it is defined at the level of frames, i.e., at the level of states and relation and not valuations, which is in our case defined in terms of specific agents.

Finally, we would like to explain our motivation for choosing a variant of *CTL* instead of other formalisms such as for example the linear time temporal logic *LTL*. Such a choice may not be trivial as one might argue that linear time temporal logic would be enough to specify and verify an agent's behavior. The idea would be to consider the execution behavior of the corresponding agent program as a set of linear traces. However, our consideration to use a variant of *CTL* is based on the fact that agents have choices (e.g., to select and execute plans from their plan library) and that these choices are essential characteristic of their autonomy. The computational

tree logic CTL with its branching time structure enables the specification and verification of such choices. In order to illustrate the characterising difference between CTL and LTL that is essential for capturing an agent's choices, consider the two execution models illustrated in Figure 6.1. While these two execution models include the same set of linear traces, the execution models A and B differ in the choices available to the agent. This is reflected by the fact that the CTL formula $AXE\Box p$ is true in state $s0$ of model B, while this is not the case for state $s0$ of model A. In other words, we have $A, s0 \not\models AXE\Box p$ and $B, s0 \models AXE\Box p$. In the next section, we will present an agent property which is related to the agent's choices. This property justifies the choice for using a variant of CTL for our specification and verification of agent programs.

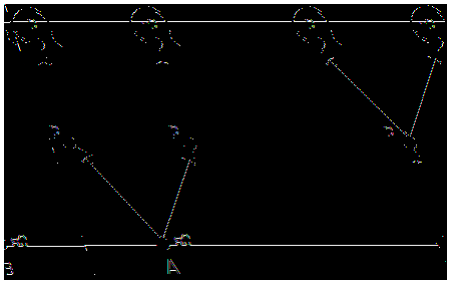

Fig. 6.1 Two execution models with two different choice moments.

6.4 Properties

Given the semantics of the programming language APL and the specification language CTL_{apl}, we can prove that certain properties expressed in CTL_{apl} hold for agents programmed in APL. Other properties for an extension of the APL language are provided in [130].

6.4.1 Proving the Properties

In this section, we present a number of desired properties and prove that they hold for arbitrary agent programs implemented in the APL language.

First, since the accessibility relation $R_A^{\mathcal{T}}$ of $M_{\mathcal{T}}^A$ is based on the transition system $\mathcal{T}$, we present some properties of the accessibility relation with respects to specific subsets of the transition system. In particular, the following proposition shows

persistence of unachieved goals through transitions that are derived based on transition rule R_1. Note that this transition rule modifies an agent's beliefs and remove achieved goals.

Proposition 6.1. *If $c_{i-1} \rightarrow c_i$ is a transition derived based on transition rule $R_1 \in \mathcal{T}$, $M^A_{\mathcal{T}}, c_{i-1} \models G(\phi)$, and $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, then $M^A_{\mathcal{T}}, c_i \models G(\phi)$.*

Proof. Following Definition 6.10 and using notation $sub(\psi) = \{\psi' \mid \psi \models_p \psi'\}$, we have $V_g(c_i) = \{sub(\psi) \mid \psi \in \gamma_i\}$ where γ_i is the goal based of configuration c_i. Following the definition of transition rule $R_1 \in \mathcal{T}$ for determining γ_i, we have $\{sub(\psi) \mid \psi \in \gamma_i\} = \{sub(\psi) \mid \psi \in \gamma_{i-1} \setminus \{\psi' \mid \sigma_i \models_p \psi'\}\}$, where σ_i is the belief base of configuration c_i. Using Definition 6.10 again, we have $\{sub(\psi) \mid \psi \in \gamma_{i-1} \setminus \{\psi' \mid \sigma_i \models_p \psi'\}\} = \{sub(\psi) \mid \psi \in \gamma_{i-1} \setminus V_b(c_i)\} = \{sub(\psi) \mid \psi \in \gamma_{i-1}\} \setminus \{sub(\psi) \mid \psi \in V_b(c_i)\} = V_g(c_{i-1}) \setminus \{sub(\psi) \mid \psi \in V_b(c_i)\}$. Suppose now that $sub(\phi) \in V_g(c_{i-1})$ and $\phi \notin V_b(c_i)$. Then, from the above equations we have $sub(\phi) \in V_g(c_i)$, which proves the proposition.

We can now generalize this proposition by showing the persistence of unachieved goals through all transitions.

Proposition 6.2. *If $c_{i-1} \rightarrow c_i$ is a transition, $M^A_{\mathcal{T}}, c_{i-1} \models G(\phi)$, and $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, then $M^A_{\mathcal{T}}, c_i \models G(\phi)$.*

Proof. This is the direct consequence of the following facts: 1) an agent's goals persist through transitions that are derived based on transition rules R_2 and R_3 as these transition rules do not modify the agent's goals, 2) an agent's goals persists through reflexive transitions, and 3) unachieved goals of an agent persist through transitions derived based on transition R_1 (Proposition 6.1).

The next property satisfied by the programs implemented in *APL* language is a variant of what has been termed "blind commitment" in [360], and what are called "persistent goals" in [110]. This property expresses that the execution of an *APL* agent program should not drop a goal before it is believed to be achieved.

Proposition 6.3. *(blind commitment)* $\models_{\mathcal{T}} G(\phi) \rightarrow A(G(\phi)\ U\ B(\phi))$

Proof. Using Definitions 6.13 and 6.12, we have to to prove that for any agent models $M^A_{\mathcal{T}}$ and all its configurations c:
if $M^A_{\mathcal{T}}, c \models G(\phi)$ then $M^A_{\mathcal{T}}, c \models A(G(\phi)\ U\ B(\phi))$.
Using Definition 6.10, we have to prove:
if $M^A_{\mathcal{T}}, c \models G(\phi)$ then $\forall$ fullpath $x = c, c', c'', \ldots \in M^A_{\mathcal{T}} : M^A_{\mathcal{T}}, x \models G(\phi)\ U\ B(\phi)$.
We prove that for arbitrary $M^A_{\mathcal{T}}$ and configuration c_0, it holds:
if $M^A_{\mathcal{T}}, c_0 \models G(\phi)$ then $\forall$ fullpath $x = c_0, c', c'', \ldots \in M^A_{\mathcal{T}} : M^A_{\mathcal{T}}, x \models G(\phi)\ U\ B(\phi)$.
Assume $M^A_{\mathcal{T}}, c_0 \models G(\phi)$ and take an arbitrary path $c_0, c_1, c_2, \ldots$ starting with c_0. We have to prove that $M^A_{\mathcal{T}}, c_0, c_1, c_2, \ldots \models G(\phi)\ U\ B(\phi)$. Following definition 6.10, we

have to prove the following:
a) $\exists k \geq 0$ such that $M, c_k \models B(\phi)$ and for all $0 \leq j < k : \ M, c_j \models G(\phi)$, or
b) $\forall j \geq 0 : \ M, c_j \models G(\phi)$
For the path $c_0, c_1, c_2, \ldots$, we distinguish two cases. Either for all consecutive states c_{i-1} and c_i in the path it holds that the transition $c_{i-1} \rightarrow c_i$ is such that $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, or there exists consecutive states c_{k-1} and c_k such that transition $c_{k-1} \rightarrow c_k$ is the first with $M^A_{\mathcal{T}}, c_k \models B(\phi)$. The first case (formulation of clause b above) is proven by induction as follows:
(Basic case) Given $M^A_{\mathcal{T}}, c_0 \models G(\phi)$ (assumption), $M^A_{\mathcal{T}}, c_1 \not\models B(\phi)$, Proposition 6.2 guarantees that $M^A_{\mathcal{T}}, c_1 \models G(\phi)$.
(Inductive case) Let $M^A_{\mathcal{T}}, c_{i-1} \models G(\phi)$. Using Proposition 6.2 together with the fact $M^A_{\mathcal{T}}, c_i \not\models B(\phi)$, we have $M^A_{\mathcal{T}}, c_i \models G(\phi)$.
The second case (formulation of clause a above) is proven as follows. As $c_{k-1} \rightarrow c_{k-1}$ is the first transition such that $M^A_{\mathcal{T}}, c_k \models B(\phi)$, we have $\forall 0 < i < k \ : \ M^A_{\mathcal{T}}, c_i \not\models B(\phi)$. Since $M^A_{\mathcal{T}}, c_0 \models G(\phi)$, we can apply Proposition 6.2 to all transitions $c_0 \rightarrow c_1 \ , \ldots, \ c_{k-2} \rightarrow c_{k-1}$ to show that $M^A_{\mathcal{T}}, c_{k-1} \models G(\phi)$. This is exactly the formulation of clause a above.

It should be noted that blind commitment in [360] is defined for intentions, rather than goals. Goals are also present in their framework, but are contrasted with intentions in that the agent is not necessarily committed to achieving its goals (but is committed in some way to achieving its intentions).

We now proceed to give a definition of intention, and show how intentions defined in this way are related to an agent's goals. We define that an agent intends κ, if κ follows from the second component of one of the plans in an agent's plan base. The second component of a plan specifies the subgoal for which the plan was selected, and it is these subgoals for which the agent is executing the plans, that we define to form the agent's intentions. This is analogous to the way the semantics of intention is defined in [77].

Definition 6.14. (intention) Let $V_i : C \rightarrow 2^{2^{L_p}}$ be defined as $V_i(\langle \sigma, \gamma, \Pi \rangle) = \{\{\kappa' \mid \kappa \models_p \kappa'\} \mid (\phi, \kappa, \pi) \in \Pi\}$. Then $M, c \models I(\kappa)$ is defined as $\exists s \in V_i(c) : \kappa \in s$.

Given this definition, we prove that an agent's intentions are a "subset" of the agent's goals, i.e., we prove the following proposition.

Proposition 6.4. *(intentions)*

$$\models_{\mathcal{T}} I(\kappa) \rightarrow G(\kappa)$$

Proof. The proof is based on induction by showing that for arbitrary agent model $M^A_{\mathcal{T}}$ and initial state c_0 the proposition holds for the initial state (basic case), and if it holds for a state of the model, then it holds for the next state of the model as well.

(Basic case) $M^A_{\mathcal{T}}, c_0 \models I(\kappa) \rightarrow G(\kappa)$ for the initial state c_0. Since agents are assumed to have no plans initially, we have $\nexists s \in V_i(c_0) : k \in s$. Using the definition of $I(\kappa)$ we conclude that $M^A_{\mathcal{T}}, c_0 \not\models I(\kappa)$ and thus $M^A_{\mathcal{T}}, c_0 \models I(\kappa) \rightarrow G(\kappa)$.

(Inductive case) Suppose $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$ and there is a transition $c \rightarrow c'$. We show that $M^A_{\mathcal{T}}, c' \models I(\kappa) \rightarrow G(\kappa)$. We distinguish two cases for the assumption $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$: 1) $M^A_{\mathcal{T}}, c \models I(\kappa)$, and 2) $M^A_{\mathcal{T}}, c \not\models I(\kappa)$.
(Case 1) Suppose $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$, $M^A_{\mathcal{T}}, c \models I(\kappa)$, and thus $M^A_{\mathcal{T}}, c \models G(\kappa)$. We show $M^A_{\mathcal{T}}, c' \models \neg G(\kappa) \rightarrow \neg I(\kappa)$ (contraposition), i.e., we show if $M^A_{\mathcal{T}}, c' \not\models G(\kappa)$ then $M^A_{\mathcal{T}}, c' \not\models I(\kappa)$. By Definition 6.10, $M^A_{\mathcal{T}}, c \models G(\kappa) \Leftrightarrow \exists s \in V_g(c) : k \in s$. Given $M^A_{\mathcal{T}}, c \models G(\kappa)$ and $M^A_{\mathcal{T}}, c' \not\models G(\kappa) \Leftrightarrow \nexists s \in V_g(c') : \kappa \in s$, we conclude that the transition $c \rightarrow c'$ is derived based on transition rule R_1 (as this is the only transition rule that modifies the agent's goals) and therefore $\forall \psi, \pi : (\psi, \kappa, \pi) \notin \Pi_{c'}$. Note that if $\exists s \in V_g(c') : \psi \in s$ would be the case, then we should also have $k \in s$ (because $\psi \models \kappa$), which contradict the assumption $\nexists s \in V_g(c') : \kappa \in s$. Since $\forall \psi, \pi : (\psi, \kappa, \pi) \notin \Pi_{c'}$ we conclude $\nexists s \in V_i(c') : \kappa \in s$ and thus $M^A_{\mathcal{T}}, c' \not\models I(\kappa)$.
(case 2) Suppose $M^A_{\mathcal{T}}, c \models I(\kappa) \rightarrow G(\kappa)$, $M^A_{\mathcal{T}}, c \not\models I(\kappa)$, and thus $M^A_{\mathcal{T}}, c \not\models G(\kappa)$. We show then $M^A_{\mathcal{T}}, c' \models I(\kappa) \rightarrow G(\kappa)$, i.e., if $M^A_{\mathcal{T}}, c' \models I(\kappa)$, then $M^A_{\mathcal{T}}, c' \models G(\kappa)$. Assume $M^A_{\mathcal{T}}, c' \models I(\kappa)$. Given $M^A_{\mathcal{T}}, c \not\models I(\kappa)$ we can conclude that the transition $c \rightarrow c'$ is derived based on transition rule R_3 by applying a planning rule. $M^A_{\mathcal{T}}, c' \models I(\kappa) \Leftrightarrow \exists s \in V_i(c') : \kappa \in s \Leftrightarrow \exists (\phi, \kappa', \pi) \in \Pi_{c'} : \kappa' \models \kappa$ and $\phi \models \kappa'$. The fact that a planning rule is applied means that $M^A_{\mathcal{T}}, s \models G(\phi)$ and therefore $M^A_{\mathcal{T}}, s \models G(\kappa)$.

Intuitively, this property holds since "intentions" or plans are generated on the basis of goals such that a plan cannot be created without a corresponding goal. Moreover, if a goal is removed, its corresponding plans are also removed. Note that while the commitment strategies were defined for intentions in [360] and hold for goals in our framework, the property of the BDI logic that relates goals and intentions *does* map directly to goals and (what we have defined as) intentions in our framework. Note also that the opposite of Proposition 6.4 does not hold, as it can be the case that an agent has a goal for which it has not yet selected a plan.

Finally, we present a property related to the choices of an agent, implemented by an *APL* program. This property shows our motivation for choosing a variant of *CTL* as the specification language. The property states that an agent can choose to commit to one of its goals and generate an intention, if the agent has appropriate means.

Proposition 6.5. *(intention = choice + commitment) Assume an agent program A with a planning rule $\beta, \kappa \Rightarrow \pi$. Let $\mathcal{T}$ be the transition system generated by the transition rules $R_1, \ldots, R_3$ based on this agent program.*

$$\models_{\mathcal{T}} (B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)) \rightarrow EX\, I(\kappa)$$

Proof. We prove that for arbitrary $M^A_{\mathcal{T}}$ and configuration c_0, it holds: if $M^A_{\mathcal{T}}, c_0 \models B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)$, then $M^A_{\mathcal{T}}, c_0 \models EX\ I(\kappa)$. Following definition 6.10, we have to prove that if $M^A_{\mathcal{T}}, c_0 \models B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)$, then $\exists$ fullpath $x = c_0, c_1, c_2, \ldots \in M^A_{\mathcal{T}} : M^A_{\mathcal{T}}, x \models X\ I(\kappa)$. Assume $M^A_{\mathcal{T}}, c_0 \models B(\beta) \wedge G(\kappa) \wedge \neg I(\kappa)$. Then, following definitions 6.10 and 6.14, we have $\beta \in V_b(c_0)$, $\exists s \in V_g(c_0) : k \in s$, and $\forall s \in V_i(c_0) : \kappa \notin s$. Note that definition 6.14 ensures that $\forall \phi\ \nexists \pi' \in \mathsf{Plan} : (\phi, \kappa, \pi') \in \Pi_{c_0}$ (where Π_{c_0} is the plan base that corresponds to configuration c_0). This means that the transition rule R_3 is applicable in c_0, which in turn means that a transition $c_0 \rightarrow c_1$ is derivable in the transition system $\mathcal{T}$ such that $(\phi, \kappa, \pi) \in \Pi_{c_1}$ for some $\phi \in V_g(c_1)$. By definition 6.14, we conclude that $M, c_1 \models I(\kappa)$. This ensures the existing of a path $x = c_0, c_1, \ldots : M, x \models X\ I(\kappa)$.

6.5 Debugging Multi-Agent Programs

In previous sections, we showed how a BDI-based agent-oriented programming language can be related to a BDI specification language. The relation allows us to prove that certain *generic* properties expressed in the specification language hold for the agent programming language, and thus for all executions of all agent programs that are implemented using this agent programming language.

However, one may want to verify properties for a specific execution of a specific multi-agent program. Of course, model-checking and theorem proving are two verification approaches that can be used to check properties of specific programs. The problem with these verification approaches is that they are often less effective for complex and real application programs. In order to check properties of such complex programs, one may consider a debugging approach and check a specific execution of a specific program. Thus, in contrast to model checking and theorem proving that analyze all possible full execution traces of a program at once, the debugging approach analyzes one specific execution trace of a specific program. It is important to emphasize that model-checking and theorem proving can therefore be used to prove the correctness of programs, while debugging can only be used to find possible defects of programs (as displayed in particular runs).

In the following sections, we propose a debugging approach that can be used to check temporal and cognitive properties of specific BDI-based multi-agent programs, e.g., if two or more implemented agents[2] can have the same beliefs, whether the number of agents is suited for the environment (e.g. it is useless to have a dozen explorers on a small area, or many explorers when there is only one cleaner that cannot keep up with them.), whether the protocol is suited for the given task (e.g. there might be a lot of overhead because facts are not shared, and therefore, needlessly rediscovered), whether important beliefs are shared and adopted, or rejected, once

[2] In the following, we write 'agents' and 'implemented agents' interchangeably since we focus on programs that implement agents.

they are received. We may also want to check if unreliable sources of information are ignored, or whether the actions of one agent are rational to take based on the knowledge of other agents.

6.5.1 Debugging Modes

Ideally one would specify a *cognitive* and *temporal* property and use it in two different debugging modes. In one debugging mode, called *continuous mode*, one may want to execute a multi-agent program and get notified when the specified property evaluates to true *during its execution*. In the second debugging mode, called *post mortem*, one may want to execute a multi-agent program, stop it after some execution steps, and check if the specified property evaluates to true for the performed execution. For both debugging modes, the specified properties are evaluated in the *initial state* of the multi-agent program *execution trace generated thusfar*. For the post mortem debugging mode, the generated execution trace thusfar is the trace generated from the start of the program execution until the execution is stopped. However, for the continuous debugging mode, the specified properties are evaluated after each execution step and with respect to the execution trace generated thusfar, i.e., the execution trace generated from the start of the program execution until the last execution step. This is because during a program execution a trace is modified and extended after each execution step. It should be noted that subsequent execution steps generate new program states and therefore new traces.

In the continuous debugging mode, the evaluation of a specified property *during* the execution of a multi-agent program means a continuous evaluation of the property on its evolving execution trace as it develops by consecutive execution steps. This continuous evaluation of the property can be used to halt the program execution as soon as a trace is generated which satisfies the property. It is important to know that properties are evaluated in the initial state of the execution trace so that the trace properties should be specified as temporal properties. A developer of multi-agent programs is assumed to know these aspects of our debugging framework in order to debug such programs effectively. Similar ideas are proposed in Jadex [343].

In the following, we introduce a specification language, called MDL (multi-agent description language), to specify the *cognitive* and *temporal* behavior (i.e., execution traces) of the BDI-based multi-agent programs. The MDL description language is taken to be a variant of LTL (Linear Temporal Logic) because execution traces of multi-agent programs, which are used to debug[3] such programs, are assumed to be linear traces. Note that this assumption is realistic as the interpreter of most (multi-agent) programs performs one execution step at a time and thereby generates a linear trace. An MDL expression is evaluated on the (finite) execution trace of a

[3] In contrast to debugging that analyzes one linear execution trace of a program, other verification techniques such as model checking and theorem proving analyze all possible execution traces of a program at once.

multi-agent program and can activate a debugging tool when it is evaluated to true. The debugging tools are inspired by traditional debugging tools, extended with the functionality to verify a multi-agent program execution trace. One example of such a debugging tool is a multi-agent version of the breakpoint. The breakpoint can halt the execution of a single agent program, a group of agent programs or the complete multi-agent program. This multi-agent version of the breakpoint can also have an MDL expression as a condition, making it a conditional breakpoint.

6.5.2 Specification Language for Debugging: Syntax

In this section, we present the syntax of the MDL written in EBNF notation. An expression of this language describes a property of an execution of a multi-agent program in *APL* and can be used to perform/activate debugging actions/tools. In the following, ⟨*group_id*⟩ is a group identifier (uncapitalized string), ⟨*agent_id*⟩ an agent identifier (uncapitalized string), ⟨*query_name*⟩ a property description name (a reference to a specified property used in the definition of macros; see later on for a discussion on macros), ⟨*Var*⟩ a variable (Variables are capitalized strings), [*all*] indicates the group of all agents, and ⟨*agent_var*⟩ an agent identifier, a group identifier, or a variable. Finally, we use *Bquery*, *Gquery*, and *Pquery* to denote an agent's Beliefs, Goals, and Plans, respectively.

⟨*group_def*⟩ ::= "[" ⟨*group_id*⟩ "]" " = " ⟨*agent_list*⟩
⟨*agent_list*⟩ ::= "[" ⟨*agent_id*⟩ ("," ⟨*agent_id*⟩) ∗ "]"
⟨*mdl_pd*⟩ ::= ⟨*query_name*⟩ "{" ⟨*mdl_query*⟩ "}"
⟨*mdl_query*⟩ ::= "{" ⟨*mdl_query*⟩ "}"
| ⟨*agent_var*⟩ "@Beliefs(" ⟨*Bquery*⟩ ")"
| ⟨*agent_var*⟩ "@Goals(" ⟨*Gquery*⟩ ")"
| ⟨*agent_var*⟩ "@Plans(" ⟨*Pquery*⟩ ")"
| ⟨*UnOp*⟩ ⟨*mdl_query*⟩
| ⟨*mdl_query*⟩ ⟨*BinOp*⟩ ⟨*mdl_query*⟩
| "?" ⟨*query_name*⟩
⟨*BinOp*⟩ ::= "and" | "or" | "implies" | "until"
⟨*UnOp*⟩ ::= "not" | "next" | "eventually" | "always"
⟨*agent_var*⟩ ::= ⟨*Var*⟩ | ⟨*agent_id*⟩ | ⟨*group_id*⟩ | "[all]"

Note that ⟨*mdl_pd*⟩ is a specified property that describes the (temporal and cognitive) behavior of a multi-agent program execution.

In order to specify that either `all` agents believe that there is a bomb at position `2,3` (i.e., `bomb(2,3)`) or `all` agents believe that there is no bomb at that position (i.e. `not bomb(2,3)`), we can use the following MDL expression.

```
[all]@Beliefs(bomb(2,3))  or
[all]@Beliefs(not bomb(2,3))
```

Since specified properties in our framework are always evaluated in the initial state of the program execution trace (and thus specified by the multi-agent program), the above property will evaluate to true if it holds in the initial state. Therefore, if this property is evaluated to true in a program execution trace, then it will evaluate to true for the rest of the program execution. Note that if this property should hold in all states of the program execution, then it should be put in the scope of the 'always' operator. Moreover, if the property should hold in the last state of the program execution, then it should be put in the scope of the 'eventually' operator.

We can generalize the above property by assigning a name to it and parameterizing the specific beliefs (in this case `bomb(X,Y)`). This generalization allows us to specify a property as a macro that can be used to define more complex properties. For example, consider the following generalization (macro) that holds in a state of a multi-agent program if and only if either all agents believe the given belief ϕ or all agents do not believe ϕ.

```
isSharedBelief(φ){
    [all]@Beliefs(φ) or
    [all]@Beliefs(not φ)
}
```

Note that `isSharedBelief(`ϕ`)` can now be used (e.g., in other property specifications) to check whether or not ϕ is a shared belief. In general, one can use the following abstract scheme to name an MDL expression. Parameters `Var1, Var2,` and `Var3` are assumed to be used in the MDL expression.

```
name( Var1, Var2, Var3, ...) { MDL expression }
```

The following example demonstrates the use of macros. To use an MDL expression inside another one, the macro's names should be preceded by a "?" mark. We now define a cell as detected when agents agree on the content of that cell. We define `detectedArea(R)` as follows.

```
detectedArea(X, Y) { ?isSharedBelief( bomb(X,Y) ) }
```

The next example shows an MDL expression that can be used to verify whether the `gridworld` will eventually be clean if an agent has the goal to clean it. In particular, the expression states that if an agent `A` has the goal to clean the `gridworld` then eventually that agent `A` will believe that the `gridworld` is clean.

```
cleanEnvironment(A) {
    A@Goals(clean(gridworld))
    implies
    eventually A@Beliefs(clean(gridworld))
}
```

It is important to note that if this property evaluates to false for an execution thusfar, it may *not* continue to be false for the rest of the execution (cf. Definition 6.15). This is due to the evaluation of the `eventually` operator in the context of finite traces. In particular, if the above property evaluates to false for a finite program execution trace, then it may not evaluate to false for a suffix of that trace. One particular use of the eventually operator is therefore to check and stop the execution of a multi-agent program when it reaches a state with a specific property.

The following MDL expression states that an agent `A` will not unintentionally drop the bomb that it carries. More specifically, the expression states that if an agent believes to carry a bomb, then the agent will believe to carry the bomb until it has a plan to drop the bomb. It is implicitly assumed that all plans will be successfully executed.

```
doesNotLoseBomb(A) {
  always (  A@Beliefs(carry(bomb))
            implies
            (  A@Beliefs(carry(bomb))
               until
               A@Plans(dropped_bomb)
            )
         )
}
```

6.5.3 Specification Language for Debugging: Semantics

The semantics of the MDL language describes how a property is evaluated against a trace of a BDI-based multi-agent program. In the context of debugging, we consider *finite traces* generated by partial execution of multi-agent programs (a partial execution of a program starts in the initial state of the program and stops after a finite number of deliberation steps). A finite trace is a finite sequence of multi-agent program states in which the state of each agent is a tuple consisting of beliefs, goals, and plans.

An MDL expression is evaluated with respect to a finite multi-agent program trace that results from a partial execution of a multi-agent program. In the following, we use t to denote a finite trace, $|t|$ to indicate the length of the trace t (a natural number; a trace consists of 1 or more states), st to indicate a trace starting with state

s followed by the trace t, $|st| = 1 + |t|$, and functions *head* and *tail*, defined as follows: $head(st) = s$, $head(t) = t$ if $|t| = 1$, $head(t,i) = A_i$ if $head(t) = \{A_1, \ldots, A_i, \ldots, A_n\}$, $tail(st) = t$ and $tail(t)$ is undefined if $|t| \leq 1$ (*tail* is a partial function). Moreover, given a finite trace $t = s_1 s_2 \ldots s_n$, we write t_i to indicate the suffix trace $s_i \ldots s_n$.

Definition 6.15. Let $s_i = \{A_1, \ldots, A_n\}$ be a multi-agent program configuration and let $t = s_1 s_2 \ldots s_n$ be a finite trace of a multi-agent program such that $|t| \geq 1$. Let also the evaluation functions V_b, V_g, and V_i be as defined in Definitions 6.10 and 6.14, respectively. The satisfaction of MDL expressions by the trace t is defined as follows:

$t \models \texttt{i@Beliefs}(\phi) \Leftrightarrow \phi \in V_b(head(t,i))$

$t \models \texttt{i@Goals}(\phi) \Leftrightarrow \exists s \in V_g(head(t,i)) : \phi \in s$

$t \models \texttt{i@Plans}(\phi) \Leftrightarrow \exists s \in V_i(head(t,i)) : \phi \in s$

$t \models \phi \texttt{ and } \psi \Leftrightarrow t \models \phi \ \textit{and} \ t \models \psi$

$t \models \phi \texttt{ or } \psi \Leftrightarrow t \models \phi \ \textit{or} \ t \models \psi$

$t \models \phi \texttt{ implies } \psi \Leftrightarrow t \models \phi \ \textit{implies} \ t \models \psi$

$t \models \texttt{not } \phi \Leftrightarrow t \not\models \phi$

$t \models \texttt{next } \phi \Leftrightarrow tail(t) \models \phi \ \textit{and} \ |t| > 1$

$t \models \texttt{eventually } \phi \Leftrightarrow \exists i \leq |t| \ \ (t_i \models \phi)$

$t \models \texttt{always } \phi \Leftrightarrow \forall i \leq |t| \ \ (t_i \models \phi)$

$t \models \phi \texttt{ until } \psi \Leftrightarrow \exists i \leq |t| \ \ (t_i \models \psi \ \textit{and} \ \forall j < i \ \ (t_j \models \phi))$

Based on this definition of MDL expressions, we have implemented some debugging tools that are activated and updated when their corresponding MDL expression holds in a partial execution of a multi-agent program. These debugging tools are described in the next section. This definition of the satisfaction relation can behave different than the standards definition of satisfaction relation of LTL which is defined on infinite traces. For example, some LTL properties such as $\neg\texttt{next}\phi = \texttt{next}\neg\phi$ are valid only for infinite traces. However, the validity of such properties is not relevant for our debugging framework as debugging is only concerned with the execution thusfar and therefore with finite traces. We would like to emphasize that different LTL semantics for finite traces of program executions have been proposed. See [28] for a comparison between different proposals.

6.6 Multi-Agent Debugging Tools

A well-known technique often used for debugging single sequential and concurrent programs is a *breakpoint*. A breakpoint is a marker that can be placed in the program's code. Breakpoints can be used to control the program's execution. When

the marker is reached program execution is halted. Breakpoints can be either conditional or unconditional. Unconditional breakpoints halt the program execution when the breakpoint marker is reached. Conditional breakpoints only halt the program execution when the marker is reached and some extra condition is fulfilled. Another (similar) functionality, that can be used to re-synchronize program executions, is called a process barrier breakpoint. Process barrier breakpoints are much like normal breakpoints. The difference is they halt the processes that reached the barrier point until the last process reaches the barrier point. A different debugging technique used for traditional programming practices is called the *watch*. The watch is a window used to monitor variables' values. Most watch windows also allow the developer to type in a variable name and if the variable exists the watch will show the variable's value. In the IDEs of most high-level programming languages the watch is only available when the program's execution is halted. Other traditional debugging techniques are logging and visualization. Logging allows a developer to write some particular variable's value or some statement to a logging window or a file. Visualization is particularly helpful in the analysis and fine tuning of concurrent systems. Most relevant in light of our research is the ability to visualize the message queue.

Despite numerous proposals for BDI-based multi-agent programming languages, there has been little attention on building effective debugging tools for *BDI-based* agent-oriented programs. The existing debugging tools for BDI-based programs enable the observation of program execution traces (the sequence of program states generated by the program's execution) [63,114,121,122,343] and browsing through these execution traces, allowing to run multi-agent programs in different execution modes by for example using breakpoints and assertions [63, 114, 122, 343], observing the message exchange between agents and checking the conformance of agents' interactions with a specific communication protocol [84,114,343,345,346,425]. Although most proposals are claimed to be applicable to other BDI-based multi-agent programming languages, they are presented for a specific multi-agent platform and the corresponding multi-agent programming language. In these proposals, debugging multi-agent aspects of such programs are mainly concerned with the interaction between individual agents and the exchanged messages. Finally, the temporal aspects of multi-agent program execution traces are only considered in a limited way and not fully exploited for debugging purposes.

This section presents a set of Multi-Agent Debugging Tools (MADTs) to illustrate how the MDL language can be used to debug multi-agent programs. In order to use the debugging tools, markers are placed in the multi-agent programs to denote under which conditions which debugging tool should be activated. A marker consists of an (optional) MDL expression and a debugging tool. The MDL expression of a marker specifies the condition under which the debugging tool of the marker should be activated. In particular, if the MDL expression of a marker evaluates to true for a given finite trace/partial execution of a multi-agent program, then the debugging tool of the marker will be activated. When the MDL expression of a marker is not given (i.e., not specified), then the associated debugging tool will be activated as soon as the multi-agent program is executed. Besides an MDL expression, a marker can also have a group parameter. This group parameter specifies which

agents the debugging tool operates on. The general syntax of a marker is defined as follows:

⟨*marker*⟩ ::= "`MADT`("⟨*madt*⟩["," ⟨*mdl_query*⟩][", @"⟨*group*⟩]")"
⟨*group*⟩ ::= "["⟨*group_id*⟩"]"|⟨*agent_list*⟩

The markers that are included in a multi-agent program are assumed to be processed by the interpreter of the corresponding multi-agent programming language. In particular, the execution of a multi-agent program by the interpreter will generate consecutive states of a multi-agent program and, thereby, generating a trace. At each step of the trace generation (i.e., at each step where a new state is generated) the interpreter evaluates the MDL expression of the specified markers in the initial state of the finite trace (according to the definition of the satisfaction relation; see definition 6.15) and activates the corresponding debugging tools if the MDL expressions are evaluated to true. This means that the trace of a multi-agent program is verified after every change in the trace. This mode of processing markers corresponds to the *continuous debugging mode* and does not stop the execution of the multi-agent program; markers are processed *during* the execution of the program. In the *post mortem debugging mode*, where a multi-agent program is executed and stopped after some deliberation steps, the markers are processed based on the finite trace generated by the partial execution of the program. It is important to note again that MDL expressions are always evaluated in the initial state of traces as we aim at debugging the (temporal) behavior of multi-agent programs and thus their execution traces from the initial state. The following example illustrates the use of a marker in a multi-agent program:

```
MADT( breakpoint_madt ,
      eventually cleaner@Beliefs(bomb(X,Y))
    )
```

This marker, which can be placed in the multi-agent program, activates a breakpoint as soon as the cleaner agent believes that there is a bomb in a cell of the `gridworld`. It is important to note that if no MDL expression is given in a specified marker, then the associated debugging tool will be activated after each update of the trace. Removing the specified MDL expression from the abovementioned marker means that the execution of the multi-agent program will be stopped after each trace update. This results in a kind of stepping execution mode. Furthermore, if no group parameter is given in the marker, the "[all]" group is used by default.

In the rest of this section, we illustrate the use of a set of debugging tools that have shown to be effective in debugging software systems. Examples of debugging tools are breakpoint, logging, state overview, or message list. The behavior of these debugging tools in the context of markers are explained in the rest of this section. The proposed set of debugging tools is by no means exhaustive and can be extended with other debugging tools. We thus do neither propose new debugging tools nor

evaluate their effectiveness. The focus of this chapter is a framework for using (existing) debugging tools to check cognitive and temporal behavior of multi-agent program. Our approach is generic in the sense that a debugging tool can be associated with an MDL expression by means of a marker and that markers can be used in two debugging modes.

6.6.1 Breakpoint

The breakpoints for multi-agent programs are similar to breakpoints used in concurrent programs. They can be used to pause the execution of a single agent program, a specific group of agent programs, or the execution of the entire multi-agent program. Once the execution of a program is paused, a developer can inspect and browse through the program execution trace generated so far (including the program state in which the program execution is paused). The developer can then continue the program execution in a stepping mode to generate consecutive program states. An attempt to further execute the program continuously (not in stepping mode) pauses immediately since the MDL expression associated to the breakpoint will be evaluated in the initial state of an extension of the same trace. In general, if an MDL expression evaluates to true in a state of a trace, then it will evaluate to true in the same state of any extension of that trace.

The example below demonstrates the use of a conditional breakpoint on the agents `explorer1` and `explorer2`. The developer wants to pause both agents as soon as agent `cleaner` has the plan to go to cell `(5, 5)`.

```
MADT( breakpoint_madt,
      eventually cleaner@Plans(goto(5, 5)) ,
      @[explorer1, explorer2]
    )
```

Note that it is possible to use the cognitive state of more than one agent as the break condition. The next example demonstrates how a developer can get an indication about whether the number of explorer and cleaner agents are suitable for a certain scenario. In fact, if there are not enough cleaners to remove bombs, or when all explorers are located at the same area, then all explorers will find the same bomb.

```
MADT( breakpoint_madt,
      eventually [explorers]@Beliefs(bomb(X,Y))
    )
```

The breakpoint tool is set to pause the execution of all agents, once all agents that are part of the "explorers" group have the belief that a bomb is located at the same cell `(X,Y)`. Note that it need not be explicitly defined to pause the execution of *all*

agents. The breakpoint is useful in conjunction with the watch tool to investigate the mental state of the agent. Other agent debugging approaches, e.g., [114], propose a similar concept for breakpoints, but for a single BDI-based agent program. Also, Jason [63] allows annotations in plan labels to associate extra information to a plan. One standard plan annotation is called a breakpoint. If the debug mode is used and the agent executes a plan that has a breakpoint annotation, execution pauses and the control is given to the developer, who can then use the step and run buttons to carry on the execution. Note that in contrast with other approaches, the condition in our approach may contain logic and temporal aspects.

6.6.2 Watch

The watch can display the current mental state of one or more agents. Furthermore, the watch allows the developer to query any of the agents' bases. The developer can, for example, use the watch to check if a belief follows from the belief base. It is also possible to use an MDL expression in the watch; if the expression evaluates to *true*, the watch will show the substitution found. The watch tool can also be used to visualize which agents have shared or conflicting beliefs. The watch tool is regularly used in conjunction with a conditional breakpoint. Once the breakpoint is hit, the watch tool can be used to observe the mental state of one or more agents. In general, the watch tool should be updated unconditionally and for all agents in the system. Adding `MADT(watch_madt)` to a multi-agent program will activate the watch on every update of its execution trace. In Jason and Goal [63, 224], a similar tool is introduced which is called the mind inspector. This mind inspector, however, can only be used to observe the mental state of individual agents. Jadex [343] offers a similar tool called the BDI-inspector which allows visualization and modification of internal BDI-concepts of individual agents.

6.6.3 Logging

Logging is done by the usage of probes which, unlike breakpoints, do not halt the multi-agent program execution. When a probe is activated it writes the current state of a multi-agent program, or a part of it, to a log screen or a file (depending on the type of probe). Using a probe without an MDL expression and without a group specification is very common and can be done by adding `MADT(probe_madt)` in multi-agent programs. The probe will be activated on every update of the program trace such that it keeps a log of all multi-agent program states. The next example saves the state of the multi-agent program when the cleaner agent drops a bomb in a depot, but there is still some agent who believes the bomb is still at its original place.

```
MADT( probe_madt,
      eventually( cleaner@Plans(dropBomb(X,Y))
                  and
                  A@Beliefs(bomb(X,Y))
                )
)
```

This means that the `probe_madt` will be activated directly after an execution step that generates a trace on which the MDL expression evaluates to true. A developer can thus use such expressions (of the form `eventually`ϕ) in order to be notified at once and as soon as the program execution satisfies it. Once this expression evaluates to true, the developer should know that any continuation of the program execution will evaluates it to true. Thus, from a developer's perspective, properties specified by expressions of the form `eventually`ϕ can be used to get notified (or stop the execution) only once and as soon as it is satisfied. Similar work is done in Jadex [410] where a logging agent is introduced to allow collection and viewing of logged messages from Jadex agents. It should be noted that the probes in our approach offer the added functionality of filtering on a cognitive condition of *one ore more agents*.

6.6.4 Message-list

Another visualization tool is the message-list, which is one of the simplest forms of visualization. The message-list keeps track of the messages sent between agents, by placing them in a list. This list can be sorted on each of the elements of the messages. For example, sorting the messages on the "sender" element can help finding a specific message send by a known agent. Besides ordering, the list can also be filtered. For example, we could filter on "Senders" and only show the message from the sender with the name "cleaner". To update the message-list on every update of the trace, we can place the marker `MADT(message_list_madt)` in the multi-agent program. Another use of the message-list could be to show only the messages from within a certain group, e.g., `MADT(message_list_madt, @[explorers])` can be used to view the messages exchanged between the members of the explorers group. Finally, in our proposal one can also filter exchanged messages based on conditions on the mental states of individual agents. For example, in the context of our gridworld example, one can filter useless messages, i.e., messages whose content are known facts. Exchanging too many useless messages is a sign of non-effective communication. The example below triggers the message list when an agent `A`, who believes there is a bomb at coordinates `X,Y`, receives a message about this fact from another agent `S`.

```
MADT( message_list_madt,
      eventually( A@Beliefs(bomb(X,Y))
                  and
```

```
                A@Beliefs(message(S,P,bombAt(X,Y)))
            )
    )
```

In this example, it is assumed that a received message is automatically added to the belief base of the receiving agent, and that the added message has the form $message(Sender, Performative, Content)$. All existing agent programming platforms offer a similar tool to visualize exchanged messages. The main difference with our approach is the ability to log when certain cognitive conditions hold.

6.6.5 Causal tree

The causal tree tool shows each message and how it relates to other messages in a tree form. The hierarchy of the tree is based on the relation between messages (replies become branches of the message they reply to). Messages on the same hierarchical level, of the same branch, are ordered chronologically. The advantage of the causal tree (over the message-list) is that it is easier to spot communication errors. When, for example, a reply is placed out of context (not in relation with its cause) this implies there are communication errors. The causal tree also provides an easy overview to see if replies are sent when required. The causal tree tool can be used by adding the marker `MADT(causal_tree_madt)` to multi-agent programs. Another example could be to set the group parameter and only display message from a certain group, e.g., `MADT(causal_tree_madt, @[explorers])`. It should be noted that for the causal tree to work, the messages need to use performatives such as inform and reply.

6.6.6 Sequence diagram

The sequence diagram is a commonly used diagram in the Unified Modeling Language (UML) or its corresponding agent version (AUML). An instantiation of a sequence diagram can be used to give a clear overview of (a specific part of) the communication in a multi-agent program. They can help to find irregularities in the communication between agents. The sequence diagram tool can be used by adding the marker `MADT(sequence_diagram_madt)` to multi-agent programs. This example updates the sequence diagram on every update of the trace. Another example could be to use the group parameter and only update the sequence diagram for the agents in a certain group, e.g., `MADT(sequence_diagram_madt, @[cleaner, explorer2])`. Adding this marker to our multi-agent program will show the communication between the agents “cleaner” and “explorer2”. The sequence diagram tool is useful in conjunction with a conditional breakpoint and the stepwise execution mode where the diagram can be constructed step by step. The sequence diagram

is also useful in conjunction with the probe. The probe can be used to display detailed information about the messages. Similar tools are proposed in some other approaches, e.g., the sniffer agent in [32]. However, we believe that the sequence diagram tool in our approach is more effective since it can be used for specific parts of agent communication.

6.6.7 Visualization

Sometimes the fact that a message is sent is more important than the actual contents of the message. This is, for example, the case when a strict hierarchy forbids certain agents to communicate. In other cases it can be important to know how much communication takes place between agents. For such situations the *dynamic agent communication* tool is a valuable add-on. This tool shows all the agents and represents the communication between the agents by lines. When agents have more communication overhead the line width increase in size and the agents are clustered closer together. This visualization tool, which can be triggered by adding the marker `MADT(dynamic_agent_madt)` to multi-agent program, is shown in figure 6.2.

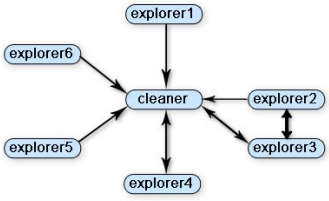

Fig. 6.2 The dynamic agent communication tool.

Another visualization tool is the static group tool. This debugging tool, which shows specific agent groups, is illustrated in figure 6.3. The line between the groups indicates the (amount) of communication overhead between the groups. In addition the developer can "jump into" a group and graphically view the agents and the communication between them.

Fig. 6.3 The static group tool.

The static group tool can be helpful to quickly check if the correct agents are in the correct group. It can also be used to check communication between different groups. If two groups show an unusual amount of communication overhead the developer can jump into the group and locate the source of the problem. The marker to activate the static group tool can be specified as follow:

```
MADT(static_group_madt, @[explorers])
MADT(static_group_madt, @[cleaners])
```

The above markers update the tool on every change of the multi-agent program trace. According to these markers, the groups "explorers" and "cleaners" will be visualized. Generally it is most valuable to have a visualization of all communication between agents. However, to pinpoint the exact problem in a communication protocol it can be an invaluable addition to use a condition, which filters the messages that are shown. These same principles apply to the filtered view. As discussed in the related works section, other approaches (e.g., [84]) offers similar tools.

6.7 Conclusion and Future Work

In this chapter we have shown two methods for showing the correctness of BDI-based multi-agent programs that are complementary to the well-known methods of model checking and theorem proving that are used in the realm of multi-agent verification. Various chapters in this volume present different model-checking and theorem proving approaches for multi-agent programs.

The first one is a general one allowing us to prove that certain properties expressed in a specification language hold for the agent programming language, and thus for all individual agents that are implemented in this agent programming language. To this end we showed how a BDI-based agent-oriented programming language can be related to a BDI specification language in a systematic and natural manner. We used here a very simple agent programming language which can be extended in many different ways. In [77], a comparable effort is undertaken for the agent programming language AgentSpeak. The specification language in that work is, however, not a temporal logic, and the properties proven are different (not related to dynamics of goals).

The multi-agent programming language and its corresponding logic, presented in this chapter, are designed to focus on the cognitive aspects and related properties of individual agent programs. The multi-agent programming language can be extended with communication actions and shared environments to allow the implementation of multi-agent systems in which individual agents interact by either sending and receiving messages or performing actions in their shared environment. Most existing agent programming languages have already proposed specialized constructs to implement communication and shared environments. Future research is needed to

propose logical frameworks to specify and verify the interaction properties of multi-agent programs.

The properties that we have studied in this chapter are general in the sense that we consider the set of all possible execution traces of multi-agent programs. In practice, the execution of a multi-agent program is often based on an interpreter that uses a specific execution strategy. For example, an interpreter may apply one/all planning rules before executing one/all plans, or executing one/all plans before applying one/all planning rules. In principle, there are many different strategies that can be used to execute a multi-agent program. In future work we will study properties that are related to a specific execution strategy. This enables the verification of properties of a multi-agent program for a given interpreter.

The second approach to the correctness of BDI-based agent programs we showed is based on debugging. Our proposal extends previous approaches by debugging the *interaction* between implemented agents, not only in terms of the exchanged messages, but also in terms of the relations between their internal states. A developer/debugger of a multi-agent program is assumed to have access to the multi-agent program code, which is a realistic assumption, and therefore to the internal state of those programs. The proposed approach is based on a specification language to express cognitive and temporal properties of the executions of multi-agent programs. The expressions of the specification language can be used to trigger debugging tools such as breakpoints, watches, probes, and different visualization tools to examine and debug communication between individual agents.

Since the specification language is abstract, our debugging approach is generic and can be modified and applied to other BDI-based agent programming languages. The only modification is to align the evaluation function of the specification language with the programming language at hand. We have already applied this debugging approach to 2APL [122] platform by modifying its corresponding interpreter to process debugging markers in both debugging modes. The 2APL interpreter evaluates the expressions of the specification language based on the partial execution trace of the multi-agent programs. We have also implemented the proposed debugging tools that are discussed in this paper for the 2APL platform.

We plan to extend the MDL language by including constructs related to the external environments of a multi-agent program. In this way, one can specify properties that relates agent states to the state of the external environments. Moreover, we plan to extend our debugging framework with the society aspects that may be involved in multi-agent programming languages [325]. Recent developments in multi-agent programming languages [131, 166, 188, 243] have proposed specific programming constructs enabling the implementation of social concepts such as norms, roles, obligations, and sanctions. Debugging such multi-agent programs requires therefore specific debugging constructs to specify properties related to the social aspects and facilitate finding and resolving defects involved in such programs.

The presented debugging framework assumes all agents are developed on one single platform such that their executions for debugging purposes are not distributed on different platforms. One important challenge and a future work on debugging

multi-agent systems remains the debugging of multi-agent programs that run simultaneously on different platforms. The existing debugging techniques are helpful when errors manifest themselves directly to the system developers. However, errors in a program do not always manifest themselves directly. For mission and industrial critical systems it is therefore necessary to extensively test the program before deploying it. This testing should remove as many bugs (and possible defects) as possible. However, it is infeasible to test every single situation the program could be in. We believe that a systematic integration of debugging and testing approaches can be effective in verifying the correctness of multi-agent programs and therefore essential for their developments. A testing approach proposed for multi-agent programs is proposed by Poutakidis and his colleagues [345, 346].

Acknowledgements

We would like to thank Birna van Riemsdijk for her contribution to work on which this chapter is partly based.

Chapter 7

The Norm Implementation Problem in Normative Multi-Agent Systems

D. Grossi, D. Gabbay, and L. van der Torre

Abstract The norm implementation problem consists in how to see to it that the agents in a system comply with the norms specified for that system by the system designer. It is part of the more general problem of how to synthesize or create norms for multi-agent systems, by, for example, highlighting the choice between regimentation and enforcement, or the punishment associated with a norm violation. In this paper we discuss how various ways to implement norms in a multi-agent system can be distinguished in a formal game-theoretic framework. In particular, we show how different types of norm implementation can all be uniformly specified and verified as types of transformations of extensive games. We introduce the notion of retarded preconditions to implement norms, and we illustrate the framework and the various ways to implement norms in the blocks world environment.

D. Grossi
ILLC University of Amsterdam, The Netherlands e-mail: d.grossi@uva.nl

D. Gabbay
Computer Science King's College London, U.K. and ICR University of Luxembourg, Luxembourg
e-mail: dov.gabbay@kcl.ac.uk

L. van der Torre
ICR University of Luxembourg, Luxembourg e-mail: leendert@vandertorre.com

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_7,

7.1 Introduction

Normative multi-agent systems (NMAS) [57] study the specification, design, and programming of systems of agents by means of systems of norms. Norms allow for the explicit specification of the standards of behavior the agents in the systems are supposed to comply with. Once such a set of norms is settled, the question arises of how to organize the agents' interactions in the system, in such a way that those norms do not remain—so to say—dead letter, but they are actually followed by the agents. Designing a NMAS does not only mean to state a number of standards of behavior in the form of a set of norms, but also to organize the system in such a way that those standards of behavior are met by the agents participating in the system. In other words, norm creation [53] distinguishes between the creation of the obligation and norm implementation, because these two problems have different concerns. On the one hand the creation of the obligation says how the ideal can be reached, and the creation of the sanction says how agents can be motivated to comply with the norms such that the ideal will (probably) be reached. The paper moves the first steps towards a formal understanding of the norm implementation problem, defined as follows.

> *The norm implementation problem.* How to make agents comply with a set of norms in a system?

In this paper we introduce a formal framework that can represent various solutions to the norm implementation problem, which can be used to analyze them, or to make a choice among them. For example, in some cases a norm cannot be regimented, such as the norm to return books to the library within two weeks, but in other cases there is the choice between regimentation and enforcement. It is often assumed that regimenting norms makes the system more predictable, since agents cannot violate the norms, but as a consequence it also makes the system less flexible and less efficient. Conceptually, regimentation is easier than enforcement, and since agents are bounded reasoners who can make mistakes, regimentation is often favored by policy makers. However, policy makers are bounded reasoners too, who have to make norms in uncertain circumstances, and therefore most people prefer enforcement over regimentation—at least, when the legal system is reliable. As another example, it is often assumed that very high punishments make the system less efficient than lower ones, due to the lack of incentives for agents once they have violated a norm. Such assumptions are rarely studied formally.

Although ideas about norm implementation can be found scattered over, in particular, the multi-agent systems literature (for instance, in OperA [145], $\mathcal{M}$oise$^+$ [242], AMELI [167], J-Moise+ [243], and in programming languages for multi-agent system programs [125]), they have not yet been presented in a systematic and uniform way. One of the aims of the paper is to do so, providing a formal overarching framework within which it becomes possible to place and compare existing contributions. So the first requirement on a framework for norm implementation is that it can represent existing widely discussed norm implementation methods such

as regimentation and enforcement via sanctioning. Moreover, a second requirement is that such a framework can also represent and reason about new ways of norm implementation, such as changing the existing norms or the method, which we will study in detail, of retarded preconditions. A formal framework may even suggest new ways to implement norms, not discussed before. Our research problem therefore breaks down into the following sub-questions:

1. How is the specification and verification of norm implementation methods related to general specification and verification of multiagent systems?
2. Which formal model can we use to specify and verify norm implementation, and more generally study the norm implementation problem?
3. How to model regimentation in the general formal model of norm implementation?
4. How to model enforcing in the general formal model of norm implementation?
5. How to model norm change in the general formal model of norm implementation?

The specification of normative multi-agent systems often considers a set of agents and a set of norms or organization, which can be specified and verified independently. However, when the set of norms is not designed off-line, but created dynamically at run-time, then this approach does not work. Instead, one can only specify the way in which norms are implemented in a system.

The perspective assumed for the formal framework is based on formal logic and the primary aim of the paper is to present a simple class of logical models, and of transformations on them, as salient representations of the implementation problem. Moreover, we use a game-theoretic approach. We use the simplest approach possible, so we can focus on one same framework for many kinds of norm implementation, and are not lost in technical details about individual approaches. As in classical game theory, our actions are abstract and we do not consider issues like causality. We consider only perfect information games, and we thus do not consider the problem of how norms are distributed and communicated to a society. Moreover, we do not consider concurrent actions of the agents in the composition of actions in plans, although this feature could easily be added. However, we do represent the order of actions, that is, we use extensive form games, because we need to do so to distinguish some of the norm implementation methods, and thus we do not use the more abstract strategic form used in most related work (such as Tennenholtz and Shoham's artificial social systems [400]). We do not go into details of solution concepts of game theory, and we thus basically use a kind of state automata or process models. Within a game-theoretic approach, we need to represent four things: the game without the implemented norm, the game together with the implemented norm, the procedure to go from the former to the latter, and the compliance criterion stating the conditions when norms are fulfilled.

We test our model also by introducing the notion of implementation via retarded preconditions. For example, assume that in the tax regime of a country, for people who leave the country there is a period of three years after which it is checked

whether someone has really left the country. In this example, the precondition is checked only after three years, and if the person has returned to the country, the consequences of leaving the country are retracted. Likewise, with actions with non-deterministic effects, we can say that the precondition depends on the effect. For example, if there are no concurrent updates in the database, then the update will be accepted, otherwise it will be rolled back. In the blocks world, which will be used as running example throughout the paper, assume that a block may not be put on another block if it stays there for three minutes. If it stayed there for three minutes, then we can undo the action of putting the block on top of the other one (alternatively, one can sanction it, of course). Retarded preconditions offer more flexibility than simple regimentation. For example, consider the norm that it is forbidden to throw 6 on a dice. With retarded preconditions, we can throw the dice and do a roll-back when 6 appears. Without retarded preconditions, the only way to regiment it, is to forbid throwing the dice. We distinguish norm regimentation from automatic enforcement and enforcement agents, assuming actions can be taken only if preconditions hold. Some of such actions are forbidden, so all actions in order to be taken must satisfy the precondition. For regimentation we consider violation conditions as retarded preconditions of actions. In this action model, assumed actions can be taken in some cases even though the preconditions do not hold. When a violation occurs, i.e., when the retarded precondition does not hold, the various strategies to implement norms follow as a consequence.

We illustrate the framework and the various ways to implement norms in the blocks world environment, because the well-known planning environment explains the use of normative reasoning and the challenges of norm implementation for a large AI audience. There are many variants of the blocks world around, we use a relatively simple one with deterministic actions and without concurrent actions. An alternative well-known example we could have chosen is the Wumpus world from Russel and Norvig's textbook [380].

We assume that all norms and their implementations are known once they are created, and we thus do not study the norm distribution problem. Moreover, we assume that everyone accepts the existence of a new norm, even when he does not comply with it. Thus, we do not consider the norm acceptance problem. We do not consider cognitive aspects of agents, and we thus do not consider the bridge between our framework for MAS and existing BDI frameworks for cognitive agents (see, e.g., [53]).

The paper follows the research questions and proceeds as follows. In Section 7.2 we give a short introduction in the use of normative systems in computer science in general, and specification and verification of normative multi-agent systems in particular. In Section 7.3 we start with the game-theoretic framework for norm implementation and a logic for representing extensive games, and we introduce a running example. Sections 7.4, 7.5, 7.6, and 7.7 provide formal semantics to the four implementation strategies of regimentation, enforcement, enforcers, and, respectively, normative change. In presenting such semantics due care will be taken to relate our framework to existing literature showing how the framework is general enough to

categorize, at a higher abstraction level, the various contributions available in the literature. The findings of each section is illustrated by means of the running example. In Section 7.8 related work at the intersection of norms and multi-agent systems is discussed. Conclusions follow in Section 7.9.

7.2 Normative multi-agent systems

In this section we first give a short summary of the main issues in using normative systems in computer science, and thereafter we discuss the specification and verification of normative multi-agent systems.

7.2.1 Normative systems in computer science

The survey of the role of normative systems in computer science in this section is taken from [50]. For a discussion on philosophical foundations for normative multi-agent systems, see [51, 207].

There is an increasing interest in normative systems in the computer science community, due to the observation five years ago in the so-called AgentLink Roadmap [293, Fig. 7.1], a consensus document on the future of multi-agent systems research, that norms must be introduced in agent technology in the medium term (i.e., now!) for infrastructure for open communities, reasoning in open environments and trust and reputation. The first definition of a normative multi-agent system emerged after two days of discussion at the first workshop on normative multi-agent systems NorMAS held in 2005 as a symposium of the Artificial Intelligence and Simulation of Behaviour convention (AISB) in Hatfield, United Kingdom:

The normchange definition. "A normative multi-agent system is a multi-agent system together with normative systems in which agents on the one hand can decide whether to follow the explicitly represented norms, and on the other the normative systems specify how and in which extent the agents can modify the norms" [57].

A distinction has been made between systems in which norms must be explicitly represented in the system (the 'strong' interpretation) or that norms must be explicitly represented in the system specification (the 'weak' interpretation). The motivation for the strong interpretation of the explicit representation is to prevent a too general notion of norms. Any requirement can be seen as a norm the system has to comply with; but why should we do so? Calling every requirement a norm makes the concept empty and useless. The weak interpretation is used to study the following two important problems in normative multi-agent systems.

Norm compliance. How to decide whether systems or organizations comply with relevant laws and regulations? For example, is a hospital organized according to medical regulations? Does a bank comply with Basel 2 regulations?

Norm implementation. How can we design a system such that it complies with a given set of norms? For example, how to design an auction such that agents cannot cooperate?

Norms are often seen as a kind of (soft) constraints that deserve special analysis. Examples of issues which have been analyzed for norms but to a less degree for other kinds of constraints are ways to deal with violations, representation of permissive norms, the evolution of norms over time (in deontic logic), the relation between the cognitive abilities of agents and the global properties of norms, how agents can acquire norms, how agents can violate norms, how an agent can be autonomous [116] (in normative agent architectures and decision making), how norms are created by a legislator, emerge spontaneously or are negotiated among the agents, how norms are enforced, how constitutive norms are used to describe institutions, how norms are related to other social and legal concepts, how norms structure organizations, how norms coordinate groups and societies, how contracts are related to contract frames and contract law, how legal courts are related, and how normative systems interact?

Norms can be changed by the agents or the system, which distinguishes this definition of normative multi-agent system from the common framework used in the Deontic Logic in Computer Science (or ⊿EON) community, and led to the identification of this definition as the "normchange" definition of normative multi-agent systems. For example, a norm can be made by an agent, as legislators do in a legal system, or there can be an algorithm that observes agent behavior, and suggests a norm when it observes a pattern. The agents can vote on the acceptance of the norm. Likewise, if the system observes that a norm is often violated, then apparently the norm does not work as desired, and it undermines the trust of the agents in the normative system, so the system can suggest that the agents can vote whether to retract or change the norm.

After four days of discussion, the participants of the second workshop on normative multi-agent systems NorMAS held as Dagstuhl Seminar 07122 in 2007 agreed to the following consensus definition:

The mechanism design definition. "A normative multi-agent system is a multi-agent system organized by means of mechanisms to represent, communicate, distribute, detect, create, modify, and enforce norms, and mechanisms to deliberate about norms and detect norm violation and fulfilment." [59]

According to Boella *et al.*, "the emphasis has shifted from representation issues to the mechanisms used by agents to coordinate themselves, and in general to organize the multi-agent system. Norms are communicated, for example, since agents in open systems can join a multi-agent system whose norms are not known. Norms are distributed among agents, for example, since when new norms emerge the agent

could find a new coalition to achieve its goals. Norm violations and norm compliance are detected, for example, since spontaneous emergence norms of among agents implies that norm enforcement cannot be delegated to the multi-agent infrastructure." [59] They refer to game theory in a very liberal sense, not only to classical game theory studied in economics, which has been criticized for its ideality assumptions. Of particular interest are alternatives taking the limited or bounded rationality of decision makers into account.

Games can explain that norms should satisfy various properties to be effective as a mechanism to obtain desirable behavior. For example, the system should not sanction without reason, as the norms would loose their force to motivate agents. Moreover, sanctions should not be too low, but they also should not be too high. Otherwise, once a norm is violated, there is no way to prevent further norm violations.

Games can explain also the role of various kinds of norms in a system. For example, assume that norms are added to the system one after the other and this operation is performed by different authorities at different levels of the hierarchy. Lewis "master and slave" game [284] shows that the notion of permission alone is not enough to build a normative system, because only obligations divide the possible actions into two categories or spheres: the sphere of prohibited actions and the sphere of permitted (i.e., not forbidden) actions or "the sphere of permissibility". More importantly, Bulygin [93] explains why permissive norms are needed in normative systems using his "Rex, Minister and Subject" game. "Suppose that Rex, tired of governing alone, decides one day to appoint a Minister and to endow him with legislative power. [...] an action commanded by Minister becomes as obligatory as if it would have been commanded by Rex. But Minister has no competence to alter the commands and permissions given by Rex." If Rex permits hunting on Saturday and then Minister prohibits it for the whole week, its prohibition on Saturday remains with no effect.

As another example, Boella and van der Torre's game theoretic approach to normative systems [55] studies the following kind of normative games.

Violation games: interacting with normative systems, obligation mechanism, with applications in trust, fraud and deception.

Institutionalized games: counts-as mechanism, with applications in distributed systems, grid, p2p, virtual communities.

Negotiation games: MAS interaction in a normative system, norm creation action mechanism, with applications in electronic commerce and contracting.

Norm creation games: multi-agent system structure of a normative system, permission mechanism, with applications in legal theory.

Control games: interaction among normative systems, nested norms mechanism, with applications in security and secure knowledge management systems.

Norms are not only seen as the mechanism to regulate behavior of the system, but they are often also part of a larger institution. This raises the question what

precisely the role of norms is in such an organization. Norms are rules used to guide, control, or regulate desired system behavior. However, this is not unproblematic, since norms can be violated, and behavior of agents may change in unexpected ways when norms are introduced due to self organization. Norms can also be seen as one of the possible incentives to motivate agents, which brings us again back to economics. The fact that norms can be used as a mechanism to obtain desirable system behavior, i.e., that norms can be used as incentives for agents, implies that in some circumstances economic incentives are not sufficient to obtain such behavior. For example, in a widely discussed example of the so-called centipede game, there is a pile of thousand pennies, and two agents can in turn either take one or two pennies. If an agent takes one then the other agent takes turn, if it takes two then the game ends. A backward induction argument implies that it is rational only to take two at the first turn. Norms and trust have been discussed to analyze this behavior, see [235] for a discussion.

A rather different role of norms is to organize systems. To manage properly complex systems like multi-agent systems, it is necessary that they have a modular design. While in traditional software systems, modularity is addressed via the notions of class and object, in multi-agent systems the notion of organization is borrowed from the ontology of social systems. Organizing a multi-agent system allows to decompose it and defining different levels of abstraction when designing it. Norms are another answer to the question of how to model organizations as first class citizens in multi-agent systems. Norms are not usually addressed to individual agents, but rather they are addressed to roles played by agents [56]. In this way, norms from a mechanism to obtain the behavior of agents, also become a mechanism to create the organizational structure of multi-agent systems. The aim of an organizational structure is to coordinate the behavior of agents so to perform complex tasks which cannot be done by individual agents. In organizing a system all types of norms are necessary, in particular, constitutive norms, which are used to assign powers to agents playing roles inside the organization. Such powers allow to give commands to other agents, make formal communications and to restructure the organization itself, for example, by managing the assignment of agents to roles. Moreover, normative systems allow to model also the structure of an organization and not only the interdependencies among the agents of an organization. Roles are played by other agents, real agents (human or software) who have to act as expected by their role. Each of these elements can be seen as an institution in a normative system, where legal institutions are defined by Ruiter [375] as "systems of [regulative and constitutive] rules that provide frameworks for social action within larger rule-governed settings". They are "relatively independent institutional legal orders within the comprehensive legal orders".

The second NorMAS workshop identified a trend towards a more dynamic interactionist view: "This shift of interest marks the passage of focus from the more static legalistic view of norms (where power structures are fixed) to the more dynamic interactionist view of norms (where agent interaction is the base for norm related regulation)." This ties in to what Strauss [409] called "negotiated order", Goff-

man's [197] view on institutions, and Giddens' [194] structuration theory. See [59] for a further discussion.

7.2.2 *Specification and verification of normative multi-agent systems*

The motivation of our work is to provide an answer to the more general issue of finding a logical formalism that could play for programming NMAS the role that BDI logics (e.g. [360]) have played for the programming of single agents. Such an issue was recognized as central for the NMAS community during the NorMAS'07 Datstuhl Seminar [55], and it was raised in the following incisive form:

$$\text{BDI} : \text{Agent Programming} = ? : \text{NMAS Programming}.$$

This equation represents two issues. First, it raises the question about which concepts should be used for programming NMAS, given that cognitive concepts like beliefs, desires and intentions are used to program individual agents. There is some consensus that instead of cognitive concepts, for normative multi-agent systems social and organizational concepts are needed, such as trust, norms and roles. Second, from a logical perspective, it raises the question which logical languages used for specification and verification can be used for NMAS, like BDI-CTL is used for single agents. Thus far, only partial answers have been given to this question. For example, deontic logic can be used to represent norms, but it cannot be used to say how agents make decisions in normative systems, and about the multi-agent structure of normative systems.

In the traditional framework of artificial social systems, norms are designed offline [400]. Thus, a norm is part of the specification of the multi-agent system, and the normative multi-agent system can be specified and verified using traditional techniques. For example, since BDI-CTL [110] is used as a formal specification and verification language for agent programming, and it has been extended with deontic concepts such as obligations and permissions, called BOID-CTL [87, 88]. Such a logic is simply a modal combination of an agent logic and a modal deontic logic. One drawback of this approach is that the norms are not represented explicitly, see Section 7.8. However, a more fundamental problem with this approach for the specification and verification of normative multi-agent systems is that it is difficult to generalize this approach to the case where norms are created or synthesized at runtime.

The main challenge of specification and verification of normative multiagent systems is the specification and verification of norm change, and in particular the specification and verification of norm creation. Norm creation distinguishes between the creation of the obligation or prohibition, and the creation of the associated sanction. For example, the obligation may be to return books to the library within three weeks,

and the sanction associated with its violation is that a penalty has to be paid, and no other books can be borrowed. The creation of the obligation is often called the norm design or synthesis problem [400], and the creation of the sanction is an example of what we call the norm implementation problem. Thus, in the library example, the norm implementation problem is that given that we want people to return their books within three weeks, how can we build a system such that they will actually do so? However, introducing sanctions is not the only way to implement norms. In other cases, the norm can be regimented, or instead of penalties, rewards can be introduced.

An alternative motivation to break down the specification of a normative multi-agent system is common in computer science: divide and conquer. We distinguish the specification and verification of normative multi-agent systems in three steps: the specification and verification of the agents, the specification and verification of the normative system, and the specification and verification of combining these two systems: the norm implementation problem. This reflects a common ontology of normative multi-agent systems. For example, Figure 7.1 shows the ontology of Boella et al [58] containing a number of concepts related to each other. They divide their ontology in three submodels: the agent model, the institutional model, and the role assignment model, as shown in Figure 7.1. Roughly, an institution is a structure of social order and cooperation governing the behavior of a set of individuals. Institutions are identified with a social purpose and permanence, with the enforcing of rules governing cooperative human behavior. The figure visualizes the three submodels which group the concepts of their ontology.

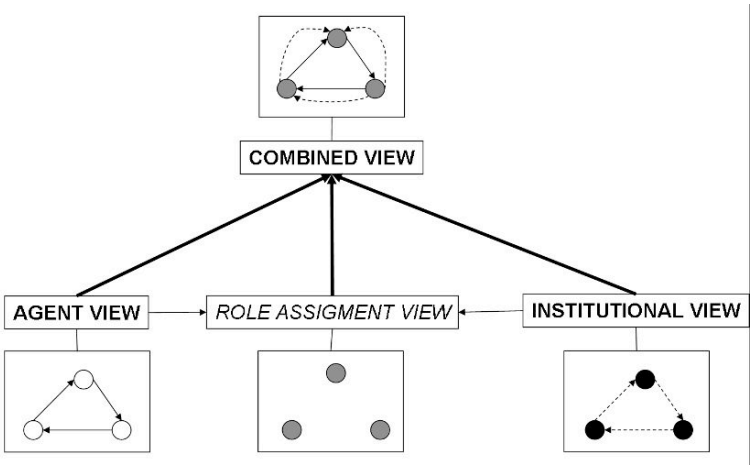

Fig. 7.1 The conceptual metamodel [58].

As Boella *et al.* observe, such a decomposition is common in organizational theory, because an organization can be designed without having to take into account the agents that will play a role in it. For example if a node with the role of sim-

ple user becomes a VO administrator, then this remains transparent for the organizational model. Likewise, agents can be developed without knowing in advance in which institution they will play a role. As shown in Figure 7.1, the agent view is composed by concepts like agent, goal and skill or ability and they are represented by means of a social dependence networks in which nodes are the agents and the edges are the representation of goal-based dependencies. The institutional view, instead, is composed by the notion of role and its institutional goals, skills and facts. As for the agent view, also the institutional one is represented by means of a social institutional dependence network representing the norm-based dependency relations among roles. The role assignment view associates to each agent the roles it plays, depending on the organization in which the agent is playing. All these notions are unified in the combined view where the dependence network represents at the same time both goal-based dependencies and norm-based ones connecting the agents playing roles.

The norm implementation problem combines the specification and verification of agents and norms, analogous to the role assignment problem combines these two specifications. However, it does not consider organizational issue of role assignment, but the question how to ensure that agents do comply with the norms. The decomposition in the role assignment problem is based on the rationale that organizations must be designed independently of the agents that will play a role in it. The decomposition for the norm implementation problem is based on the rationale that in specifying a normative system, it makes sense to first specify the sub-ideal states the system should avoid, and thereafter how to ensure that the system avoids these sub-ideal states. If one norm implementation method does not work, then it can be replaced by another one, without changing the underlying norms.

7.2.3 Assumptions of norm implementation

Summarizing, the norm implementation problem is the part of the more general problem of norm creation which lends itself to specification and verification, since it focusses on the well-defined choice between regimentation and enforcement, or the punishment associated with a norm violation. For example, whether it is hard to give general guidelines for the violation states, since they can be defined by the agents at run-time, it is more straightforward to specify how these violation states must be avoided.

The assumption underlying our research problem is that the norm implementation problem can be studied in isolation. We thus disagree with the common idea that norm implementation can be studied only together with the norm design problem, in the context of norm creation. For example, when a system designer has to choose among various kinds of norms, at the same time he has to take into account how the norm can be implemented. If a norm is chosen which cannot be implemented, such that it will not be complied with, then the norm may even be counterproductive,

undermining the belief or faith into the normative system (in particular, this holds for legal systems). Though we agree that a choice among norms also has to take the available implementations into account, we believe that this is not an argument against studying norm implementation in isolation.

7.3 Formal framework and running example

The present section sets the stage of our formal investigations.

7.3.1 Norms and logic

The formal representation of norms by means of logic has a long-standing history. In the present paper we assume a very simple perspective based on [15, 268, 313] representing the content of norms as labeling of a transition systems in legal and illegal states, which we will call *violation states*. In this view, the content of a normative system can be represented by a set of statements of the form:

$$\texttt{pre} \rightarrow [\mathbf{a}]\texttt{viol} \tag{7.1}$$

that is, under the conditions expressed in `pre`, the execution of action **a** necessarily leads to a violation state. Such statements can be viewed as constraints on the labeling of transition systems. Restating Formula (7.1), all states which are labelled `pre` are states such that by executing an **a**-transition, states which are labelled `viol` are always reached.[1]

It follows that a set of formulae as Formula (7.1) defines a set of labelled transition systems (i.e., the set of transition systems satisfying the labeling constraints stated in the formulae), and such a set of transition systems can be viewed as representing the content of the normative system specified by those formulae.

Now, within a set of transition systems modeling a set of labeling constraints, transition systems may make violation states possibly reachable by transitions in the systems, and others possibly not. So, from a formal semantics perspective, we can think of the implementation problem as the problem of selecting those transition systems which:

1. Model a given normative system specification in terms of labeling constraints like Formula (7.1);
2. Make some violation states unreachable within the transition system, hence *regimenting* [260] the corresponding norms;

[1] The reader is referred to [42] for more details on the logical study of labelled transition systems.

3. Make other violation states reachable but, at the same time, disincentivizing the agents to execute the transitions leading to those states, for instance by triggering appropriate systems reactions such as sanctioning, thus *enforcing* the corresponding norms [203].

To sum up, normative systems can be studied as sets of labeling constraints on the systems' transitions generated by agents' interaction, and the implementation problem amounts to designing the NMAS according to those transition systems which, on the one hand, model the labeling constraints and, on the other hand, make the agents' access to violation states either impossible (regimentation), or irrational (enforcement). What we mean by the term "irrational" is precisely what is studied by game theory [327], because due to punishments for norm violations the agent is motivated to fulfill the norm. Of course, this does not exclude the possibility that in some circumstances an agent may ignore this incentive and violate the norm. On the contrary, this is one of the reasons why sometimes enforcement is preferred to regimentation, because the creator of the norm cannot foresee all possible circumstances, and it is left to the rational agent to accommodate the local circumstances. The next section moves to the fundamental role that —we think— game theory can play for the analysis of the norm implementation problem.

7.3.2 Norm implementation and games

In a social setting, like the one presupposed by NMAS, action essentially means interaction. Agents' actions have repercussions on other agents which react accordingly. Norm enforcement takes care that agents' actions leading to violation states happen to be successfully deterred, either by a direct system reaction or, as we will see, by means of other agents' actions. The readily available formal framework to investigate this type of social interaction is, needless to say, game theory. The present paper uses the term implementation in the technical sense of implementation theory, i.e., that branch of game theory which, together with mechanism design [245, 251, 252, 303], is concerned with the design of the interaction rules—the "rules of the game" [324] or *mechanisms*—to be put into place in a society of autonomous self-interested agents in order to guarantee that the interactions in the society always result in outcomes which, from the point of view of the society as a whole (or from the point of view of a social designer), are considered most desirable (e.g., outcomes in which social welfare is realized).[2]

In this paper we are going to work with games in extensive forms [327]. Games in extensive form have recently obtained wide attention as suitable tools for the representation of social processes [40]. However, the key advantage for us of choosing games in extensive form is that such games are nothing but tree-like transition

[2] Therefore, when we talk about norm implementation we are not referring to the term implementation in its programming acception like, for instance, in [188].

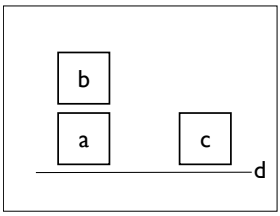

Fig. 7.2 Initial state.

systems. This allows us to directly apply the logic-based representation of norms exposed in Section 7.3.1, thus obtaining a uniform formal background for talking about both norms and games and, hence, for formulating the norm implementation problem in an exact fashion. To ease such exact formulation, we will make use of a simple running example.

7.3.3 Running example: ruling the Blocks World

We assume a multi-agent variant of the blocks world, where agents cannot do concurrent actions (so we do not consider the issue of lifting a block simultaneously). Therefore we assume that the agents have to take actions in turn.

In the standard blocks world scenario [380], the pre- and postcondition specification of the action $\mathtt{move}(\mathsf{a},\mathsf{b},\mathsf{c})$ ("move block a from the top of b to the top of block c") runs as follows:

$$(\mathsf{on}(\mathsf{b},\mathsf{a}) \wedge \mathtt{clear}(\mathsf{c}) \wedge \mathtt{clear}(\mathsf{b}) \wedge \mathtt{turn}(i)) \leftrightarrow \langle \mathtt{move}(\mathsf{b},\mathsf{a},\mathsf{c})(i)\rangle \top \tag{7.2}$$

$$(\mathsf{on}(\mathsf{b},\mathsf{a}) \wedge \mathtt{clear}(\mathsf{c}) \wedge \mathtt{clear}(\mathsf{b}) \wedge \mathtt{turn}(i)) \rightarrow [\mathtt{move}(\mathsf{b},\mathsf{a},\mathsf{c})(i)](\mathsf{on}(\mathsf{b},\mathsf{c}) \wedge \mathtt{clear}(\mathsf{b}) \wedge \mathtt{clear}(\mathsf{a})) \tag{7.3}$$

that is to say: the robotic arm i can execute action $\mathtt{move}(\mathsf{b},\mathsf{a},\mathsf{c})$ iff it is the case that both blocks b and c are clear, and it is its 'turn' to move;and the effect of such action is that block b ends up to the top of block c while block a becomes clear. By permutation of the block identifiers, it follows that action $\mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})$ cannot be executed in the state depicted in Figure 7.2, in which block d represents the floor. We assume background knowledge such that, for example, $\mathsf{on}(\mathsf{b},\mathsf{c})$ implies $\neg\mathtt{clear}(\mathsf{c})$.

Suppose now the robotic arm to be in state of executing action $\mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})$ also if block a is not clear, thus possibly moving a whole stack of blocks at one time. Suppose also that the system designer considers such actions as undesirable. In this case the robotic arm can be considered as an autonomous agent, and the designer as a legislator or policymaker. In order to keep the example perspicuous, the scenario

is limited to one agent, but we can express multiple agents analogously. The action $\texttt{move}(\texttt{a},\texttt{d},\texttt{c})$ would get the following specification. Formula (7.4) does no longer demand $\texttt{clear}(\texttt{a})$, but Formula (5) does not specify the effect when this is the case, i.e., when two or more blocks are moved simultaneously.

$$(\texttt{on}(\texttt{a},\texttt{d}) \wedge \texttt{clear}(\texttt{c}) \wedge \texttt{turn}(i)) \leftrightarrow \langle \texttt{move}(\texttt{a},\texttt{d},\texttt{c})(i) \rangle \top \tag{7.4}$$

$$(\texttt{on}(\texttt{a},\texttt{d}) \wedge \texttt{clear}(\texttt{c}) \wedge \texttt{clear}(\texttt{a}) \wedge \texttt{turn}(i)) \rightarrow [\texttt{move}(\texttt{a},\texttt{d},\texttt{c})(i)](\texttt{on}(\texttt{a},\texttt{c}) \wedge \texttt{clear}(\texttt{a}) \wedge \neg \texttt{clear}(\texttt{c})) \tag{7.5}$$

$$(\texttt{on}(\texttt{a},\texttt{d}) \wedge \texttt{clear}(\texttt{c}) \wedge \neg \texttt{clear}(\texttt{a}) \wedge \texttt{turn}(i)) \rightarrow [\texttt{move}(\texttt{a},\texttt{d},\texttt{c})(i)](\texttt{on}(\texttt{a},\texttt{c}) \wedge \neg \texttt{clear}(\texttt{a}) \wedge \neg \texttt{clear}(\texttt{c}) \wedge \texttt{viol}(i)) \tag{7.6}$$

where $\texttt{viol}(i)$ intuitively denotes a state brought about by agent i which is undesirable from the point of view of the system designer.

Suppose also that the system designer wants to implement the norm expressed by Formula (7.6).[3] The paper tackles this question displaying a number of strategies for norm implementation (Sections 7.4, 7.5, 7.6 and 7.7).

7.3.4 Talking about norms and extensive games in the Blocks World

In this section we bring together the logic-based perspective on norms sketched in Section 7.3.1 with the game-theoretic setting argued for in Section 7.3.2. This will be done in the context of the Blocks World scenario of the previous section. As a result we obtain a very simple modal logic language[4] which suffices to express the properties of extensive games relevant for the purpose of the norm implementation analysis of the Blocks World.

7.3.4.1 Language

The language is the standard propositional modal logic language with n modal operators, where $n = |\texttt{Act}|$, that is, one modal operator for each available transition label. In addition, the non-logical alphabet of the language, consisting of the set of atomic propositions $\texttt{Pr}$ and of atomic actions $\texttt{Act}$, contains at least:

[3] Notice that Formula (7.6) is an instance of Formula (7.1).

[4] For a comprehensive exposition of modal logic the reader is referred to [47].

- Atoms in $\mathtt{Pr}$ denoting game structure: for all agents $i \in I$, $\mathtt{turn}(i), \mathtt{payoff}(i,x)$, labeling those states where it is player's i turn, and where the payoff for player i is x, where x is taken from a finite set of integers. The set of atoms denoting payoffs is referred to as $\mathtt{Pr}^{\mathtt{pay}}$.
- Atoms in $\mathtt{Pr}$ denoting Blocks World states-of-affairs: for all blocks $\mathsf{a},\mathsf{b} \in B$,$\mathsf{on}(\mathsf{a},\mathsf{b})$, $\mathtt{clear}(\mathsf{a})$, labeling those states where block a is on block b, and where block a has no block on it.
- Atoms in $\mathtt{Pr}$ denoting normative states-of-affairs: for all agents $i \in I$, $\mathtt{viol}(i)$, labeling those states where player i has committed a violation.
- Atoms in $\mathtt{Act}$ denoting deterministic transitions: for all agents $i \in I$ and blocks $\mathsf{a},\mathsf{b},\mathsf{c} \in B$: $\mathtt{move}(\mathsf{a},\mathsf{b},\mathsf{c})(i)$, labeling those state transitions where player i moves block a from the top of block b to the top of block c.

The inductive definition of the set of formulae obtained from compounding via the set of Boolean connectives $\{\bot, \neg, \wedge\}$ and the modal connectives $\{\langle \mathbf{a} \rangle\}_{\mathbf{a} \in \mathtt{Act}}$ is the standard one.

7.3.4.2 Semantics

Models are labelled transition systems $m = \langle W, W_{end}, \{R_a\}_{a \in \mathtt{Act}}, \mathcal{I} \rangle$ such that:

- W is a non-empty set of system states;
- $\{R_a\}_{a \in \mathtt{Act}}$ is a family of labelled transitions forming a rooted finite tree, i.e. there is a node such that there is a unique path from this node to all other nodes;
- W_{end} are the leaves of the finite tree;
- $\mathcal{I} : \mathtt{Pr} \longrightarrow 2^W$ is the state labeling function.

The standard satisfaction relation $\models$ between pointed models (m, w) and modal formulae is assumed [47]. In addition, the models are assumed to satisfy the determinism condition, for all $\mathbf{a} \in \mathtt{Act}$:

$$\langle \mathbf{a} \rangle \phi \rightarrow [\mathbf{a}] \phi.$$

Please note that such a condition is typical of the representation of actions within games in extensive form.[5]

Now everything is put into place to formulate with exactness the question that will be addressed in the next sections. Consider a model m as represented in Figure 7.3. State w_1 is assumed to satisfy the relevant Blocks World description of Figure 7.2: $(m, w_1) \models \mathsf{on}(\mathsf{a},\mathsf{d}) \wedge \mathtt{clear}(\mathsf{c}) \wedge \mathtt{clear}(\mathsf{b}) \wedge \neg \mathtt{clear}(\mathsf{a})$.[6] Notice that in the

[5] The reader is referred, for more details, to [41].

[6] To avoid clutter in figures and notation, in what follows forbidden actions (e.g. $\mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})(i)$ at w_1) are denoted by "−", and allowed actions (e.g. $\mathtt{move}(\mathsf{b},\mathsf{a},\mathsf{c})(i)$ at w_1) are denoted by "+". We are confident that this notational simplification will not give rise to misunderstandings.

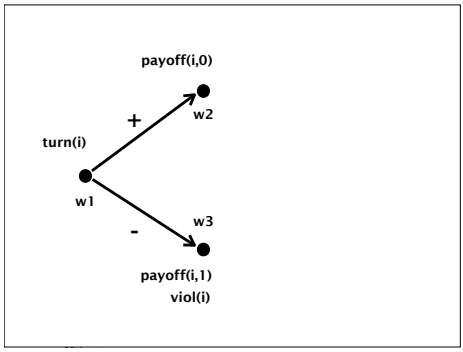

Fig. 7.3 Initial model.

model it is also assumed that agent i leans towards executing the action "−" leading to the `viol`(i)-state which has got a higher payoff. The actions and the violation conditions are given in the norm implementation problem. For example, there may have been an obligation to do +, a prohibition to do −, an obligation to reach state w_2, or a prohibition to reach w_3. Which norm is created is part of the norm synthesis or creation problem, but not part of the norm implementation problem. For the latter problem discussed in this paper, the structure of Figure 7.3.

Consider now a normative specification as represented by formulae like Formula (7.6), together with an initial model (such as the one in Figure 7.3). What are the transformations of the model m, guaranteeing that the agents in the system will comply with the normative specification? This is, in a nutshell, what we are going to investigate in the remainder of the paper.

7.3.5 Two important caveats

Before starting off with our analysis, we find it worth stating explicitly also what this work is not about.

The issue of norm implementation as intended here has already received attention in the literature on MAS in the form of the quest for formal languages able to specify sanctioning and rewarding mechanisms to be coupled with normative systems specifications. An example in this sense—but not the unique one—is [291], where authors are concerned with the development of a whole framework for the formal specification of NMAS. Such a framework is able to capture also norm-implementation mechanisms such as sanctioning and rewarding systems. As our research question discussed in Section 7.1 shows, our aim in this paper differs from all such studies which can be found in the literature. The purpose of the paper is

not to develop a formalism for the specification of one or another mechanism which could be effectively used for implementing norms in MAS. Rather, the paper aims at moving a first step towards the development of a comprehensive formal theory of norm implementation. Such a theory should be able to capture all forms of norm implementation mechanisms highlighting their common features and understanding them all as system transformations.

Finally, we want to stress that the present contribution abstracts completely from the issue concerning the motivating aspect of norms, that is to say, their capacity to influence and direct agents' mental states and actions. We are not assuming here that agents have the necessary cognitive capabilities to autonomously accept or reject norms [116]. To put it yet otherwise, the perspective assumed here is the one of a social designer aiming at regulating a society of agents by just assuming such agents to be game-theoretic agents. We are of course aware of this simplification, which is on the other hand necessary as we are facing the very first stage of the development of a formal theory.

7.4 Making violations impossible

The present sections studies two simple ways of making illegal states unreachable within the system.

7.4.1 Regimentation

Regimentation [260] is the simplest among the forms of implementation. Consider our running example, and suppose the social designer wants to avoid the execution of $\mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})$ by (i) in the case block a is not clear, as expressed in Formula (7.6).

The implementation via regimentation for a transition $\mathbf{a}$ can be represented by a transformation (or update) $m \longmapsto m'$ of the model m into the model m' such that:

$$R_{\mathbf{a}}^{m'} := R_{\mathbf{a}}^{m} - \{(w,w') \mid (m,w) \models \mathtt{pre}_{\mathbf{a}} \ \&\ (m,w') \models \mathtt{viol}(i)\}$$

where $\mathtt{pre}_{\mathbf{a}}$ are the preconditions of the execution of $\mathbf{a}$ leading to a violation. In other words, it becomes in m' impossible to execute a transition with label $\mathbf{a}$ in $\mathtt{pre}_{\mathbf{a}}$-states leading to a violation state.

In the running example, where $\mathbf{a} = \mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})(i)$, such update results in pruning away the edges labeled by "$-$" (i.e., $\mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})(i)$ form the frame of m (Figure 7.4). The regimentation of the prohibition expressed in Formula (7.6) corresponds therefore to the validity of the following property:

$$(\mathtt{on}(a,d) \wedge \mathtt{clear}(\mathsf{c}) \wedge \neg\mathtt{clear}(\mathsf{a}) \wedge \mathtt{turn}(i)) \rightarrow [\mathtt{move}(a,d,c)(i)]\bot$$

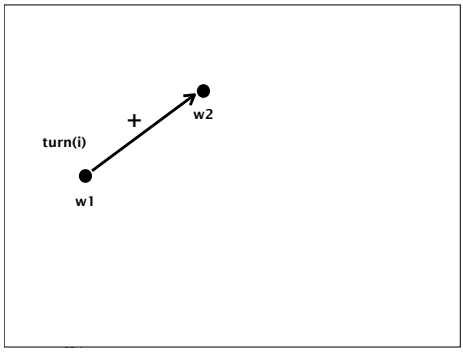

Fig. 7.4 Regimentation.

and hence, by modal logic and some additional background knowledge on the Blocks World:

$$(\mathtt{on}(a,d) \wedge \mathtt{clear(c)} \wedge \neg \mathtt{clear(a)} \wedge \mathtt{turn}(i)) \leftrightarrow \langle \mathtt{move}(a,d,c)(i) \rangle \top$$

which, notice, is a strengthening of Formula (7.4). In other words, regimentation is an update restricting the possibility of actions of the agents by limiting them exactly to the ones generating legal states. It is instructive to notice that the standard Blocks World scenario can be viewed precisely as a result of the regimentation of the normative variant of the scenario which we are considering here.

Within multi-agent systems, regimentation has been the first technique used for norm implementation. A typical example of this is AMELI [167], where all executable actions of agents are actions which are allowed according to the rule of the institutions. A formal semantics for a multi-agent program capturing regimentation is also studied in [125].

7.4.2 Retarded preconditions

Ordinary action logic describes the actions using preconditions and postconditions. If **a** is an action with precondition $\mathtt{pre}_{\mathbf{a}}$ and postcondition $\mathtt{post}_{\mathbf{a}}$ then

$$\mathtt{pre}_{\mathbf{a}} \leftrightarrow \langle \mathbf{a} \rangle \top \tag{7.7}$$

$$\mathtt{pre}_{\mathbf{a}} \rightarrow [\mathbf{a}]\mathtt{post}_{\mathbf{a}} \tag{7.8}$$

express that action **a** can be executed if and only if $\mathtt{pre}_{\mathbf{a}}$ hold (Formula (7.7)) and with the effect expressed by $\mathtt{post}_{\mathbf{a}}$ (Formula (7.8)). So in the standard account of

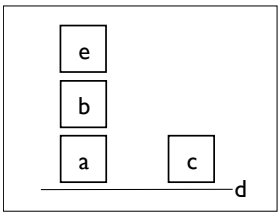

Fig. 7.5 Retarded preconditions. Initial state.

the blocks world, if the world is in the situation as depicted in Figure 7.2, b can be moved on top of c but a cannot be moved.

According to the normative perspective we have assumed in the running example, instead of imposing logically strong preconditions, we state logically weak preconditions for action, which means allow their execution in a wider range of states and assuming indeterminacy. In addition, we label states reached by performing actions as violation states when they are executed under undesirable conditions (see Section 7.3.3). In short, actions are allowed to be executed under circumstances which can possibly lead to violations, but only if the effects are still acceptable. If they are not, then nothing has happened.

These intuitions lead us to introduce, within the framework exposed in Section 7.3.4, two new modal operators: $\langle \phi \mid \mathbf{a} \rangle \psi$ and $[\phi \mid \mathbf{a}] \psi$. The semantics of these new operators is defined as follows:

$$m, w \models [\phi \mid \mathbf{a}] \psi \text{ iff } \forall w' \in W \text{ if } wR_{\mathbf{a}}|_{[\![\phi]\!]}w' \text{ then } m, w' \models \psi$$
$$m, w \models \langle \phi \mid \mathbf{a} \rangle \psi \text{ iff } \exists w' \in W \text{ such that } wR_{\mathbf{a}}|_{[\![\phi]\!]}w' \text{ and } m, w' \models \psi$$

where $[\![\phi]\!]$ denotes, as usual, the truth-set of ϕ and $R_{\mathbf{a}}|_{[\![\phi]\!]}$ is the subset of $R_{\mathbf{a}}$ containing those state pairs (w, w') such that the second element w' of the pair satisfies ϕ.[7] Notice, therefore, that the new modal operators take an action (e.g., $\mathbf{a}$) and a formula (e.g., ϕ) yielding a new complex action type (e.g., $\phi \mid \mathbf{a}$). Such action type corresponds, semantically, to those state transitions which are of the given action type ($\mathbf{a}$) and which end up in the given states (ϕ). So, retarded preconditions are represented, rather than as a formula, as part of an action type. This is in fact natural, as retarded preconditions are a way to further specify an action type.

Note that if we consider only a single update, then it would suffice to introduce $R_{\mathbf{a}}|_{[\![\phi]\!]}$ as the subset of $R_{\mathbf{a}}$ containing pairs (w, w') such that $w' \models \phi$. However, for sequential actions we have multiple updates and this would not suffice. This illustrates that we have a reduction from our logic to the fragment without the dynamic operators, as usually done in update logic (see, e.g., [420]).

[7] It might be instructive to notice that such operators are definable within standard dynamic logic [209] by means of the sequencing operator ; and the test operator ?: $[\phi \mid \mathbf{a}] \psi := [\mathbf{a}; ?\phi]\psi$. However, the full expressivity of dynamic logic is not required given our purposes.

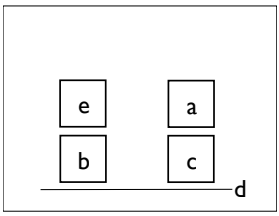

Fig. 7.6 Situation A

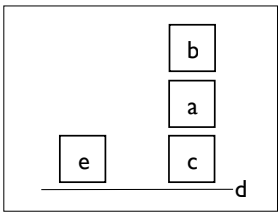

Fig. 7.7 Situation B

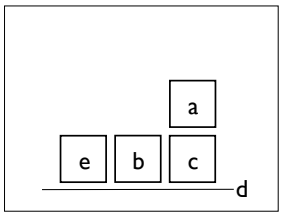

Fig. 7.8 Situation C

By means of this newly introduced operators, we can express that the execution of a given action **a** is possible only under the condition that it has certain precise effects ϕ (Formula (7.9)), and that each time it is executed having such effects ϕ, it also guarantees that ψ holds (Formula (7.10)):

$$\mathtt{pre}_{\mathbf{a}} \rightarrow \langle \mathtt{ret\_pre}_{\mathbf{a}} \mid \mathbf{a} \rangle \top \tag{7.9}$$

$$\mathtt{pre}_{\mathbf{a}} \rightarrow [\mathtt{ret\_pre}_{\mathbf{a}} \mid \mathbf{a}]\, \mathtt{post}_{\mathbf{a}} \tag{7.10}$$

where $\mathtt{pre}_{\mathbf{a}}$ represents the precondition of **a** where the execution of **a** possibly leads to a violation; $\mathtt{ret\_pre}_{\mathbf{a}}$ the postcondition of $\mathtt{pre}_{\mathbf{a}}$ which are tolerated, i.e., its *retarded preconditions*; and $\mathtt{post}_{\mathbf{a}}$ the postconditions of $\mathtt{ret\_pre}_{\mathbf{a}} \mid \mathbf{a}$.

Let us now give an example. Suppose we have the situation depicted in Figure 7.5. We move a, and we might end up with one of the three options in Figures 7.6-7.8. Suppose also that only the situation depicted in Figure 7.6 is tolerable to us. That is, a can be moved on c only if it is slid out carefully from the tower composed by a, b, e. Such tolerance can be expressed by means of *retarded precondition*, that is, a precondition which is evaluated as a result of the action performed. In the example at hand, the execution of action move(a,d,c) is tolerated in the case a is moved from within a tower only if the result of the action yields the situation depicted in Figure 7.6:[8]

$$\mathsf{on}(\mathsf{a},\mathsf{d}) \wedge \mathsf{on}(\mathsf{e},\mathsf{b}) \wedge \mathtt{clear}(\mathsf{c}) \rightarrow \langle \mathsf{on}(\mathsf{e},\mathsf{b}) \mid \mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c}) \rangle \top \tag{7.11}$$

$$\mathsf{on}(\mathsf{a},\mathsf{d}) \wedge \mathsf{on}(\mathsf{e},\mathsf{b}) \wedge \mathtt{clear}(\mathsf{c}) \rightarrow [\neg \mathsf{on}(\mathsf{e},\mathsf{b}) \mid \mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})] \bot \tag{7.12}$$

$$\mathsf{on}(\mathsf{a},\mathsf{d}) \wedge \mathsf{on}(\mathsf{e},\mathsf{b}) \wedge \mathtt{clear}(\mathsf{c}) \rightarrow [\mathsf{on}(\mathsf{e},\mathsf{b}) \mid \mathtt{move}(\mathsf{a},\mathsf{d},\mathsf{c})] \mathsf{on}(\mathsf{a},\mathsf{c}). \tag{7.13}$$

Block a can be moved also in the case it is not clear, provided that this does not change the respective disposition of other blocks b and e (Formula 7.11). If that is not the case, than it will not be possible to move it (Formula 7.12). The effect of the execution of a under the retarded precondition that the stack of b and e is left intact results in a being placed on c (Formula 7.13).

The specification of retarded preconditions for actions can be viewed as a smoothening of regimentation requirements. As shown in the example above, in-

[8] We drop the turn(i) atoms in the following formalization.

stead of regimenting the non-execution of action $\mathtt{move(a,d,c)}$ in case block $\mathtt{a}$ as positioned within a tower, we can express that the execution can be tolerated, provided it gives rise to specific results (Figure 7.6).

In a nutshell, the use of retarded preconditions is typical of situations where the execution of a given action **a** under certain circumstances ϕ can possibly lead to a violation state:

$$\phi \wedge \langle \mathbf{a} \rangle \mathtt{viol}.$$

In such cases, we might not want to impose a regimentation, requiring that:

$$\phi \rightarrow [\mathbf{a}]\bot$$

but we would rather still allow the agent to perform the action, provided that it does not end up in violation states, that is, we allow the execution of the action under the potentially problematic conditions ϕ but only by assuming the retarded precondition $\neg\mathtt{viol}$:

$$\phi \rightarrow \langle \neg\mathtt{viol} \mid \mathbf{a} \rangle \top$$
$$\phi \rightarrow [\mathtt{viol} \mid \mathbf{a}]\bot$$

We conclude spending a few more words on the notion of retarded precondition. Such notion of retarded precondition is implicit in our culture. The saying "you can't argue with success" illustrates that way of thinking. An agent can take action without following the rules and if he is successful then we have to accept it. A major example is Admiral Nelson defying command and defeating the Spanish fleet. He is a hero. Had he failed, he would have been court marshalled.

7.5 Perfect enforcement

Perfect enforcement takes place when the execution of an action leading to a violation state is directly deterred by modifying the payoffs that the agent would obtain from such an execution. The following condition says that the best action does not imply a violation of the norm. It covers both penalties and rewards, or combinations of them.

Let $\mathtt{Pr}^{\mathtt{pay}}$ denote the set of payoff atoms and let $Max(i)$ denote the maximum payoff an agent i gets at a violation end state, if such a state exists. The implementation for i via perfect enforcement with respect to a transition **a**, is a model update changing $m = (W, \{R_a\}_{a\in\mathtt{Act}}, \mathcal{I})$ to $m' = (W', \{R'_a\}_{a\in\mathtt{Act}}, \mathcal{I}')$ as follows:

- $W = W'$;
- $W_{end} = W'_{end}$;
- $\{R_a\}_{a\in\mathtt{Act}} = \{R'_a\}_{a\in\mathtt{Act}}$;

- $\mathcal{I}^m \lceil \mathtt{Pr} - \mathtt{Pr}^{\mathtt{pay}} = \mathcal{I}^{m'} \lceil \mathtt{Pr} - \mathtt{Pr}^{\mathtt{pay}}$, where $\lceil$ denotes the domain restriction operation on functions;
- $\mathcal{I}^{m'} \lceil \mathtt{Pr}^{\mathtt{pay}}$ is such that if $W_{end} \cap -\mathcal{I}(\mathtt{viol}(i)) \neq \emptyset$, then for some payoff atom $\mathtt{payoff}(i, x)$ with $x > Max(i)$ and state $w \in W'_{end}$, $w \in \mathcal{I}'(\mathtt{payoff}(i, x))$.

Notice that the update does not modify the interpretation of atoms which are not payoff atoms nor the frame of the model. What it does is to change $\mathcal{I}$ to a valuation $\mathcal{I}'$ which guarantees that at least one state in the end states of the game which are not violation states (if such states exist), the payoff for i is higher than the payoff in any of the violation states. Intuitively, such an update guarantees that each agent faced with a decision between executing a transition leading to a violation state, and one leading to a legal one, will—if they act rationally from a decision-theoretic perspective—choose for the latter.

A number of different implementation practices can be viewed as falling under this class such as, for instance, fines or side payments. However, the common feature consists in viewing the change in payoffs as infallibly determined by the enforcement, thereby giving rise to perfect deterrence. The next section will show what happens if such an assumption is dropped.

Getting back to our running example, the perfect enforcement of the prohibition expressed in Formula (7.6) results, therefore, in the validity of the following property:

$$\mathsf{on}(\mathsf{a}, \mathsf{d}) \wedge \mathtt{clear}(c) \wedge \mathsf{on}(\mathsf{b}, \mathsf{a}) \wedge \mathtt{turn}(i) \rightarrow ([+]\mathtt{payoff}(i, 1) \wedge [-]\mathtt{payoff}(i, 0))$$

where $+ = \mathtt{move}(\mathsf{b}, \mathsf{a}, \mathsf{c})(i)$ and $- = \mathtt{move}(\mathsf{a}, \mathsf{d}, \mathsf{c})(i)$.

We deem worth stressing again the subtle difference between perfect enforcement and regimentation. While regimentation makes it impossible for the agents to reach a violation state, automatic enforcement makes it just irrational in a decision-theoretic sense. In other words, it is still possible to violate the norm, but doing that would be the result of an irrational choice. As such, perfect enforcement is the most simple form of implementation which leaves the game form (i.e., the frame of the modal logic models) intact. Although the extensive game considered is a trivial one-player game, it should be clear that taking more player into consideration would not be a problem. In such case, the application of solution concepts such as sub-game perfect Nash [327] would become relevant.

In the multi-agent systems literature, perfect enforcement is used to provide a formal semantics to multi-agent programs in [125].

7.6 Enforcers

Commonly, perfect deterrence is hard to realize as each form of sanctioning requires the action of some third-party agent whose role consists precisely in making the

sanctions happen. Enforcement via agents (the enforcers) corresponds to the update of model m to a model m' defining a new game form between a player i and enforcer j. The actions of enforcer j are $\texttt{punish}(i)$ and $\texttt{reward}(i)$. As a result of such an update, the original model m results in a sub-model of m'. The update is defined as follows:

- $W'_{end} = \{(w,n) \mid w \in W_{end} \ \& \ n \in \{1,2\}\}$;
- $W' = W \cup W'_{end}$, that is, each dead end of W is copied twice and added to the domain;
- $\{R'_a\}_{a\in\texttt{Act}} = \{R_a\}_{a\in\texttt{Act}}$, that is, the labeling via $\texttt{Act}$ remains the same;
- $R'_{\texttt{reward}(i)} = \{(w,w') \mid w \in W_{end} \ \& \ w' = (w,1)\}$ and $R'_{\texttt{punish}(i)} = \{(w,w') \mid w \in W_{end} \ \& \ w' = (w,2)\}$, that is, the added transitions are labeled as rewarding and punishing;
- $\mathcal{I}' = \mathcal{I}$ for all states in W;
- $\mathcal{I}'$ for the states in W'_{end} is such that $\forall w,w'$ s.t. $w \in W_{end}$ and $(m',w) \models \texttt{payoff}(i,x))$ and $wR^{m'}_{\texttt{reward}(i)}w' : (m',w') \models \texttt{payoff}(i,y)$ with $x \leq y$; and $\forall w,w'$ s.t. $w \in W_{end}$ and $(m',w) \models \texttt{payoff}(i,x))$ and $wR^{m'}_{\texttt{punish}(i)}w' : (m',w') \models \texttt{payoff}(i,y)$ with $x > y$;

What the definition above states is that the update consists in adding to every dead end in m a trivial game consisting of a binary choice by enforcer j between punishing or rewarding agent i. The result of a reward leaves the payoff of i intact (or it increases it), while the result of a punishment changes i's payoff to a payoff which is lower than the payoff i would have obtained by avoiding to end up in a violation state. In the running example, the action of the enforcer swaps the payoffs of agent i from 0 to 1 or from 1 to 1 in case of a reward; from 1 to 0 or from 0 to 0 in case of a punishment, just like in the case of automatic enforcement.

However, the use of agents as enforcers implies the introduction of a further normative level, since the enforcer can choose whether to comply or not with its role, that is, punish if i defects, and reward if i complies:

$$(\texttt{turn}(j) \wedge \texttt{viol}(i)) \rightarrow [\texttt{reward}(i)]\texttt{viol}(j) \tag{7.14}$$

$$(\texttt{turn}(j) \wedge \neg\texttt{viol}(i)) \rightarrow [\texttt{punish}(i)]\texttt{viol}(j) \tag{7.15}$$

Whether the enforcement works or not, depends on the payoffs of the enforcer j. We are, somehow, back to the original problem of guaranteeing the behavior of an agent (the enforcer in this case) to comply with the wishes of the social designer. The implementation of norms calls for more norms (Figure 7.9). Enforcement via enforcing agents lifts the implementation problem from the primary norms addressed to the agents in the system, to norms addressed to special agents with 'institutionalized' roles.

To the best of our knowledge, the only systematic multi-agent system frameworks addressing norm implementation at the level of enforcement is $\mathcal{M}$oise$^+$ and its variants (e.g., [243]), although not providing a formal semantics for it.

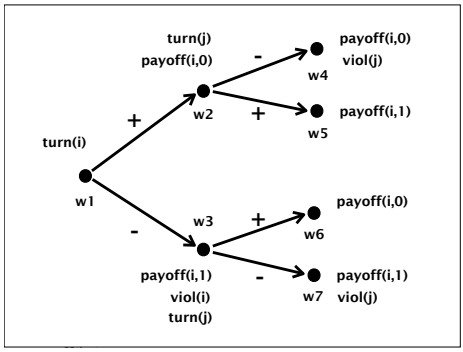

Fig. 7.9 Enforcement norms.

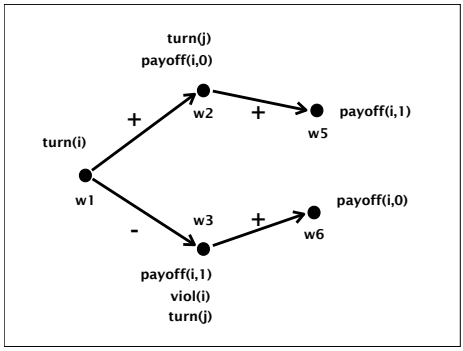

Fig. 7.10 Regimentation of enforcement norms.

7.6.1 Regimenting enforcement norms

At this point, the norms expressed in Formulae (7.14) and (7.15) need implementation. Again, regimentation can be chosen. The result of regimentation of enforcement norms in the running example is depicted in Figure 7.10. Formally, this corresponds to an update $m \longmapsto m'$ of m where:

$$R^{m'}_{\texttt{punish(i)}} = R^{m}_{\texttt{punish(i)}} - \{(w,w') \mid m,w \models \texttt{turn}(j) \wedge \texttt{viol}(i)\ \&\ m,w' \models \texttt{viol}(j)\}$$
$$R^{m'}_{\texttt{reward(i)}} = R^{m}_{\texttt{reward(i)}} - \{(w,w') \mid m,w \models \texttt{turn}(j) \wedge \neg\texttt{viol}(i)\ \&\ m,w' \models \texttt{viol}(j)\}$$

As a result, the enforcer j always complies with what expected from its role. In a way, regimented enforcement can be viewed as an equivalent variant of perfect

enforcement since its result is an adjustment of the payoffs of agent i w.r.t. to the system's norms.

7.6.2 Enforcing enforcement norms

If the payoffs of the enforcer are appropriately set in order for the game to deliver the desired outcome, then the system is perfectly enforced by enforcer j who autonomously complies with the enforcement norms expressed in Formulae (7.14) and (7.15), punishing player i when i commits a violation and rewarding i when i complies (Figure 7.11). In the running example, perfect enforcement of enforcement norms can be defined by a simple update $m \longmapsto m'$ of the interpretation functions of the two models such that:

$$\mathcal{I}^{m'}(\texttt{payoff}(j,0)) = \mathcal{I}^{m}(\texttt{viol}(j))$$
$$\mathcal{I}^{m'}(\texttt{payoff}(j,1)) = W - \mathcal{I}^{m}(\texttt{viol}(j))$$

which results in a perfect match between higher payoffs and legal behavior. Figure 7.12 represents, in strategic form, the extensive game depicted in Figure 7.11 between player i and enforcer j. It is easy to see that the desired outcome in which both i and j comply is the only Nash equilibrium [327]. It goes without saying that much more complex game forms could be devised, and different equilibrium notions could be chosen for norm implementation purposes. It is at this level that a number of concepts and techniques could be imported from Mechanism Design and Implementation Theory [245, 251, 252, 303] to the formal theory of NMAS.

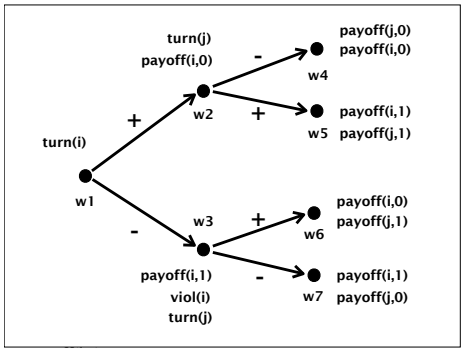

Fig. 7.11 Perfect enforcement.

7.6.3 Who controls the enforcers?

Our analysis clearly shows the paradox hiding behind norm implementation. In order to implement norms, it is likely to need more norms.

The implementation of a set of norms can be obtained either via regimentation or via automatic enforcement or by the specification of an enforcement activity to be carried out by an enforcer. Enforcement specification happens at a normative level, i.e., via adding more norms to the prior set which, in turn, also require implementation. Schematically, suppose X to be the non-empty set of to-be-implemented norms, $Regiment(X)$ to denote the set of norms from X which are regimented or automatically enforced, and $Enforce(X)$ to denote the set of norms containing X together with all the norms specifying the enforcement of X ($X \subseteq Enforce(X)$). The implementation of S is the enforcement of the norms in S which are not regimented: $Implement(X) = Enforce(X \setminus Regiment(X))$.

In other words, to implement a set of norms amounts to implement the set of unregimented norms together with their enforcement. These observations clearly suggest that the implementation of a set of norms yields a set of norms. Somehow, it is very difficult to get rid of norms when trying to implement them. The only possibility is via full regimentation or automatic enforcement. If $Regiment(X) = X$ then there is no norm left to be implemented. Instead if $Regiment(X) \subset X$ then $\emptyset \subset Implement(S)$, which means that the implementation operation should be iterated on $Implement(X)$. In principle, such iteration is endless, unless there exists a final implementation level whose norms are all regimented or automatically enforced.

7.7 Implementation via norm change

This section concerns the ways of obtaining desired social outcomes by just modifying the set of norms of the system. The formal analysis of such phenomena, which is pervasive in human normative systems, is strictly related with the formal study of counts-as [202] and intermediate concepts [287].

As an example, consider the model m' obtained via the update of the initial model m corresponding to perfect enforcement (Figure 7.11). Suppose now the social designers wants to punish player i no matter what it does. One way for doing this

i \ j	-	+
-	(1,0)	(0,1)
+	(0,0)	(1,1)

Fig. 7.12 Enforcement of the Blocks World scenario in strategic form

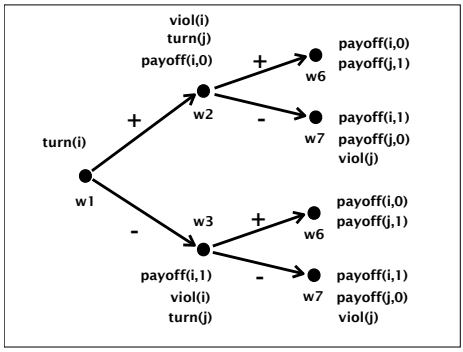

Fig. 7.13 Implementation via norm change.

would be to go back to the initial model m, to replace the enforcer norms expressed in Formulae (7.14) and (7.15) by the following norm:

$$\texttt{turn}(j) \rightarrow [\texttt{reward}(i)]\texttt{viol}(j) \tag{7.16}$$

and then update m to implement the norm expressed in Formula (7.16), for instance via perfect enforcement.

A much quicker procedure would consist in updating model m' trying to inherit its implementation mechanism. This can be done by simply modifying the extension of atom $\texttt{viol}(i)$ in order for it to include state w_2, thereby automatically triggering the enforcement norms expressed in Formulae (7.14) and (7.15). As a result, the enforcement mechanism in place in model m' are imported "for free" by simply changing the meaning of $\texttt{viol}(i)$ (Figure 7.13). As you can see, the payoffs for enforcer j are different from Figure 7.11.

The update of the extension of $\texttt{viol}(i)$ can be obtained, for instance, by adding the following norm to the system:

$$\texttt{on}(\texttt{b},\texttt{a}) \wedge \texttt{clear}(\texttt{b}) \wedge \texttt{clear}(\texttt{c}) \wedge \texttt{turn}(i) \rightarrow [\texttt{move}(\texttt{b},\texttt{a},\texttt{c})(i)]\texttt{viol}(i) \tag{7.17}$$

To put it otherwise, such procedure exploits the nature of $\texttt{viol}(i)$ as an intermediate concept occurring as precondition of other norms. In this case the norms involved are the enforcement norms expressed in Formulae (7.14) and (7.15).

7.8 Related work

In this section we consider whether existing work in normative multi-agent systems is able to answer the equation discussed in the introduction.

BDI : Agent Programming = ? : NMAS Programming.

Since BDI-CTL [110] is used as a formal specification and verification language for agent programming, an obvious candiate for our question mark is an extension of this language with deontic concepts such as obligations and permissions, called BOID-CTL [87, 88]. Such a logic is simply a modal combination of an agent logic and a modal deontic logic. The drawback of this approach is that the norms are nor represented explicitly.

The first candidate for the question mark is Tennenholtz and Shoham's game-theoretic approach to artificial social systems. However, the central research question of their work [400, 401, 412] consists in studying the emergence of desirable social properties under the assumption that a given social law is followed by the agents in the society at hands. The problem of how a social law can be implemented in the society is not discussed.

Another obvious candidate for the question mark is a theory of normative systems [5]. The key feature of normative systems is that they make norms explicit in such a way that we can say, at a given state of the system, whether a norm is active, in force, violated, and so on [417]. See [206] for an up to date review on the distinction between a theory of normative systems and deontic logic, and the challenge to bridge the two. A theory of normative systems is useful for norm representation and reasoning, but not for the representation of aspects such as the multi-agent structure of a normative system.

A third candidate is Boella and van der Torre's game-theoretic approach to normative multi-agent systems, which studies the more general problem of norm creation [48, 53]. For example, the introduction of a new norm with sanctions is modeled as enforceable norms in artificial social systems as the choice among various strategic games [52]. They focus in particular on the enforcement of norms using enforcers, and discuss the role of procedural norms to motivate the enforcers [54]. They consider the creation of a new norm into a system of norms, whereas in this paper we do not consider the effect of norm implementation on existing norms. They argue that the infinite regression of enforcers can be broken if we assume that enforcers control each other and do not cooperate [52]. Since they use strategic rather than extensive games they cannot distinguish some subtle features of implementation such as retarded preconditions. Moreover, they do not give a procedure to go from a norm to its implemented system. Finally, they do not consider other methods than sanctioning and rewarding to implement their norms. They do consider also cognitive extensions of their model, which we do not consider in this paper. See [53] for a detailed discussion on their approach.

There are many organizational and institutional theories, such as the ones proposed in [202], and there is a lot of work on coordination and the environment [123, 366]. Institutions are built using constitutive norms defining intermediate concepts. However, this work is orthogonal to the work presented in this paper in as much as, although sporadically addressing one or another form of implementation, it never aims at laying the ground of an overarching formal framework.

7.9 Conclusions

Aim of the paper is to illustrate how the issue of norm implementation can be understood in terms of transformations (updates) performed on games in extensive forms. The paper has sketched some of such updates by means of a toy example, the blocks world, and mapped them to norm implementation strategies, such as regimentation, automatic enforcement, enforcement via enforcers, and implementation via norm change. The full logical analysis (e.g., in a dynamic logic setting) of the update operations sketched here is future work. Such an analysis will make some intricacies of implementation explicit, such as, for instance the fact that by implementing new norms, the implementation of other norms might end up being disrupted.

Moreover, we introduce two views on representing forbidden actions, the classical one in which the precondition has to be satisfied before the action can be executed, and one based on so-called retarded preconditions. The two views coincide if the language allows for action names, and we can include as part of the state a list of which actions are allowed in this state. This can be formalised by the predicate *allowed*(X), where X are names for actions. The *allowed*(X) predicate can be part of the preconditions of X. We can use the feedback arrows of retarded preconditions in Kripke models to change accessibility. This will implement the severed connections in the diagrams, and the semantics would then be *reactive Kripke models.* Consider for example the restriction "you should not take any action three times in a row." With retarded preconditions, we can do a "roll-back" when the action occurs three times in a row, whereas with regimentation we have to predict whether the action is going to be executed three times rather than two or four times. A further comparison of the two views is topic for further research.

Finally, topics for further research are also the development of a more detailed classification of norm implementation methods, the application of retarded preconditions to the analysis of ambiguous norms.

Acknowledgments

The authors would like to thank the reviewers of the volume for their helpful comments. Davide Grossi wishes to acknowledge support by *Ministère de la Culture, de L'Enseignement Supérieur et de la Recherche, Grand-Duché de Luxembourg* (grant BFR07/123) and by *Nederlandse Organisatie voor Wetenschappelijk Onderzoek* (VENI grant 639.021.816).

Chapter 8
A Verification Logic for GOAL Agents

K.V. Hindriks

Abstract Although there has been a growing body of literature on verification of agents programs, it has been difficult to design a verification logic for agent programs that fully characterizes such programs and to connect agent programs to agent theory. The challenge is to define an agent programming language that defines a computational framework but also allows for a logical characterization useful for verification. The agent programming language GOAL has been originally designed to connect agent programming to agent theory and we present additional results here that GOAL agents can be fully represented by a logical theory. GOAL agents can thus be said to execute the corresponding logical theory.

K.V. Hindriks
Delft University of Technology, The Netherlands e-mail: k.v.hindriks@tudelft.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_8,

8.1 Introduction

As technology for developing agent systems is becoming more mature, the availability of techniques for verifying such systems also becomes more important. Such techniques do not only complement tools for debugging agent systems but may also be used to supplement the techniques available for debugging agents. For example, model checking techniques may be used to find *counter examples* that show that an agent system does not satisfy a particular property. A counter example produces a run of a system that violates a property and as such indicates what is wrong with an agent. Program model checking discussed in [66] is an approach that supports this type of verification. It involves the construction of a semantic model M that correctly represents an agent's execution and that can be used to check satisfaction of a property φ, i.e. $M \models \varphi$. The key problem that needs to be solved to be able to use model checking for verification concerns the *efficient* construction of (part of) a model M from a given agent system that is sufficient for verifying this system.

Model checking is one approach to verifying agents. An alternative approach to verifying agents involves the use of *deduction*. This approach assumes that a *logical theory* of an agent is available. The task of verifying that an agent satisfies a particular property φ amounts to deducing φ from the given theory T, i.e. $T \vdash \varphi$. The key problem that needs to be solved to be able to use deduction as a verification tool concerns the construction of a corresponding *logical theory* T from a given agent system. It is the goal of this chapter to introduce such a theory for the Goal agent programming language [221].

Verification techniques based on deduction have been widespread in Computer Science and have been provided for a broad range of programming languages. The programming constructs and the structure of programs in a programming language often naturally give rise to an associated *programming logic*. This has been particularly true for imperative programming languages but also for concurrent programming languages [23, 296, 298].

The verification approach presented here for Goal consists of two parts. First, an operational semantics that provides a model for executing agent programs is defined. This provides a computational framework that specifies how Goal agents are to be executed. Second, a logic for verification is introduced and it is shown that the logical semantics corresponds with the operational semantics. It is our aim in this chapter to stay as close as possible to the actual implementation of the Goal language, although we do abstract away a number of features that are present in the interpreter for the language and focus on single agents. In particular, we have provided a semantics for logic programs as part of the operational semantics to model the Prolog engine that is used in the implementation. Similarly, we have aimed for a verification logic which semantics corresponds in a precise sense with the operational semantics and can be used to fully characterize agent programs. As we will show, basic Goal agent programs discussed here may be mapped into a corresponding logical theory by means of a straightforward translation scheme that

fully characterizes the runs of these programs. This result shows that GOAL agent programs may be perceived as executing the corresponding logical theory.

8.2 Related work

There is a growing body of literature on deductive verification of rational agents and agent programs that are based on the Belief-Desire-Intention metaphor. The work related most to our approach concerns logics for the languages 3APL [223] and its successor 2APL [122], and work on ConGolog [193] - a closely related language to 3APL (cf. [225]). The CASL framework [395], also discussed in this volume, that can be viewed as an extension of the situation calculus, also aims at defining a logical framework for specifying rational agents.

Early work on designing a verification logic for 3APL is reported in [216] and introduces a dynamic logic to reason about 3APL programs without self-modifying reasoning rules. The logic assumes that such programs terminate, which derives from the use of a dynamic logic [210], but also accounts for the use of free variables in the execution of 3APL programs. In [6], a logic for reasoning about agent programs in a simplified version of 3APL is introduced, called SimpleAPL. [6] presents a propositional logic to reason about the core features of 3APL, including beliefs and goals, which is proven sound and complete. Finally, [7] discusses a logic for reasoning about the deliberation cycle of an agent in 3APL. This paper addresses reasoning at another level, the execution strategy of the interpreter, rather than the execution of actions and action selection by the agent itself.

[288] presents a Hoare-style proof system for verifying Golog programs, a subset of the ConGolog language, which is proven sound and complete. The work is in many ways similar to that discussed in the previous paragraph. The logic and aims are similar in various respects, but agents in Golog do not have explicit beliefs and goals. The latter restriction has motivated the extension of the basic Golog framework with explicit knowledge and goal operators in CASL [395]. CASL extends the situation calculus with a semantics for such operators using situations in a way similar to how modal worlds are used in classical modal logic. The approach, however, is very expressive allowing for quantification over formulas (as terms) and it is less clear what the computational properties of the framework are.

In previous work on GOAL [60], a verification framework for GOAL agents has been introduced that consists of two parts: A Hoare-style logic for reasoning about actions and a temporal logic for reasoning about runs of such agents. This framework allows for the verification of GOAL agents and has been related to Intention Logic [217]. The work presented here differs in various ways from [60]. First, here we use a temporal logic for reasoning about actions and do not introduce Hoare-style axioms. Second, we show that the resulting logic can be used to fully characterize GOAL agents. Third, the logic allows for quantification and is a first-order verification logic.

8.3 The Agent Programming Language Goal

The agent programming language Goal is presented here by defining its *operational* semantics. The section is organized as follows. In 8.3.1 we very briefly informally introduce the key concepts that make up a Goal agent program. Goal agents derive their choice of action from their beliefs and goals and need a knowledge representation language to represent these which is introduced in 8.3.2. As Goal agents derive what to do next from their mental state, we then continue by introducing the semantics of mental states in 8.3.3. Using the mental state semantics, the meaning of a Goal agent program is specified using *structural operational semantics* [340]. The operational semantics determines which *transitions* from one mental state to another can be made. It makes precise how the mental state of a Goal agent changes when it performs an action.

8.3.1 Goal *Agent Programs*

A Goal agent program consists of the agent's *knowledge*, *beliefs*, its *goals*, a set of *action rules* and a set of *action specifications*. Other features present in the language for e.g. percept handling, modules and communication are not discussed here. For a more thorough and comprehensive introduction to the language see [221].[1]

The knowledge, the beliefs and the goals of an agent are specified *declaratively* by means of a knowledge representation language, which facilitates the design of agent programs at the *knowledge level* [321]. The knowledge of a Goal agent is assumed to be static and does not change over time. The knowledge, beliefs and goals of an agent define the agent's *mental state*. Goal does not commit to any particular knowledge representation technology but for purposes of illustration we will use a variant of PDDL here [192].[2] This has the additional benefit that PDDL action specficiations with conditional actions are supported, and this variant of Goal is able to support the full expressivity of ADL action specifications [336]. A Goal agent derives its choice of action from its beliefs and goals. It does so by means of action rules of the form **if** ψ **then** $a(\mathbf{t})$ where ψ is a condition on the mental state of the agent and $a(\mathbf{t})$ is an action the agent can perform. Whenever the mental state condition ψ holds the corresponding action $a(\mathbf{t})$ is said to be an *option*. At any time, there may be multiple options from which the agent will select one *nondeterministically*.

[1] The reader is referred to [215] for a semantics of modules and [220] for a semantics of communication.

[2] The first-order logic variant of PDDL that we will discuss includes so-called axioms for derived predicates. This variant has been implemented as one of the options for choosing a knowledge representation language in Goal. A programmer can also choose to use Prolog, for example. Although the first-order language presented is richer than that of Prolog, the fragment discussed here can be compiled into Prolog (cf. [289]).

8.3.2 Knowledge Representation Language

We will use a first-order language $\mathcal{L}_0$ for representing the knowledge, beliefs and goals of an agent.[3] $\mathcal{L}_0$ is built using a vocabulary that consists of a *finite* sets of predicates $\mathcal{P}$ with typical elements p, function symbols $\mathcal{F}$ with typical elements f, constant symbols C with typical elements a, b, c, and an infinite supply of variables $\mathcal{V}$ with typical elements x, y, z. We also assume that $\mathcal{L}_0$ includes equality $=$. The set of predicates $\mathcal{P}$ consists of two disjoint sets of so-called *basic* predicates $\mathcal{B}$ and *derived* predicates $\mathcal{D}$, such that $\mathcal{P} = \mathcal{B} \cup \mathcal{D}$ and $\mathcal{B} \cap \mathcal{D} = \emptyset$. The distinction is used to differentiate predicates that may be updated by actions from those that cannot be updated so. The idea is that basic predicates may be updated whereas derived predicates may only be used to define additional concepts that are defined in terms of the more basic predicates.

Definition 8.1. *(Syntax of $\mathcal{L}_0$)*

$$\begin{array}{lll} t \in \mathcal{T} & ::= & x \mid c \mid f(\mathbf{t}) \\ \phi \in \mathcal{L}_0 & ::= & t = t \mid p(\mathbf{t}) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \forall x(\phi) \end{array}$$

A *term* t is either a variable $x \in \mathcal{V}$, a constant $c \in C$, or a function symbol f applied to a vector $\mathbf{t}$ of terms of the appropriate arity. Vectors of variables and terms are denoted by bold face $\mathbf{x}$ respectively $\mathbf{t}$. Formulas $p(\mathbf{t})$ with $p \in \mathcal{P}$ are called *atoms*. Atoms $p(\mathbf{t})$ or their negations $\neg p(\mathbf{t})$ are also called *literals*. Literals l and $\neg l$ are said to be *complementary*. As usual, $\phi \rightarrow \phi'$ is an abbreviation for $\neg\phi \vee \phi'$, and $\exists x(\phi)$ abbreviates $\neg\forall x(\neg\phi)$. We write $\phi[\mathbf{x}]$ to indicate that all free variables of ϕ occur in the vector $\mathbf{x}$. A formula that does not contain free variables is said to be *closed*. Closed formulas without quantifiers, i.e. formulas without any variables, are also said to be *ground*. The set of all *ground* atoms of the form $p(\mathbf{t})$ is denoted by F. The subset $F_b \subseteq F$ consists of all atoms of the form $p(\mathbf{t})$ with $p \in \mathcal{B}$, and, similarly $F_d \subseteq F$ consists of all atoms $p(\mathbf{t})$ with $p \in \mathcal{D}$. Elements from F_b (F_d) are also called *basic (derived) facts*, and basic (derived) facts and their negations are called *basic (derived) literals*. Finally, we use $\forall(\phi)$ to denote the universal closure of ϕ.

The distinction between basic and derived predicates that is made here is used to distinguish basic facts about an environment from conceptual or domain knowledge that can be defined in terms of these basic facts. The use of such defined predicates facilitates programming and reduces the size of the program. We adopt the definition of *derived predicate axioms* and related definitions below from [413].

Definition 8.2. *(Derived Predicate Axiom)*
A *derived predicate axiom* is a formula of the form $\forall\mathbf{x}(\phi[\mathbf{x}] \rightarrow d(\mathbf{x}))$ with $d \in \mathcal{D}$.[4]

[3] This language is referred to as a *knowledge representation language* traditionally, even though it is also used to represent the goals of an agent.

[4] We do not allow terms in the head of a derived predicate axiom here, mainly because it simplifies the presentation and definition of *completion* below.

In our setting, the antecedent ϕ of such an axiom may not contain occurrences of other derived predicates that "depend on" the definition of d.[5] Technically, the requirement is that a set of derived predicate axioms needs to be *stratified*. Before we define stratification it is useful to introduce the notion of a negated normal form. A formula ϕ is in *negated normal form* if all occurrences of negations occur directly in front of atoms. For example, $\forall x(\neg p(x) \vee (q(x) \wedge r(x)))$ is in negated normal form but $\neg\exists x(\neg(p(x) \wedge q(x)))$ is not because negation occurs in front of the existential quantifier and in front of a conjunction. We remark here that this definition assumes that all implications $\rightarrow$ have been expanded into their unabbreviated form. That is, occurrences of, for example, $(p(x) \wedge q(x)) \rightarrow r(x)$ are not allowed and must be replaced with the negated normal form of, in this case, $\neg(p(x) \wedge q(x)) \vee r(x)$, i.e. $\neg p(x) \vee \neg q(x) \vee r(x)$. It is clear that each formula $\phi \in \mathcal{L}_0$ can be transformed into an equivalent formula in negated normal form and we write $NNF(\phi)$ to denote this formula.

Definition 8.3. *(Stratified Derived Predicate Axiom Set)*
A set of derived predicate axioms is called *stratified* iff there exists a partition of the set of derived predicates $\mathcal{D}$ into (non-empty) subsets $\{\mathcal{D}_i, 1 \leq i \leq n\}$ such that for every $d_i \in \mathcal{D}_i$ and every axiom $\forall\mathbf{x}(\phi[\mathbf{x}] \rightarrow d_i(\mathbf{x}))$ we have that:

- if $d_j \in \mathcal{D}_j$ occurs positively in $NNF(\phi)$, then $j \leq i$.
- if $d_j \in \mathcal{D}_j$ occurs negated in $NNF(\phi)$, then $j < i$.

The semantics of $\mathcal{L}_0$ is defined relative to a state S of basic facts and a set of derived facts D. We first present this semantics and then show how the set D of derived facts can be obtained from a set of basic facts and a stratified axiom set. The *closed world assumption* applies, so any ground positive literal not in S is assumed to be false.

Definition 8.4. *(Truth conditions)* Let $S \subseteq F_b$ be a set of basic facts and $D \subseteq F_d$ be a set of derived facts. The truth conditions of *closed* formulas from $\mathcal{L}_0$ are defined by:

$$
\begin{array}{lll}
\langle S, D\rangle \models p(\mathbf{t}) & \text{iff} & p(\mathbf{t}) \in S \cup D \\
\langle S, D\rangle \models \neg\phi & \text{iff} & \langle S, D\rangle \not\models \phi \\
\langle S, D\rangle \models (\phi_1 \wedge \phi_2) & \text{iff} & \langle S, D\rangle \models \phi_1 \text{ and } \langle S, D\rangle \models \phi_2 \\
\langle S, D\rangle \models (\phi_1 \vee \phi_2) & \text{iff} & \langle S, D\rangle \models \phi_1 \text{ or } \langle S, D\rangle \models \phi_2 \\
\langle S, D\rangle \models \forall x(\phi) & \text{iff} & \langle S, D\rangle \models \phi[t/x] \text{ for all ground } t \in \mathcal{T} \\
\langle S, D\rangle \models \exists x(\phi) & \text{iff} & \langle S, D\rangle \models \phi[t/x] \text{ for some ground } t \in \mathcal{T}
\end{array}
$$

We have assumed that all terms refer to different objects, which is also known as the *unique names assumption*. In addition, it is assumed that all objects are named by some term. These are common assumptions in logic programming, and in PDDL. By making these assumptions a substitutional interpretation of quantifiers can be used as we have done in Definition 8.4.

[5] More precisely, recursion through negation is not allowed.

Formulas with free variables may be used to compute answers, i.e. substitutions. Substitutions are used to instantiate variables to values in the domain, i.e. bind variables to closed terms.

Definition 8.5. *(Substitution)*
A *substitution* is a mapping from variables $\mathcal{V}$ to *closed* terms in $\mathcal{T}$.

For details of what it means to apply a substitution θ to an expression e with result $e\theta$ see [289]. $e\theta$ is also called an *instance* of e. We use $\theta[t/x]$ to denote the substition σ such that $\sigma(x) = t$ and $\sigma(y) = \theta(y)$ for all $y \neq x$ in the range of θ.

The semantics of open formulas $\phi \in \mathcal{L}_0$ is defined by $\langle S, D\rangle \models \phi$ iff there is a substitution θ such that $\phi\theta$ is closed and $\langle S, D\rangle \models \phi\theta$.

In the definition of the semantics of $\mathcal{L}_0$ we have assumed that the set of derived facts D was given. Intuitively, we can derive $d(\mathbf{t})$ using axiom $a = \forall \mathbf{x}(\phi \rightarrow d(\mathbf{x}))$ if we have $\langle S, D\rangle \models \phi[\mathbf{t}]$, and add $d(\mathbf{t})$ to D if not already present; we write $[\![a]\!](S, D) = \{d(\mathbf{t}) \mid \langle S, D\rangle \models \phi[\mathbf{t}], \mathbf{t} \text{ is ground}\}$ to denote these consequences. $[\![a]\!](S, D)$ yields all immediate consequences of a stratified axiom set given that a pair $\langle S, D\rangle$ has been fixed.

Then the set of consequences of an axiom set A can be computed as follows, assuming that we have a stratification $\{A_i, 1 \leq i \leq n\}$ of A:

$$[\![A]\!]_0(S) = \emptyset, \text{ and, for all } 1 \leq i \leq n\text{:}$$
$$[\![A]\!]_i(S) = \bigcap \left\{ D \mid \bigcup_{a \in A_i} [\![a]\!](S, D) \cup [\![A]\!]_{i-1}(S) \subseteq D \right\}$$

The set of all derived facts, written $[\![A]\!](S)$, that can be obtained from A then is defined as $[\![A]\!]_n(S)$. Using this set we can define the consequences of a (belief) state S relative to a set of derived predicate axioms A as follows.

Definition 8.6.

$$S \models_A \phi \text{ iff } \langle S, [\![A]\!](S)\rangle \models \phi$$

The semantics of axioms has been defined as a fixed point above. It is well-known, however, that this semantics corresponds with a logical semantics of the *completion* of an axiom set. See, for example, the discussion of logic programming for "unrestricted" programs in [289]. As this equivalence is useful for showing that the verification logic introduced below can be used to characterize Goal agents, we briefly discuss the key results that we need. Details can be found in the Appendix.

We first introduce the *completion comp*(A) of a *finite* stratified axiom set A. Intuitively, the completion *comp*(A) replaces implications with equivalences [17, 289]. We assume that with each derived predicate $d \in \mathcal{D}$ at least one axiom is associated. Then, in our setting, completion can be defined as follows.

Definition 8.7. *(Completion)*
Let A be a finite stratified axiom set. Then the *completion comp*(A) of that set is obtained by applying the following operations to this set:

1. For each derived predicate $d \in \mathcal{D}$, collect all associated axioms of the form $\forall \mathbf{x}(\phi_1 \rightarrow d(\mathbf{x})), \ldots, \forall \mathbf{x}(\phi_n \rightarrow d(\mathbf{x}))$. Replace these axioms by:

$$\forall \mathbf{x}((\phi_1 \vee \ldots \vee \phi_n) \rightarrow d(\mathbf{x}))$$

   Note that variables may need to be renamed to ensure that all axioms for d have $d(\mathbf{x})$ as their head with unique variables $\mathbf{x}$.
2. For each of the formulas obtained in the previous step, replace $\rightarrow$ with $\leftrightarrow$, i.e. replace each formula $\forall \mathbf{x}(\phi \rightarrow d(\mathbf{x}))$ by $\forall \mathbf{x}(\phi \leftrightarrow d(\mathbf{x}))$.

It is well-known that the completion of a stratified axiom set is consistent [289].

Definition 8.8. *(Answer, Correct Answer)*
Let S be a set of ground atoms and A be a stratified axiom set. A substitution θ is an *answer* for ϕ with respect to S and A if θ is a substitution for free variables in ϕ and $S \models_A \forall(\phi\theta)$. A substitution θ is a *correct* answer with respect to S and A if $comp(A) \cup S \models_c \forall(\phi\theta)$ and θ is an answer for ϕ. Here, $\models_c$ refers to the usual consequence relation for *classical* first-order logic.

The completion of a stratified axiom set defines the meaning of such a set in terms of classical first-order semantics. It shows that a *declarative reading* can be imposed on a stratified axiom set. It may moreover be used to verify that the semantics of Definition 8.4 is well-defined.

Theorem 8.1. (Correctness)
Let S be a set of basic facts, A be a stratified axiom set, and θ be an answer for ϕ with respect to S and A. Then θ is a correct answer. That is, we have:

$$S \models_A \forall(\phi\theta) \;\; \textit{iff} \;\; comp(A) \cup S \models_c \forall(\phi\theta)$$

Proof. See the Appendix.

8.3.3 Mental States

The knowledge representation language $\mathcal{L}_0$ is used by GOAL agents to represent their knowledge, beliefs and goals. We first discuss knowledge and beliefs. The difference between knowledge and beliefs is based on the distinction between derived and basic predicates discussed above. Knowledge is assumed to be *static* and concerns conceptual and domain knowledge which is defined using derived predicate axioms. Beliefs may change and represent the basic facts the agent believes to be true about the environment. Accordingly, the knowledge base maintained by a GOAL agent is a set of stratified derived predicate axioms as defined above, and the belief base is a set of ground atoms that only use basic predicates.

Although it is common in planning to allow complex goal descriptions, in contrast with typical planning problems [192] the goals maintained by a GOAL agent may change over time. Moreover, a GOAL agent needs to be able to inspect its goals and therefore it is important that the goal base can be efficiently queried. For these reasons, the goal base consists of conjunctions of ground atoms here. For example, $p(a) \wedge q(b)$ and $r(a) \wedge r(b) \wedge r(c)$ may be part of a goal base but $\neg p(c)$ and $\exists x(q(x) \rightarrow r(x))$ may not. It is clear that a conjunction of ground atoms can be identified with the set of corresponding atoms and we will abuse notation here and will also denote the corresponding set of ground atoms by means of a conjunction. This will allow us to write $p(a) \wedge q(b) \subseteq F$ to denote that $p(a)$ and $q(b)$ are in the set F of facts. We will make use of this below in the definition of the semantics of mental state conditions.

Definition 8.9. *(Mental State)*
A *mental state* is a triple $\langle K, \Sigma, \Gamma \rangle$ where K is a finite, stratified derived predicate axiom set, called a *knowledge base*, $\Sigma \subseteq F_b$ is a *belief base* that consists of a finite set of basic facts, and $\Gamma \subseteq 2^{F_b}$ is a *goal base* that consists of a finite set of finite subsets (or, conjunctions) of basic facts. Finally, the following *rationality constraint* is imposed on mental states:

$$\forall \gamma \in \Gamma : \Sigma \not\models_K \gamma$$

This constraint excludes mental states where a goal in the goal base is believed to be achieved. This constraint imposed on mental states is motivated by the principle that agents should not invest resources into achieving goals that have already been achieved. Goals thus are viewed as *achievement goals*, i.e. states that the agent wants to realize at some future moment.

It is usual to impose various *rationality constraints* on mental states [60]. These constraints typically include that (i) the knowledge base combined with the belief base is consistent, that (ii) individual goals are consistent with the knowledge base, and that (iii) no goal in the goal base is believed to be (completely) achieved. Constraint (iii) is part of the definition of a mental state but we do not need to impose the first two constraints explicitly as these follow by definition; both the belief base and goal base consist of basic facts only, and the knowledge base only consists of rules for derived predicates. Note that although goals cannot be logically inconsistent it is still possible to have conflicting goals, e.g. $on(a,b)$ and $on(b,a)$ in a Blocks World where one block cannot be simultaneously on top of and below another block.

In order to select actions an agent needs to be able to inspect its mental state. In GOAL, an agent can do so by means of *mental state conditions*. Mental state conditions are conditions on the mental state of an agent, expressing that an agent believes something is the case, has a particular goal, or a combination of the two. Special operators to inspect the belief base of an agent, we use **bel**(φ) here, and to inspect the goal base of an agent, we use **goal**(φ) here, are introduced to do so. In addition, a special operator **o-goal**(ϕ) will be useful later and represents that ϕ is the "only goal" of an agent. This operator will allow us to introduce 'successor state axioms'

for goals below. We allow boolean combinations of these basic conditions but do not allow the nesting of operators. Basic conditions may be combined into a conjunction by means of $\wedge$ and negated by means of $\neg$. For example, $\mathbf{goal}(\varphi) \wedge \neg \mathbf{bel}(\varphi)$ with $\varphi \in \mathcal{L}_0$ is a mental state condition, but $\mathbf{bel}(\mathbf{goal}(\varphi))$ which has nested operators is not.

Definition 8.10. *(Syntax of Mental State Conditions)*
The language $\mathcal{L}_\Psi$ of mental state conditions, with typical elements ψ, is defined by:

$$\begin{array}{lll} \phi & ::= & \text{any element from } \mathcal{L}_0 \\ \psi \in \mathcal{L}_\Psi & ::= & \mathbf{bel}(\phi) \mid \mathbf{goal}(\phi) \mid \mathbf{o\text{-}goal}(\phi) \mid \psi \wedge \psi \mid \neg\psi \end{array}$$

Note that we allow variables in mental state conditions, i.e. a mental state condition ψ does not need to be closed. A mental state condition with free variables can be used in an agent program to retrieve particular bindings for these free variables. That is, mental state conditions can be used to compute a *substitution*.

The next step is to define the semantics of *mental state conditions*. The meaning of a mental state condition is derived from the mental state of the agent. A belief condition $\mathbf{bel}(\phi)$ is true whenever ϕ follows from the belief base combined with the knowledge stored in the agent's knowledge base. The meaning of a goal condition $\mathbf{goal}(\phi)$ is slightly different from that of a belief condition. Instead of simply defining $\mathbf{goal}(\phi)$ to be true whenever ϕ follows from *all* of the agent's goals (combined with the knowledge in the knowledge base), we will define $\mathbf{goal}(\phi)$ to be true whenever ϕ follows from *one* of the agent's goals (and the agent's knowledge). The intuition here is that each goal in the goal base has an implicit *temporal dimension* and two different goals need not be achieved at the same time. Goals are thus used to represent *achievement goals*. Finally, $\mathbf{o\text{-}goal}(\phi)$ is true iff all goals of the agent are logically equivalent with ϕ; that is, the only goal present is the goal ϕ.

Definition 8.11. *(Semantics of Mental State Conditions)*
Let $m = \langle K, \Sigma, \Gamma \rangle$ be a mental state. The semantics of *closed* mental state conditions ψ is defined by the following semantic clauses:

$$\begin{array}{lll} m \models_\Psi \mathbf{bel}(\phi) & \text{iff} & \Sigma \models_K \phi, \\ m \models_\Psi \mathbf{goal}(\phi) & \text{iff} & \exists \gamma \in \Gamma : \gamma \models_K \phi, \\ m \models_\Psi \mathbf{o\text{-}goal}(\phi) & \text{iff} & m \models_\Psi \mathbf{goal}(\phi) \text{ and } \forall \phi' (m \models_\Psi \mathbf{goal}(\phi') \Rightarrow \models_c \phi \leftrightarrow \phi'), \\ m \models_\Psi \psi_1 \wedge \psi_2 & \text{iff} & m \models_\Psi \psi_1 \text{ and } m \models_\Psi \psi_2, \\ m \models_\Psi \neg\psi & \text{iff} & m \not\models_\Psi \psi. \end{array}$$

As before, for open formulas $\psi \in \mathcal{L}_\Psi$ we define $m \models_\Psi \psi$ iff there is a substitution such that $\psi\theta$ is closed and $m \models_\Psi \psi\theta$.

Note that in the definition of the semantics of mental state conditions we have been careful to distinguish between the consequence relation that is defined, denoted by $\models_\Psi$, and the consequence relation $\models$ defined in Definition 8.4. The definition thus shows how the meaning of a mental state condition can be derived from the semantics of the underlying knowledge representation language.

Proposition 8.1. *Let $m = \langle K, \Sigma, \Gamma \rangle$ be a mental state. Then we have:*

$$m \models_\Psi \neg \boldsymbol{bel}(\phi) \;\; \textit{iff} \;\; \Sigma \models_K \neg\phi$$

Proof. We have: $m \models_\Psi \neg \boldsymbol{bel}(\phi)$ *iff* $m \not\models_\Psi \boldsymbol{bel}(\phi)$ *iff* $\Sigma \not\models_K \phi$ *iff* $\langle \Sigma, [\![K]\!](\Sigma)\rangle \not\models \phi$ *iff* $\langle \Sigma, [\![K]\!](\Sigma)\rangle \models \neg\phi$ *iff* $\Sigma \models_K \neg\phi$. □

Proposition 8.1 is a direct consequence of the closed world assumption. That is, when ϕ is not believed to be the case, by the closed world assumption it then follows that $\neg\phi$. In other words, we have that $\neg\mathbf{bel}(\phi)$ is equivalent with $\mathbf{bel}(\neg\phi)$.

P1	if ψ is an instantiation of a classical tautology, then $\models_\Psi \psi$.
P2	if $\models_c \phi$, then $\models_\Psi \mathbf{bel}(\phi)$.
P3	$\models_\Psi \mathbf{bel}(\phi \rightarrow \phi') \rightarrow (\mathbf{bel}(\phi) \rightarrow \mathbf{bel}(\phi'))$.
P4	$\models_\Psi \neg\mathbf{bel}(\bot)$.
P5	$\models_\Psi \neg\mathbf{bel}(\phi) \leftrightarrow \mathbf{bel}(\neg\phi)$.
P6	$\models_\Psi \forall x(\mathbf{bel}(\phi)) \leftrightarrow \mathbf{bel}(\forall x(\phi))$.
P7	$\not\models_\Psi \mathbf{goal}(\top)$.
P8	$\models_\Psi \neg\mathbf{goal}(\bot)$.
P9	if $\models \phi \rightarrow \phi'$, then $\models_\Psi \mathbf{goal}(\phi) \rightarrow \mathbf{goal}(\phi')$.
P10	$\models_\Psi \mathbf{goal}(\forall x(\phi)) \rightarrow \forall x(\mathbf{goal}(\phi))$.

Table 8.1 Properties of Beliefs and Goals

We briefly discuss some of the properties listed in Table 8.1. The first property (P1) states that mental state conditions that instantiate classical tautologies such as $\mathbf{bel}(\phi) \vee \neg\mathbf{bel}(\phi)$ and $\mathbf{goal}(\phi) \rightarrow (\mathbf{bel}(\phi') \rightarrow \mathbf{goal}(\phi))$ are valid with respect to $\models_\Psi$. Property (P2) corresponds with the usual necessitation rule of modal logic and states that an agent believes all validities of the base logic. (P3) expresses that the belief modality distributes over implication. This implies that the beliefs of an agent are closed under logical consequence. Property (P4) states that the beliefs of an agent are consistent. In essence, the belief operator thus satisfies the properties of the system KD (see e.g. [314]). Although in its current presentation, it is not allowed to nest belief or goal operators in mental state conditions in GOAL, from [314], section 1.7, we conclude that we may assume *as if* our agent has positive $\mathbf{bel}(\phi) \rightarrow \mathbf{bel}(\mathbf{bel}(\phi))$ and negative $\neg\mathbf{bel}(\phi) \rightarrow \mathbf{bel}(\neg\mathbf{bel}(\phi))$ introspective properties: every formula in the system KD45 (which is KD together with the two mentioned properties) is equivalent to a formula without nestings of operators. Property (P7) shows that $\neg\mathbf{goal}(\top)$ can be used to express that an agent has *no* goals. Property (P8) states that an agent also does not have inconsistent goals, that is, we have $\models_c \neg\mathbf{goal}(\bot)$. Property (P9) states that the goal operator is closed under implication in the base language. That is, whenever $\phi \rightarrow \phi'$ is valid in the base language then we also have that $\mathbf{goal}(\phi)$ implies $\mathbf{goal}(\phi')$. This is a difference with the presentation in [60] which is due to the more basic goal modality we have introduced here. It is important to note here that we do *not* that $\mathbf{bel}(\phi) \wedge \mathbf{goal}(\phi)$ is inconsistent. Finally, property (P10) is valid, but the implication cannot be reversed: $\forall x(\mathbf{goal}(\phi)) \rightarrow \mathbf{goal}(\forall x(\phi))$ is not valid.

It is clear from the properties discussed that the **goal** operator does not correspond with the more common sense notion of a goal but instead is an operator mainly introduced for technical reasons. The reason for using the label **goal** is to clearly differentiate it from the **bel** operator and make clear that this operator is related to the motivational attitudes of an agent. The **goal** operator is a *primitve* operator that does not match completely with the common sense notion of a goal that needs to be achieved in the future. The **goal** operator, however, may be used to define so-called *achievement goals* that usually require effort from an agent in order to realize the goal. The main characteristic which sets an achievement goal apart from "primitives" goals thus is that they are not believed to be achieved already. As noticed, it is possible to define the concept of an achievement goal and to introduce an achievement goal **a-goal** operator using the primitive **goal** operator and the belief **bel** operator. It is also useful to be able to express that a goal has been (partially) achieved. We therefore also introduce a "goal achieved" **goal-a** operator to be able to state that (part of) a goal is believed to be achieved. This operator can also be defined using the **goal** and **bel** operator.

Definition 8.12. *(Achievement Goal and Goal Achieved Operators)*
The *achievement goal* **a-goal**(ϕ) operator and the *goal achieved* **goal-a**(ϕ) operator are defined by:

$$\textbf{a-goal}(\phi) \stackrel{df}{=} \textbf{goal}(\phi) \wedge \neg \textbf{bel}(\phi),$$
$$\textbf{goal-a}(\phi) \stackrel{df}{=} \textbf{goal}(\phi) \wedge \textbf{bel}(\phi).$$

Both of these operators are useful when writing agent programs. The first is useful to derive whether a part of a goal has not yet been (believed to be) achieved whereas the second is useful to derive whether a part of a goal has already been (believed to be) achieved. It should be noted that an agent can only believe that part of one of its goals has been achieved but cannot believe that one of its goals has been *completely* achieved as such goals are removed automatically from the goal base. That is, whenever we have $\gamma \in \Gamma$ we must have **a-goal**(γ), or, equivalently, **goal**(γ) $\wedge$ $\neg$**bel**(γ) since it is not allowed by the third rationality constraint in Definition 8.9 that an agent believes γ in that case (see also (P21) and (P22) in Table 8.2).

Table 8.2 lists some properties of the **a-goal**, **goal-a**, and **o-goal** operators.[6] The main difference between the **a-goal** and **goal-a** operators concern Properties (P12) and (P17) and Properties (P15) and (P20), respectively. Property (P12) expresses that an achievement goal $\phi \wedge (\phi \rightarrow \phi')$ does not imply an achievement goal ϕ'. This property avoids the *side effect problem*. The **goal-a** operator, however, is closed under such effects as any side effect of a goal that has been achieved also is realized by implication. Properties (P15) and (P20) highlight the key difference between achievement goals and goals achieved: achievement goals are not believed to be achieved, whereas goals achieved are believed to be achieved.

[6] The properties of the **a-goal** operator are the same as those for the G operator listed in Lemma 2.4 in [60].

P11	$\not\models_\psi$ **a-goal**$(\phi \rightarrow \phi') \rightarrow$ (**a-goal**$(\phi) \rightarrow$ **a-goal**$(\phi'))$.
P12	$\not\models_\psi$ **a-goal**$(\phi \wedge (\phi \rightarrow \phi')) \rightarrow$ **a-goal**(ϕ').
P13	$\not\models_\psi$ (**a-goal**$(\phi) \wedge$ **a-goal**$(\phi')) \rightarrow$ **a-goal**$(\phi \wedge \phi')$.
P14	if $\models (\phi \leftrightarrow \phi')$, then $\models_\psi$ **a-goal**$(\phi) \leftrightarrow$ **a-goal**(ϕ').
P15	$\models_\psi$ **a-goal**$(\phi) \rightarrow \neg$**bel**(ϕ).
P16	$\not\models_c$ **goal-a**$(\phi \rightarrow \phi') \rightarrow$ (**goal-a**$(\phi) \rightarrow$ **goal-a**$(\phi'))$.
P17	$\models_\psi$ **goal-a**$(\phi \wedge (\phi \rightarrow \phi')) \rightarrow$ **goal-a**(ϕ').
P18	$\not\models_\psi$ (**goal-a**$(\phi) \wedge$ **goal-a**$(\phi')) \rightarrow$ **goal-a**$(\phi \wedge \phi')$.
P19	if $\models (\phi \leftrightarrow \phi')$, then $\models_\psi$ **goal-a**$(\phi) \leftrightarrow$ **goal-a**(ϕ').
P20	$\models_\psi$ **goal-a**$(\phi) \rightarrow$ **bel**(ϕ).
P21	$\models_\psi$ **o-goal**$(\phi) \rightarrow$ **goal**(ϕ).
P22	$\models_\psi$ **o-goal**$(\phi) \rightarrow \neg$**bel**(ϕ).
P23	$\not\models_\psi$ **o-goal**$(\phi \rightarrow \phi') \rightarrow$ (**o-goal**$(\phi) \rightarrow$ **o-goal**$(\phi'))$.

Table 8.2 Properties of Achievement Goals, Goals Achieved, and Only Goals

8.3.4 Actions and Action Selection

A GOAL agent derives its choice of action from its goals and beliefs (in combination with its knowledge). Action selection is implemented by means of so-called *action rules* that inspect the mental state of the agent. Actions are performed to change the environment. An agent keeps track of such changes by updating its beliefs. The updates asscociated with the execution of an action are provided by so-called *action specifications*. Actions may have *conditional effects* [336].[7] For example, the result of performing the action of switching the light button depends on the current state and may result in the light being either on or off. The effect of the light switching action thus depends on the state in which the action is executed.

We present a formal, operational semantics for GOAL with actions with conditional effects and use Plotkin-style transition semantics [340] to do so. This semantics is a computational semantics that provides a specification for executing a GOAL agent on a machine.

We distinguish between two types of actions: user-specified actions and built-in actions. Built-in actions are part of the GOAL language and include an action for *adding and deleting beliefs* and for *adopting* and *dropping goals*. The **insert**(ϕ) action, where ϕ should be a conjunction of basic literals, adds facts that occur positively in ϕ to the belief base and removes facts that occur negatively in ϕ from the belief base. This action can always be performed and has a precondition $\top$. The **adopt**(ϕ) action, where ϕ should be a conjunction of basic facts, adds a goal to the goal base. The **adopt**(ϕ) action can only be performed if ϕ is not believed to be the case; that is, the precondition of **adopt**(ϕ) is $\neg$**bel**(ϕ).[8] Finally, the **drop**(ϕ) ac-

[7] This is an extension of GOAL as presented in [60] introduced in [228].

[8] The condition that ϕ or a logically equivalent formula is not already present in the goal base may be added but is less important in this context but is relevant for efficiency reasons to avoid having to evaluate multiple times whether one and the same goal has been achieved.

tion, where ϕ should again be a conjunction of basic facts, removes any goal in the goal base that implies ϕ. The precondition of **drop**(ϕ) is $\top$ and thus can always be performed.

The semantics of these actions is formally defined by means of a *mental state transformer function* $\mathcal{M}$. This function maps an action $a(\mathbf{t})$ and a mental state m to a new mental state m' that is the result of performing the action. It is useful to introduce some notation here. We use $pos(\phi)$ and $neg(\phi)$ to denote the set facts that occur *positively* respectively *negatively* in a conjunction of literals ϕ.

Definition 8.13. *(Semantics of Built-in Actions)*
Let $m = \langle K, \Sigma, \Gamma \rangle$ be a mental state. The *mental state transformer function* $\mathcal{M}$ is defined as follows for the built-in actions **insert**(ϕ), **adopt**(ϕ), and **drop**(ϕ), where ϕ needs to be of the appropriate form: [9]

$$\mathcal{M}(\mathbf{insert}(\phi), m) = \langle K, (\Sigma \setminus neg(\phi)) \cup pos(\phi), \Gamma \rangle.$$

$$\mathcal{M}(\mathbf{adopt}(\phi), m) = \begin{cases} \langle K, \Sigma, \Gamma \cup \{\phi\} \rangle & \text{if } m \models_{\Psi} \neg \mathbf{bel}(\phi), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\mathcal{M}(\mathbf{drop}(\phi), m) = \langle K, \Sigma, \Gamma \setminus \{\phi' \in \Gamma \mid \phi' \models_c \phi\} \rangle.$$

To enable a programmer to add user-specified actions to an agent program we need a language for specifying when an action can be performed and what the effects of performing an action are. Actions are written as $a(\mathbf{t})$ where a is the *name* of the action and $\mathbf{t}$ are the *parameters* of the action. Preconditions specify when an action can be performed. These conditions can be specified using the knowledge representation language $\mathcal{L}_0$. The effects of an action may be conditional on the state in which the action is performed. To express such conditional effects we use statements of the form $\phi \Rightarrow \phi'$ where ϕ is called the *condition* and ϕ' the *effect*. Intuitively, an action with *conditional effect* $\phi \Rightarrow \phi'$ means that if the action is performed in a state where ϕ is true, then the effect of the action is ϕ'. When the condition ϕ is $\top$ we also simply write ϕ' to denote the effect. Free variables in a conditional effect may be bound universally and we write $\forall \mathbf{x}(\phi \Rightarrow \phi')$. Finally, multiple conditional effects may be associated with an action and in that case we write $\forall \mathbf{x}_1(\phi_1 \Rightarrow \phi'_1) \wedge \ldots \wedge \forall \mathbf{x}_n(\phi_n \Rightarrow \phi'_n)$. Finally, for specifying the preconditions and effects of an action $a(\mathbf{t})$ we use a Hoare-triple-style notation. That is, we use triples consisting of a precondition, an action and a conjunction of conditional effects to specify the precondition and effects of a particular action. Note, however, that although conditional effects are part of the postcondition of an action, the conditions of such effects need to be evaluated in the state where the action is executed and in this respect is similar to a precondition. The following definition summarizes the previous discussion.

Definition 8.14. *(Conditional Effect, Action Specification)*

[9] Note that since goals are conjunctions of basic facts, the formal semantics given here for the **drop** action that uses $\models_c$ is easily replaced by an efficient computational mechanism.

- A *conditional effect statement* is an expression of the form $\forall \mathbf{x}(\phi \Rightarrow \phi')$, where $\phi \in \mathcal{L}_0$ and ϕ' is a conjunction of basic literals. The quantor $\forall \mathbf{x}$ may be absent and whenever ϕ is $\top$ we also write $\forall \mathbf{x}(\phi')$.
- An *action specification* is a triple written as:

$$\{\ \phi\ \}\ a(\mathbf{t})\ \{\ \forall \mathbf{x}_1(\phi_1 \Rightarrow \phi'_1) \wedge \ldots \wedge \forall \mathbf{x}_n(\phi_n \Rightarrow \phi'_n)\ \}$$

  where ϕ is a formula from $\mathcal{L}_0$ called the *precondition*, $a(\mathbf{t})$ is an action with *name* a and *parameters* $\mathbf{t}$, and $\forall \mathbf{x}_1(\phi_1 \Rightarrow \phi'_1) \wedge \ldots \wedge \forall \mathbf{x}_n(\phi_n \Rightarrow \phi'_n)$ is a conjunction of conditional effect statements, which is also called *postcondition*. All free variables in the postcondition must occur free in either the precondition or the action parameters. Finally, the postcondition is required to be *consistent* (see Definition 8.15 below).

Note the condition on free variables in the definition of an action specification. The free variables that occur in the postcondition need to occur free in the precondition or action parameters in order to ensure they are instantiated. An action can only be performed when all free variables in the action parameters and the postcondition have been instantiated.

In GOAL, a precondition is evaluated on the belief base of the agent. This means that an agent believes it can perform an action if the agent believes the associated precondition of that action. An action may affect both the beliefs and goals of an agent. An agent's knowledge base is static and does not change since it is used to represent conceptual and domain knowledge that does not change. The postcondition of an action specifies how the belief base of an agent should be updated. Intuitively, for each conditional effect $\phi \Rightarrow \phi'$ if ϕ is believed then the facts that occur positively in ϕ' are added to the belief base and the facts that occur negatively in ϕ' are removed from the belief base.

Definition 8.15. *(Positive and Negative Effects of an Action, Consistency)*
Let $\{\ \phi\ \}\ a(\mathbf{t})\ \{\ \phi'\ \}$ be an instantiation of an action specification such that ϕ and $a(\mathbf{t})$ are ground, which implies that $\phi' = \forall \mathbf{x}_1(\phi_1 \Rightarrow \phi'_1) \wedge \ldots \wedge \forall \mathbf{x}_n(\phi_n \Rightarrow \phi'_n)$ is also ground. Then the *positive effects* respectively the *negative effects* in a mental state m are defined by:

$$Eff^{+}(\phi', m) = \bigcup_i \{\ at \in pos(\phi'_i) \mid m \models_{\Psi} \mathbf{bel}(\phi_i)\}$$
$$Eff^{-}(\phi', m) = \bigcup_i \{\ at \in neg(\phi'_i) \mid m \models_{\Psi} \mathbf{bel}(\phi_i)\}$$

A postcondition ϕ' is said to be *consistent* if for all mental states m and all instantiations of an action specification the set $Eff^{+}(\phi', m) \cup \neg Eff^{-}(\phi', m)$ is consistent, i.e. if this set does not contain complementary literals.[10]

[10] Where $\neg T$ denotes the set $\{\neg\phi \mid \phi \in T\}$, with $T \subseteq \mathcal{L}_0$.

After updating the beliefs, an agent also needs to check whether any of its goals have been realized and can be removed from its goal base. A GOAL agent only removes goals that have been completely achieved. A goal such as $on(a,b) \wedge on(b,table)$ is not removed or replaced by $on(b,table)$ since the goal has been achieved only when at the same time block a is on block b and b is on the table. The semantics of user-specified actions is again defined by means of the mental state transformer function $\mathcal{M}$.

Definition 8.16. *(Semantics of User-Specified Actions)*
Let $m = \langle K, \Sigma, \Gamma \rangle$ be a mental state, and $\{ \phi \}\, a(\mathbf{t})\, \{ \phi' \}$ be an instantiation of an action specification such that ϕ and $a(\mathbf{t})$ are ground. Then the *mental state transformer function* $\mathcal{M}$ is defined as follows for action $a(\mathbf{t})$:

$$\mathcal{M}(a(\mathbf{t}), m) = \begin{cases} \langle K, \Sigma', \Gamma' \rangle & \text{if } m \models_\Psi \mathbf{bel}(\phi) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where:

- $\Sigma' = (\Sigma \setminus \mathit{Eff}^-(\phi')) \cup \mathit{Eff}^+(\phi')$.
- $\Gamma' = \Gamma \setminus \{\phi \in \Gamma \mid \Sigma \models_K \phi\}$.

Note that the formula ϕ' in Definition 8.16 is closed since any free variables in a postcondition ϕ' need to be free in either the precondition ϕ or action parameters $a(\mathbf{t})$.

GOAL agents derive their choice of action from their beliefs and goals. They do so by means of *action rules* of the form **if** ψ **then** α. Here ψ is a mental state condition and α an action, either built-in or user-specified. An action rule specifies that action α may be selected for execution if the mental state condition ψ and the precondition of action α hold. In that case, we say that action α is an *option*. At runtime, a GOAL agent non-deterministically selects an action from the set of options to perform. This is expressed in the following transition rule, describing how an agent's mental state changes from one to another.[11]

Definition 8.17. *(Action Semantics)*
Let m be a mental state, and **if** ψ **then** α be an action rule, and θ a substitution. The transition relation $\xrightarrow{\alpha\theta}$ is the smallest relation induced by the following transition rule.

$$\frac{m \models_\Psi \psi\theta \quad \mathcal{M}(\alpha\theta, m) \text{ is defined}}{m \xrightarrow{\alpha\theta} \mathcal{M}(\alpha\theta, m)}$$

A GOAL agent (program) consists of the knowledge, initial beliefs and goals, action rules and action specifications.

[11] A transition rule is an inference rule for deriving transitions or computation steps. The statements above the line are called the premises of the rule and the transition below the line is called the conclusion. See also [340].

Definition 8.18. *(Goal Agent Program)*
A Goal *agent program* is a tuple $\langle K, \Sigma, \Gamma, \Pi, A \rangle$ with:

- $\langle K, \Sigma, \Gamma \rangle$ a mental state,
- Π a set of action rules, and
- A a set of action specifications.

Below we will assume that there is exactly one action specification associated with each action name a. Although it is possible to use multiple action specifications in a Goal agent, this assumption is introduced to somewhat simplify the presentation below.

The execution of a Goal agent results in a *run* or computation. We define a computation as a sequence of mental states and actions, such that each mental state can be obtained from the previous by applying the transition rule of Definition 8.17. As Goal agents are non-deterministic, the semantics of a Goal agent is defined as the *set* of possible computations of the Goal agent, where all computations start in the initial mental state of the agent.

Definition 8.19. *(Run)*
A *run* or *computation* of an agent $Agt = \langle K, \Sigma, \Gamma, \Pi, A \rangle$, typically denoted by r, is an infinite sequence of mental states and actions $m_0, \alpha_0, m_1, \alpha_1, m_2, \alpha_2, \ldots$ such that:

- $m_0 = \langle K, \Sigma, \Gamma \rangle$, and
- for each i we have either that:
  - $m_i \xrightarrow{\alpha_i} m_{i+1}$ can be derived using the transition rule of Definition 8.17, or
  - for all $j > i$, $m_j = m_i$ and $m_i \not\xrightarrow{\alpha} m'$ for any α and m', and $\alpha_i = \textbf{skip}$.[12]

We also write r_i^m to denote the ith mental state and r_i^a to denote the ith action. The meaning $\mathcal{R}_{Agt}$ of a Goal agent Agt is the set of all possible runs of that agent.

Observe that a computation is infinite by definition, even if the agent is not able to perform any actions anymore from some point in time on. In the latter case, the agent is assumed to perform no action at all, which is represented by **skip**. Also note that the concept of a computation is a general notion in program semantics that is not particular to Goal. The notion of a computation can be defined for any (agent) programming language that is provided with a well-defined operational semantics.

[12] Implicitly, this also defines the semantics of **skip**. The label **skip** denotes the action that does not change the mental state of an agent.

8.4 Verifying Goal Agent Programs

The verification logic for Goal that we present here is an extension of linear temporal logic (LTL) with an action, belief and goal operator. It is similar to the verification framework presented in [60]. However, the verification framework presented here does not consist of two different components, a Hoare logic component and a linear temporal logic component, as in [60]. Instead, the Hoare logic for reasoning about actions in [60] is replaced by an action theory represented in the temporal verification logic.

A second difference with [60] is that the verification framework presented here is less abstract. In particular, the verification logic presented here incorporates Reiter's solution to the frame problem [364].

The differences with the very generic approach presented in [60] have important implications. The approach presented here introduces a deductive approach to the verification of agents that is obtained by a direct translation of an agent program into the logic. We will discuss in more detail what this means in section 8.4.2.

8.4.1 Verification Logic

To obtain a verification logic for Goal agents temporal operators are added on top of mental state conditions to be able to express temporal properties over runs and an action operator **done**(α) is introduced.

Definition 8.20. *(Temporal Language: Syntax)*
The *temporal language* $\mathcal{L}_G$, with typical elements χ, χ', is defined by:

$$\chi \in \mathcal{L}_G ::= \psi \in \mathcal{L}_\Psi \mid \mathbf{done}(\alpha) \mid \neg\chi \mid \chi \wedge \chi \mid \forall x(\chi) \mid \bigcirc\chi \mid \chi \,\mathbf{until}\, \chi$$

Using the **until** operator, other temporal operators such as the "sometime in the future operator" $\Diamond$ and the "always in the future operator" $\Box$ can be introduced as abbreviations for $\Diamond\psi ::= \top \,\mathbf{until}\, \psi$ and $\Box\psi ::= \neg\Diamond\neg\psi$.

Although the language $\mathcal{L}_G$ is intended for verification of runs of Goal agents, the semantics of $\mathcal{L}_G$ is defined more generally relative to a trace t. Each *time point* (t,i) denotes a state of the agent and is labeled with the action performed at that state by the agent. Instead of databases to model the mental state of an agent, moreover, we use sets of Herbrand interpretations as models of the agents' beliefs and goals. A *Herbrand interpretation* is a set of ground atoms [22].

Definition 8.21. *(Trace)*
A *trace* is a mapping from the natural numbers $\mathbb{N}$ (including 0) to triples $\langle B, G, \alpha\rangle$, where B consists of a set of Herbrand interpretations, G is a set of sets of Herbrand

models, and α is an action (including possibly **skip**). A pair (t,i) is called a *time point*. We use t_i^b to denote B, t_i^g to denote G, and t_i^a to denote α in $t(i) = \langle B, G, \alpha \rangle$. We use $\bigcup t_i^g$ to denote the union of all sets in t_i^g (which yields a set of Herbrand interpretations).

The trace semantics introduced above is an approximation of a more general modal semantics, and only includes those elements strictly needed in our setting. The B and G components of a time point correspond respectively to the belief and goal base of an agent, modeled as (sets of) sets of Herbrand models. This setup allows us to use a classical first-order semantics (where models have admittedly been restricted to Herbrand models). Also note that the set of traces does not need to correspond with the set of runs of a particular agent, but - as we will show - we do have that any set of runs generated by an agent may be viewed as a subset of the set of all traces.

Definition 8.22. *(Temporal Language: Semantics)*
The truth conditions of sentences from $\mathcal{L}_G$ are provided relative to a time point (t,i) and are inductively defined by:

$$
\begin{array}{ll}
t,i \models_G \textbf{bel}(\phi) & \text{iff } \forall M \in t_i^b : M \models_c \phi, \\
t,i \models_G \textbf{goal}(\phi) & \text{iff } \exists g \in t_i^g : \forall M \in g : M \models_c \phi, \\
t,i \models_G \textbf{o-goal}(\phi) & \text{iff } \forall M : M \in \bigcup t_i^g \Leftrightarrow M \models_c \phi, \\
t,i \models_G \textbf{done}(\alpha) & \text{iff } t_i^a = \alpha, \\
t,i \models_G \neg\chi & \text{iff } t,i \not\models_G \chi, \\
t,i \models_G \chi \wedge \chi' & \text{iff } t,i \models_G \chi \text{ and } t,i \models_G \chi', \\
t,i \models_G \forall x(\chi) & \text{iff } t,i \models_G \chi[t/x] \text{ for all } t \in \mathcal{T}, \\
t,i \models_G \bigcirc\chi & \text{iff } t,i+1 \models_G \chi, \\
t,i \models_G \chi \textbf{ until } \chi' & \text{iff } \exists j \geq i : t,j \models_G \chi' \text{ and } \forall i \leq k < j : t,k \models_G \chi
\end{array}
$$

We write $t \models \chi$ for $t,0 \models \chi$.

8.4.2 Logical Characterization of Agent Programs

An issue in the verification of agents is whether the behavior of an agent program can be *fully* characterized in a verification logic. A verification logic should not only enable proving properties of *some* of the possible runs of an agent but should also enable to conclude that certain properties hold on *all* possible runs of the agent. This is important because a verification logic that does not allow to fully characterize the behavior of an agent cannot be used to *guarantee* that certain properties are never violated, for example. We explore this issue in more detail in this section.

In order to show that a GOAL agent program can be fully characterized logically we transform a program into a set of corresponding axioms. To prove formally that this set of axioms fully characterizes the GOAL agent we need to show that the traces

that are models of these axioms correspond with the runs of the GOAL agent. A basic GOAL agent program $\langle K, \Sigma, \Gamma, \Pi, A \rangle$ as discussed above consists of the knowledge contained in K, the initial beliefs Σ and goals Γ, a set of action rules Π, and action specifications A. Each of these components will be transformed in a set of corresponding axioms of the language $\mathcal{L}_G$. It turns out that providing axioms that fully characterize the agent is possible by using the **o-goal** operator introduced in section 8.3.3 and imposing a restriction on the goal base of an agent.

Providing the appropriate axioms that fully characterize the knowledge and beliefs in our setting is relatively straightforward. The knowledge base represents what the agent knows about derived predicates and captures the agent's conceptual and domain knowledge. Since knowledge does not change, intuitively, we must have that $\Box\mathbf{bel}(K)$ is true on all runs of the agent program. This does not yet fully characterize the agent's knowledge with respect to derived predicates, however, as it does not exclude that the agent believes more than $\mathbf{bel}(K)$. A full characterization of the agent's knowledge can be provided using the completion $comp(K)$ as defined in Definition 8.7. The knowledge of the agent about derived predicates is characterized by the following axiom.

$$\Box\mathbf{bel}(comp(K)) \tag{8.1}$$

Similarly, we have that $\mathbf{bel}(\Sigma)$ is true in the initial (mental) state as the belief base Σ contains the initial beliefs of the agent. Again this is not sufficient to fully characterize the agent's beliefs about basic predicates and to exclude that the agent believes basic facts that are not included in the initial state. We have assumed that the base language $\mathcal{L}_0$ contains a finite number of basic predicates and the belief base is finite as well. In addition, a closed world assumption was made. It is therefore possible to finitely characterize an agent's beliefs about basic facts by axioms of the following form, where we need one axiom for each basic predicate $b \in \mathcal{B}$.

$$\forall \mathbf{x}(\mathbf{bel}(b(\mathbf{x})) \leftrightarrow (\mathbf{x} = \mathbf{t}_1 \vee \ldots \vee \mathbf{x} = \mathbf{t}_n)) \tag{8.2}$$

The particular form that the expression $(\mathbf{x} = \mathbf{t}_1 \vee \ldots \vee \mathbf{x} = \mathbf{t}_n)$ can be determined by inspecting all occurrences of the b predicate in the initial belief base of the agent program. The number of disjuncts needed corresponds with the number of ground basic facts of the form $b(\mathbf{t})$ in the initial belief base. If the predicate b does not occur in the belief base, then instead of the axiom above the axiom $\forall \mathbf{x}(\neg\mathbf{bel}(b(\mathbf{x}))$ should be used.

A set of axioms to *fully* characterize the initial goal base of the agent cannot be constructed similarly to those for the initial beliefs. Although it is clearly true that $\mathbf{goal}(\phi_1) \wedge \ldots \wedge \mathbf{goal}(\phi_n)$ holds for any agent with an initial goal base that consists of the goals $\phi_1, \ldots, \phi_n$, it is not so easy to provide an axiom that excludes the possibility that the agent may have any other goals. Although the axiom above may be sufficient for proving that the behavior of the agent will satisfy some specific properties, it is not sufficient to exclude behavior that leads to the violation of certain desired properties. In particular, it is not sufficient to prove that an agent does *not* have

certain goals. Conditions of the form $\neg\mathbf{goal}(\phi)$, which are often used in practice in Goal programs, cannot be derived from a specification $\mathbf{goal}(\phi_1) \wedge \ldots \wedge \mathbf{goal}(\phi_n)$ of goals that are pursued by the agent. We will postpone the discussion of this issue and propose a solution below.

Action rules of the form **if** ψ **then** $a(\mathbf{t})$ provide an agent with the capability to select actions. An action $a(\mathbf{t})$ can only be performed when its precondition $pre(a(\mathbf{t}))$ holds and when the mental state condition ψ of one of the action rules for $a(\mathbf{x})$ holds. We introduce the notion of *enabledness* to express that an action can be performed.

Definition 8.23. *(Enabled)*
Suppose that **if** ψ_1 **then** $a(\mathbf{t}_1)$, ..., **if** ψ_n **then** $a(\mathbf{t}_n)$ are all action rules for a user-specified action $a(\mathbf{x})$ in program $\langle K, \Sigma, \Gamma, \Pi, A\rangle$ and the variables in $\mathbf{x}$ do not occur in any of the conditions ψ_i. The definition of action $a(\mathbf{x})$ being *enabled*, written $enabled(a(\mathbf{x}))$, is the following:

$$enabled(a(\mathbf{x})) \stackrel{df}{=} pre(a(\mathbf{x})) \wedge ((\mathbf{x} = \mathbf{t}_1 \wedge \psi_1) \vee \ldots \vee (\mathbf{x} = \mathbf{t}_n \wedge \psi_n))$$

For the built-in actions **insert**, **adopt** and **drop**, the following axioms are provided, where we suppose again that **if** ψ_1 **then** α, ..., **if** ψ_n **then** α are all action rules for the built-in action α:

$$\begin{aligned} enabled(\mathbf{insert}(\phi)) &\stackrel{df}{=} \psi_1 \vee \ldots \vee \psi_n \\ enabled(\mathbf{adopt}(\phi)) &\stackrel{df}{=} \neg\mathbf{bel}(\phi) \wedge (\psi_1 \vee \ldots \vee \psi_n) \\ enabled(\mathbf{drop}(\phi)) &\stackrel{df}{=} \psi_1 \vee \ldots \vee \psi_n \end{aligned}$$

These axioms express that the action **insert** and **drop** can always be performed when the mental state conditions of the action rules in which they occur hold. For the action **adopt**, additionally, the formula ϕ to be adopted as a goal may not be believed to be the case.

We can use the notion of enabledness to introduce an axiom that characterizes action selection in Goal. The following axiom represents that $a(\mathbf{x})$ can only be performed if one of the mental state conditions ψ_i holds (where variables have been appropriately substituted to obtain a ground action):

$$\forall\mathbf{x}\Box(\bigcirc\mathbf{done}(a(\mathbf{x})) \rightarrow enabled(a(\mathbf{x}))) \tag{8.3}$$

This axiom is sufficient to express that an action is only performed when the right conditions are true. The semantics of actions in Definition 8.17 guarantees that a single action may be executed at any time (it is an interleaving model of action execution). This property is also built into the trace semantics for linear temporal logic above and we do not need an axiom to ensure this. In other words, the following axiom, which excludes that any two actions happen simultaneously, is valid on traces (by Definition 8.21).

$$\forall \mathbf{x} \Box((\mathbf{done}(a(\mathbf{t})) \wedge \mathbf{done}(a'(\mathbf{t}'))) \rightarrow a(\mathbf{t}) = a'(\mathbf{t}'))$$

We need to specify that **skip** is performed in case no other action can be performed. Using the *enabled* predicate introduced above, we obtain the following axiom:

$$\forall \mathbf{x}_1, \ldots, \mathbf{x}_n \Box(\bigcirc \mathbf{done}(\mathbf{skip}) \leftrightarrow (\neg \mathit{enabled}(a_1(\mathbf{x}_1)) \wedge \ldots \wedge \neg \mathit{enabled}(a_n(\mathbf{x}_n)))) \quad (8.4)$$

Finally, we need to state that an agent always performs some action, or otherwise performs the **skip** action.

$$\forall \mathbf{x}_1, \ldots, \mathbf{x}_n \Box \bigcirc (\mathbf{done}(a_1(\mathbf{x}_1)) \vee \ldots \vee \mathbf{done}(a_n(\mathbf{x}_n)) \vee \mathbf{done}(\mathbf{skip})) \quad (8.5)$$

The last component of a Goal agent program that we need to translate consists of action specifications of the form $\{\phi\}\ a(\mathbf{t})\ \{\phi_1 \Rightarrow \phi'_1 \wedge \ldots \wedge \phi_n \Rightarrow \phi'_n\}$. Note that the action preconditions have already been captured in the axioms above that translate the action rules of a program. What remains is to represent the *action effects*. The effects on the beliefs of an agent are represented here by a temporal logic encoding of Reiter's successor state axioms [305, 364]. The basic idea of Reiter's solution to the frame problem is that a propositional atom $p(\mathbf{t})$ may change its truth value only if an action is performed that affects this truth value. Two cases are distinguished: (i) actions that have $p(\mathbf{t})$ as effect and (ii) actions that have $\neg p(\mathbf{t})$ as effect. For each atom $p(\mathbf{x})$ the first set of actions $a_1(\mathbf{t}_1), \ldots, a_m(\mathbf{t}_m)$ is collected and a disjunction is formed of the form $(\bigcirc \mathbf{done}(a_1(\mathbf{t}_1)1) \wedge \phi_1) \vee \ldots \vee (\bigcirc \mathbf{done}(a_m(\mathbf{t}_m)) \wedge \phi_m)$ denoted by A_p^+, where the ϕ_i are the corresponding conditions of the conditional effects that need to hold to establish $p(\mathbf{t})$, and, similarly, the second set of actions $a_{m+1}(\mathbf{t}_{m+1}), \ldots, a_n(\mathbf{t}_n)$ is collected and a disjunction is formed of the form $(\bigcirc \mathbf{done}(a_{m+1}(\mathbf{t}_{m+1})) \wedge \phi_{m+1}) \vee \ldots \vee (\bigcirc \mathbf{done}(a_n(\mathbf{t}_n)) \wedge \phi_n)$ denoted by A_p^-, where ϕ_i denote the conditions associated with the effects.[13]

Then, for each proposition $p(\mathbf{x})$ a successor state axiom of the form

$$\forall \mathbf{x} \Box(\bigcirc \mathbf{bel}(p(\mathbf{x})) \leftrightarrow (A_p^+ \vee (\mathbf{bel}(p(\mathbf{x})) \wedge \neg A_p^-))) \quad (8.6)$$

is introduced. Intuitively, such formulas express that at any time, in the next state $\mathbf{bel}(p(\mathbf{x}))$ holds iff an action is performed that has $p(\mathbf{x})$ as effect (i.e. A_p^+ holds) or $p(\mathbf{x})$ is true in the current state and no action that has $\neg p(\mathbf{x})$ as effect is performed (i.e. $\neg A_p^-$ holds). Note that this axiom is consistent if the postconditions are consistent in the sense of Definition 8.15.

The more difficult part is again to represent the effects of an action on an agent's goals. Informally, whenever an agent comes to believe that one of its goals has been *completely* achieved, it will drop this goal. The difficult part here is to represent the fact that a goal has been *completely* achieved and not just part of it. For this reason an axiom such as $(\mathbf{goal}(\phi) \wedge \bigcirc \mathbf{bel}(\phi)) \rightarrow \bigcirc \neg \mathbf{goal}(\phi)$ is not valid; it may be that

[13] Note that we assume that $\bigcirc$ binds stronger than $\wedge$ and $\bigcirc(\phi) \wedge \psi$ is equivalent to $(\bigcirc(\phi)) \wedge \psi$.

$\phi = \phi_1$ is part of a larger goal $\phi_1 \wedge \phi_2$ and ϕ_2 has not been achieved yet. In that case it would be irrational to drop the goal. Note that even when ϕ in the axiom would be restricted to conjunctions of basic facts the axiom would still not be valid. The axiom $(\mathbf{goal}(\phi) \wedge \bigcirc\neg\mathbf{bel}(\phi)) \rightarrow \bigcirc\mathbf{goal}(\phi)$ is valid, but does not capture the removal of a goal when it has been achieved.[14]

We have encountered two problems with the characterization of goals. The first concerns the characterization of the *initial goals* of the agent. The second concerns the *dynamics* of goals. One solution to resolve these issues is to use the **o-goal** operator to characterize goals. The **o-goal** operator can be used to fully characterize the goal base of an agent. This comes at a cost: the class of agents needs to be restricted to those agents that only have a single goal. As less restrictive solutions are not apparent in this setting, however, we use the **o-goal** operator.[15] From now on, it is therefore assumed that agents have a single goal to start with and that the adopt action has an additional precondition that ensures that an agent never adopts a second goal if it already has one.

Using the **o-goal** operator and the assumption that the initial goal base consists of a single goal ϕ, the initial goal base of an agent can be characterized by the following axiom.

$$\mathbf{o\text{-}goal}(\phi) \tag{8.7}$$

Using the **o-goal** operator it is possible to represent the dynamics of goals and to provide an axiom that captures the removal of goals correctly. Here we use the fact that $\neg\mathbf{goal}(\top)$ can only be true if the goal base is empty. Several axioms to characterize goal dynamics are introduced which each deal with one out of a number of cases.

$$\forall \mathbf{x}\Box(\mathbf{o\text{-}goal}(\phi) \wedge \bigcirc\mathbf{bel}(\phi)) \rightarrow \bigcirc\neg\mathbf{goal}(\top) \tag{8.8}$$

This axiom covers the cases where either a user-specified action or the built-in action **insert** is performed. Notice that ϕ cannot have been believed by the agent in the state in which the action was performed as we must have $\mathbf{o\text{-}goal}(\phi)$ for this axiom to apply. Since the beliefs of an agent are not changed by an **adopt** or **drop** action, we therefore can be certain neither of these actions has been performed.

Goals persist when an agent does not believe the goal to be achieved, *and* the goal has not been explicitly dropped by a **drop** action. We thus have:

$$\forall \mathbf{x}\Box(\mathbf{o\text{-}goal}(\phi) \wedge \bigcirc(\neg\mathbf{bel}(\phi) \wedge \mathit{done}(\alpha)) \rightarrow \bigcirc\mathbf{o\text{-}goal}(\phi) \tag{8.9}$$

[14] It is easy to show that (i) $(\mathbf{goal}(\phi) \wedge \bigcirc\mathbf{bel}(\phi)) \rightarrow \bigcirc\neg\mathbf{goal}(\phi)$ is inconsistent with (ii) $(\mathbf{goal}(\phi) \wedge \bigcirc\neg\mathbf{bel}(\phi)) \rightarrow \bigcirc\mathbf{goal}(\phi)$. Suppose $\mathbf{goal}(p \wedge q)$, $\bigcirc\mathbf{bel}(p)$ and $\bigcirc\neg\mathbf{bel}(q)$ are true at a time point (t,i). It follows that we have $\bigcirc\neg\mathbf{bel}(p \wedge q)$, and, as a consequence of (ii), we then must have that $\bigcirc\mathbf{goal}(p \wedge q)$ is true. Note that this implies that both $\bigcirc\mathbf{goal}(p)$ and $\bigcirc\mathbf{goal}(q)$ are true. Using (i) and $\bigcirc\mathbf{bel}(p)$ we then should also have $\bigcirc\neg\mathbf{goal}(p)$, which yields a contradiction.

[15] One way to go is to allow temporal formulas in the goal base and define the semantics of goals in terms of time points (see e.g. [227]). It is outside the scope of this chapter to discuss this solution, and additional research is needed to address this issue. A similar approach is used in [396].

where α is either a user-specified action or an **insert** or **adopt** action.[16]

For the case where a **drop** action is performed, two additional cases need to be distinguished: (i) the case where the goal of the agent is dropped, and (ii) the case where the goal is not dropped. A goal ϕ is dropped by performing **drop**(ϕ') only if $\phi \models_c \phi'$. As we cannot express in our verification logic that ϕ' is a logical consequence of ϕ, we introduce two inference rules to represent cases (i) and (ii).

$$\frac{\phi \models_c \phi'}{\Box(\textbf{o-goal}(\phi) \wedge \bigcirc \mathit{done}(\textbf{drop}(\phi'))) \rightarrow \bigcirc \neg \textbf{goal}(\top)} \tag{8.10}$$

$$\frac{\phi \not\models_c \phi'}{\Box(\textbf{o-goal}(\phi) \wedge \bigcirc \mathit{done}(\textbf{drop}(\phi'))) \rightarrow \bigcirc \textbf{o-goal}(\phi)} \tag{8.11}$$

Finally, we need to treat the case where the agent did not have any goals. That is, we need axioms that characterize the persistence of the absence of goals. Only performing an **adopt** action can add a goal, and the following axioms characterize respectively goal adoption and the persistence of the absence of any goals.

$$\forall \mathbf{x} \Box(\neg \textbf{goal}(\top) \wedge \bigcirc \mathit{done}(\textbf{adopt}(\phi))) \rightarrow \bigcirc \textbf{o-goal}(\phi) \tag{8.12}$$

$$\forall \mathbf{x} \Box(\neg \textbf{goal}(\top) \wedge \bigcirc \mathit{done}(\alpha)) \rightarrow \bigcirc \neg \textbf{goal}(\top) \tag{8.13}$$

where α is either a user-specified action or one of the built-in actions **insert** or **drop**.

Definition 8.24. *(Logical Representation of an Agent)*
The *logical representation* of a GOAL agent $Agt = \langle K, \Sigma, \Gamma, \Pi, A \rangle$, denoted by $Rep(Agt)$, is the set of axioms listed in Table 8.3.

In order to make precise what we mean by a run that corresponds to a trace, we introduce the following definitions.

Definition 8.25. *(Run Corresponds with Trace)*
Let $Agt = \langle K, \Sigma, \Gamma, \Pi, A \rangle$ be a GOAL agent. A run r *corresponds* with a trace t, written $r \approx t$, if for each $i \in \mathbb{N}$, with $r_i^m = \langle K, \Sigma, \Gamma \rangle$ and $t_i^b = B$ and $t_i^g = G$:

- $[\![K]\!](\Sigma) = \bigcap B$,
- $\forall \gamma \in \Gamma : \exists g \in t_i^g : \gamma = \bigcap g$, and $\forall g \in t_i^g : \exists \gamma \in \Gamma : \gamma = \bigcap g$, and
- $r_i^a = t_i^a$.

We also write $\mathcal{R} \approx T$ for a set of runs $\mathcal{R}$ and traces T if for all runs there is a trace that corresponds with it, and vice versa.

[16] In line with our assumption that an agent only has a single goal, an **adopt** action cannot be performed when the agent already has a goal.

For any Goal agent $\langle K,\Sigma,\Gamma,\Pi,A\rangle$ with a goal base Γ with exactly one goal, the following set of axioms is a logical representation of this agent, in combination with inference rules (10) and (11). (See also Theorem 8.2.)

knowledge base	$\Box\mathbf{bel}(comp(K))$
belief base	$\forall\mathbf{x}(\mathbf{bel}(b(\mathbf{x})) \leftrightarrow (\mathbf{x}=\mathbf{t}_1 \vee \ldots \vee \mathbf{x}=\mathbf{t}_n))$
goal base	$\mathbf{o\text{-}goal}(\phi)$
action rules	$\forall\mathbf{x}\Box(\bigcirc\mathbf{done}(a(\mathbf{x})) \rightarrow enabled(a(\mathbf{x}))$
	$\forall\mathbf{x}_1,\ldots,\mathbf{x}_n\Box(\bigcirc\mathbf{done}(\mathbf{skip}) \leftrightarrow (\neg enabled(a_1(\mathbf{x}_1)) \wedge \ldots \wedge \neg enabled(a_n(\mathbf{x}_n))))$
	$\forall\mathbf{x}_1,\ldots,\mathbf{x}_n\Box\bigcirc(\mathbf{done}(a_1(\mathbf{x}_1)) \vee \ldots \vee \mathbf{done}(a_n(\mathbf{x}_n)) \vee \mathbf{done}(\mathbf{skip}))$
action specs	$\forall\mathbf{x}\Box(\bigcirc\mathbf{bel}(p(\mathbf{x})) \leftrightarrow (A_p^+ \vee (\mathbf{bel}(p(\mathbf{x})) \wedge \neg A_p^-)))$
goal dynamics	$\forall\mathbf{x}\Box(\mathbf{o\text{-}goal}(\phi) \wedge \bigcirc\mathbf{bel}(\phi)) \rightarrow \bigcirc\neg\mathbf{goal}(\top)$
	$\forall\mathbf{x}\Box(\mathbf{o\text{-}goal}(\phi) \wedge \bigcirc(\neg\mathbf{bel}(\phi) \wedge done(\alpha)) \rightarrow \bigcirc\mathbf{o\text{-}goal}(\phi)$
	where α is either a user-specified action or an **insert** or **adopt** action
	$\forall\mathbf{x}\Box(\neg\mathbf{goal}(\top) \wedge \bigcirc done(\mathbf{adopt}(\phi))) \rightarrow \bigcirc\mathbf{o\text{-}goal}(\phi)$
	$\forall\mathbf{x}\Box(\neg\mathbf{goal}(\top) \wedge \bigcirc done(\alpha)) \rightarrow \bigcirc\neg\mathbf{goal}(\top)$
	where α is either a user-specified action or an **insert** or **drop** action.

Table 8.3 Logical Representation of a Goal Agent

The following theorem states that the set of axioms obtained by means of the "translation" procedure discussed above characterizes a Goal agent completely, in the sense that runs of the agent correspond to traces that are models of the logical representation.

Theorem 8.2. (Logical Representation Characterizes Agent)
Let $Agt = \langle K,\Sigma,\Gamma,\Pi,A\rangle$ be a Goal *agent and $\mathcal{R}$ denote the meaning of this agent. Let $Rep(Agt)$ be the corresponding logical representation of the agent. Then we have:*

$$\mathcal{R} \approx \{t \mid t \models_G Rep(Agt)\}$$

Proof. The proof proceeds in three steps and is based on induction. First, we show that a state representation $r(i)$ in a run corresponds with a time point (t,i) in a trace. Second, we show that if there is a run in which action α is executed at i then there is also a corresponding trace that executes α at i. Third, we show that the resulting state representation at the next point $r(i+1)$ corresponds with the time point $(t,i+1)$.

For the base case, we need to show that the initial mental state $r_0^m = \langle K,\Sigma_0,\Gamma_0\rangle$ corresponds with time points $(t,0) = \langle B_0,G_0,\alpha_0\rangle$ for arbitrary traces t that are models of $Rep(Agt)$. We need to show that:

- $[\![K]\!](\Sigma_0) = \bigcap B_0$. *Use axiom (2) above to show that basic facts match, and use axiom (1) and the proof of Theorem 8.1 to demonstrate that derived facts also match.*

- $\Gamma = \bigcap G_0$.[17] *For this case, use axiom (7).*

Below, we assume $r_i^m \approx (t,i)$ *for all* $0 \leq i \leq n$ *as induction hypothesis (IH).*

Next we show that whenever α *is executed in r at point i, then it is also executed at time point* (t,i) *for some trace t that corresponds for all time points with* $i \leq n$*, and vice versa. So, suppose that* α *is executed in the run at point i. This means that there is an action rule* ***if*** ψ ***then*** α *such that* $r_i^m \models_\Psi \psi \wedge \textbf{\textit{bel}}(pre(\alpha))$*. It follows that we also have* $(t,i) \models_G \psi \wedge \textbf{\textit{bel}}(pre(\alpha))$ *by the IH. We have that axiom (3) is also satisfied if* $t_i^a = \alpha$ *and t is a model of* $Rep(Agt)$*. The other direction is similar, using axiom (3) once more in combination with axiom (5). In case* $\alpha =$ ***skip****, we need axiom (4) instead.*

Finally, we need to show that the effects of executing an action produce corresponding states in the run and trace. As knowledge does not change, it is sufficient to show that basic facts are updated correspondingly. It follows using axiom (6) that the beliefs of an agent in a run correspond with that in the trace. For the single goal of the agent, use axioms and inference rules (7-13) to show that updates correspond. □

Concluding, we have established that GOAL agents can be said to *execute* particular logical specifications, i.e. their logical representations. These logical representations fully characterize a GOAL agent. As a result, any properties that logically follow from its logical representation are properties of that GOAL agent. A logical representation can be used and provided as input to a model checker to verify a property χ or to a theorem prover to deduce χ.

8.5 Conclusion

We have discussed a logic for reasoning about agents with *declarative* goals. Our aim has been to provide a logic that facilitates the verification of computational agents that derive their choice of action from their beliefs and goals. To this end we have presented a verification logic for GOAL agents based on linear temporal logic. The agent programming language GOAL enables programming computational agents that have a declarative beliefs and goals.

It has turned out to be a difficult task to fully characterize a GOAL agent by means of logic. The main reason is that the logic of goals is hard to capture formally. Although on the one hand goals seem to have some obvious logical properties, on the other hand a goal seems to be not much more than a resource that is being used by an agent. The main issue here is to express that an agent does *not* have certain goals, and how to provide a frame axiom for goals. It may be possible to do so by characterizing goals by temporal formula (see e.g. [227]). The presentation of our

[17] Recall we have assumed there is a single goal only, which allows us to simplify somewhat here.

results is intended to further stimulate research in this area which is so important for agents with motivational attitudes.

Even though we were able to fully characterize GOAL agents logically, provided that such agents only have a single goal, we did not prove completeness of the verification logic $\mathcal{L}_G$. Providing a complete axiomatization remains for future work. A complete axiomatization is important since without it we are still not sure that we can prove all properties of an agent by means of deduction.

One of the aims of this chapter has been to show that additional research is needed in order to provide the tools to reason about goals in agent programming. This is still true for agents that only have achievement goals, the type of goals that we discussed here. But this is even more true when other types of goals are concerned, such as maintenance goals [129, 152, 227].

Finally, although there is work that deals with verifying multi-agent systems, as far as we know there is no work that combines multi-agent with cognitive agents. A considerable challenge thus remains to provide a logic to verify computational multi-agent systems, and extend work on single-agent logics to multi-agent logics.

Acknowledgements The work presented would not have been possible without many useful discussions and comments from, in particular, Frank de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. I'd like to thank them here for the interesting exchanges on the topic of verification.

Appendix

This Appendix contains proofs for some additional Propositions and Theorem 8.1 in the main text. We first provide an alternative semantics for stratified axiom sets. This semantics is more convenient as it allows for the use of induction in the proofs below.

Alternative Fix Point Semantics

It will be convenient to write $F \models \phi$ for a set of facts $F = S \cup D$ instead of $\langle S, D\rangle \models \phi$, where S is a set of basic facts and D a set of derived facts; $F \models \phi$ can be viewed simply as a notational convenience, as shorthand for $\langle S, D\rangle \models \phi$.

Definition 8.26. Let A be a stratified axiom set, and $\{A_i, 1 \leq i \leq n\}$ be a partition that stratifies A. Let S be a set of basic facts and D a set of derived facts. Then for $0 \leq i \leq n$ define:

$$\begin{array}{ll} T_i(S,D) & = \{d_i(\mathbf{t}) \mid \forall \mathbf{x}(\phi \rightarrow d(\mathbf{x})) \in A_i, \langle S, D\rangle \models \phi[\mathbf{x} \leftarrow \mathbf{t}], \mathbf{t} \text{ are ground } \} \cup D \\ T_0^{\infty}(S) & = \emptyset \\ T_i^0(S) & = T_{i-1}^{\infty}(S) \text{ for } i > 0 \\ T_i^j(S) & = T_i(S, T_i^{j-1}(S)) \text{ for } i, j > 0 \end{array}$$

where $T_i^{\infty}(S) = \bigcup_{j \in \mathbb{N}} T_i^j(S)$ for $i > 0$.

Proposition 8.2. *Let S be a set of basic facts. For $1 \leq i \leq n$, we have:*

$$T_i^{\infty}(S) = [\![A]\!]_i(S)$$

Proof. By induction on i. For the base case $i = 1$, we need to show that $T_1^{\infty}(S) = [\![A]\!]_1(S)$. We first show that:

$$\bigcup_{a \in A_1} [\![a]\!](S, T_1^{\infty}(S)) \subseteq T_1^{\infty}(S)$$

Or, equivalenty, that $\{d(\mathbf{t}) \mid \langle S, T_1^{\infty}(S)\rangle \models \phi[\mathbf{t}], \mathbf{t}$ is ground$\} \subseteq T_1^{\infty}(S)$ for all axioms $\forall \mathbf{x}(\phi \rightarrow d(\mathbf{t})) \in A_1$. If $\langle S, T_1^{\infty}(S)\rangle \models \phi[\mathbf{t}]$, we must have that $\langle S, T_1^j(S)\rangle \models \phi[\mathbf{t}]$ for some $j \in \mathbb{N}$. But then $d(\mathbf{t}) \in T_1(S, T_1^j(S)) = T_1^{j+1}(S) \subseteq T_1^{\infty}(S)$, and we are done.

What remains is to show that for any D such that $\bigcup_{a \in A_1} [\![a]\!](S, D) \subseteq D$, we have $T_1^{\infty}(S) \subseteq D$. Clearly, we have $T_1^0(S) = \emptyset \subseteq D$. So, to arrive at a contraduction, suppose that $T_1^{\infty}(S) \not\subseteq D$. Then there must be a largest $j \in \mathbb{N}$ such that $T_1^j(S) \subseteq D$ but $T_1^{j+1}(S) \not\subseteq D$. But this means that $T_i(S, T_1^j(S))$ is non-empty, which implies that for

some axiom a we have that $[\![a]\!](S,D)$ is non-empty and $[\![a]\!](S,D) \subseteq T_i(S, T_1^j(S))$. A contradiction.

The case for $i > 1$ is similar. □

Proofs

Lemma 8.1. *Let A be a stratified axiom set and M be a model for $una \cup dca \cup S \cup comp(A)$. Then:*

1. *M is a Herbrand model (up to isomorphism).*
2. *M satisfies $S \cup [\![A]\!](S)$.*

Proof.

1. *As $M = \langle D, I\rangle$ satisfies $una \wedge dca$, we must have that D is isomorph with $\mathcal{T}$.*
2. *By induction on the number of partitions k of A that stratify A. The base case $k = 0$ is trivial as in that case $[\![A]\!](S) = \emptyset$. For the inductive case, suppose we have $M \models S \cup [\![A]\!]_k(S)$. We need to show that $M \models [\![A]\!]_{k+1}(S)$, or, equivalently (by Proposition 8.26) $M \models T_{k+1}^{\infty}(S)$. We show $M \models T_{k+1}^{j}(S)$ for all $j \in \mathbb{N}$ by induction on j. The base case, $j = 0$, is trivial as in that case $T_{k+1}^{0}(S) = T_k^{\infty}(S)$. For $j > 0$, we assume that $M \models T_{k+1}^{j-1}(S)$ as induction hypothesis. By Definition 8.26 of T_{k+1} it follows immediately that $M \models T_{k+1}^{j}(S) = T_{k+1}(S, T_{k+1}^{j-1}(S))$, and we are done.*

□

Lemma 8.2. *$S \cup [\![A]\!](S)$ is a Herbrand model for $una \cup dca \cup S \cup comp(A)$.*

Proof. Clearly, $S \cup [\![A]\!](S)$ is a Herbrand model for $una \cup dca \cup S$. □

Theorem 8.3. *Let $S \subseteq F$ be a set of basic facts and A be a stratified axiom set. Let ϕ be a closed formula. Then we have that if $una \cup dca \cup S \cup comp(A) \cup \phi$ is satisfiable, then $S \cup [\![A]\!](S)$ is a Herbrand model for $una \cup dca \cup S \cup comp(A) \cup \phi$.*

Proof. The proof proceeds by induction on the maximum level k of the stratification of A.

For the base case $k = 0$ we have that A must be a definite

□

Lemma 8.3. *Let S be a set of ground atoms and A be a stratified axiom set. Then we have that $S \cup [\![A]\!](S)$ is a (minimal) Herbrand model of $comp(A)$.*

Proof. Note that $S \cup [\![A]\!](S)$ *can be interpreted as a Herbrand model H. Simply define the domain as the set of closed terms, and define an interpretation that on the set of terms is the identity function and maps ground atoms to true if and only if they are an element of* $S \cup [\![A]\!](S)$*. See [289], p. 111.* □

Finally, we can provide the proof for Theorem 8.1. Let S be a set of ground atoms, A be a stratified axiom set, and θ be an answer for ϕ with respect to S and A. Then θ is a correct answer.

Proof. We have to show that:

$$S \models_A \forall(\phi\theta) \text{ iff } S \cup una \cup dca \cup comp(A) \models \forall(\phi\theta)$$

Let B denote the Herbrand base for $\mathcal{L}_0$. We use $\neg F$ to denote the set of all negated formulas from a set of formulas F, i.e. $\{\neg\phi \mid \phi \in F\}$. Let $T = S \cup [\![A]\!](S)$. For the left to right direction, it is sufficient to show that $S \cup una \cup dca \cup comp(A)$ entails $T \cup \neg(B \setminus T)$.

For the right to left direction, as above, we use $S \cup [\![A]\!](S)$ to construct a corresponding Herbrand model H and show that H is a model of $una \cup dca \cup comp(A) \cup S$. By definition, we have $H \models una$ and $H \models dca$, as the domain of H is the set of terms $\mathcal{T}$. It is also obvious that $H \models S$. Finally, using Lemma 8.3 it follows that H is also a model of $comp(A)$.

□

Chapter 9

Using the Maude Term Rewriting Language for Agent Development with Formal Foundations

M.B. van Riemsdijk, L. Aştefănoaei, and F.S. de Boer

Abstract We advocate the use of the Maude term rewriting language and its supporting tools for prototyping, model-checking, and testing agent programming languages and agent programs. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. We use the agent programming language BUpL (Belief Update programming Language) for illustration.

M.B. van Riemsdijk
Delft University of Technology, The Netherlands e-mail: m.b.vanriemsdijk@tudelft.nl

L. Astefanoaei
CWI (Centrum voor Wiskunde en Informatica), The Netherlands e-mail: L.Astefanoaei@cwi.nl

F. de Boer
CWI (Centrum voor Wiskunde en Informatica), The Netherlands e-mail: F.S.de.Boer@cwi.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_9,

9.1 Introduction

An important line of research in the agent systems field is research on *agent programming languages* [64]. The guiding idea behind these languages is that programming languages based on agent-specific concepts such as beliefs, goals, and plans facilitate the programming of agents.

Several agent programming languages have been developed with an emphasis on the use of *formal methods*. In particular, structural operational semantics [340] is often used for formally defining the semantics of the languages. The semantics is used as a basis for prototyping and implementing the languages, and for verification. Several tools and techniques can be used for implementation and verification, such as Java for writing an interpreter and IDE, and the Java PathFinder[1] or SPIN [236] model-checkers for verification [61].

In this chapter, we advocate the use of the *Maude* language [104] and its supporting tools for prototyping, verifying, and testing agent programming languages and agent programs. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. Maude is a high-performance reflective language and system supporting *equational and rewriting logic specification and programming*. The language has been shown to be suitable both as a logical framework in which many other logics can be represented, and as a semantic framework through which programming languages with an operational semantics can be implemented in a rigorous way [301]. Maude comes with an LTL model-checker [155], which allows for verification. Moreover, Maude facilitates the specification of strategies for controlling the application of rewrite rules [154].

We will demonstrate how these features of Maude can specifically be applied for developing *agent* programming languages and programs based on solid formal foundations. We use the agent programming language BUpL (Belief Update programming Language) [19] for illustration. BUpL is a simple language that resembles the first version of 3APL [223].

The outline of this chapter is as follows. We present BUpL in Section 9.2, and then use BUpL to illustrate how Maude can be used for prototyping (Section 9.3), model-checking (Section 9.4), and testing (Section 9.5). We conclude in Section 9.6. The complete Maude source code of the implementations discussed in this chapter can be downloaded from `http://homepages.cwi.nl/~astefano/agents/bupl-strategies.php`.

[1] `http://javapathfinder.sourceforge.net/`

9.2 The BUpL Language

In this section, we briefly present the syntax and semantics of BUpL for ease of reference. We refer to Chapter [2] for more details and explanation. A BUpL agent has an initial belief base and an initial plan. A belief base is a collection of ground (first-order) atomic formulas which we refer to as beliefs. The agent is supposed to execute its initial plan, which is a sequential composition and/or a non-deterministic choice of actions or composed plans. The semantics of actions is defined using preconditions and effects (postconditions). An action can be executed if the precondition of the action matches the belief base. The belief base is then updated by adding or removing the elements specified in the effect. If the precondition does not match the belief base, we say the execution of the action (or the plan of which it is a part) fails. In this case repair rules can be applied, and this results in replacing the plan that failed by another.

9.2.1 Syntax

BUpL is based on a simple logical language $\mathcal{L}$ with typical element φ, which is defined as follows. $\mathcal{F}$ and $\mathcal{P}red$ are infinite sets of function, respectively predicate symbols, with typical element f, respectively P. Variables are denoted by the symbol x. As usual, a term t is either a variable or a function symbol with terms as parameters. Predicate symbols with terms as parameters form the *atoms* of $\mathcal{L}$, and atoms or negated atoms are called *literals*, denoted as l. Atoms are also called *positive literals* and negated atoms are called *negative literals*. The negation of a negative literal yields its positive variant. Nullary functions form the constants of the language. $\mathcal{L}$ does not contain quantifiers to bind variables. A formula or term without variables is called *ground*. Formulas from $\mathcal{L}$ are in disjunctive normal form (DNF), i.e., they consist of disjunctions of conjunctions (denoted as c) of literals. A *belief base* $\mathcal{B}$ is a set of ground atoms from $\mathcal{L}$.

$$\begin{aligned} t &::= x \mid f(t,\ldots,t) \\ l &::= P(t,\ldots,t) \mid \neg P(t,\ldots,t) \\ c &::= l \mid c \wedge c \\ \varphi &::= c \mid c \vee c \end{aligned}$$

Basic actions are defined as functions $a(x_1,\ldots,x_n) =_{def} (\varphi,\xi)$, where $\varphi \in \mathcal{L}$ is a formula which we call *precondition*, and ξ is a set of literals from $\mathcal{L}$ which we call *effect*. The following inclusions are required:

$$Var(\xi) \subseteq Var(\varphi) = \{x_1,\ldots,x_n\}.^{2}$$

[2] These inclusions thus specify that the variables occurring in the precondition and the effect also have to be in the parameter of the action. This has to do with the way actions are implemented

We use the symbol $\mathcal{A}$ for the set of basic actions. We use *Act* to refer to the set of basic action *names*, with typical element a. We say that a function call $a(t_1,\ldots,t_n)$ is a *basic action term*, and will sometimes denote it as α.

Plans, typically denoted as p, are defined as follows, where Π is a set of plan names with typical element π and $a(t,\ldots,t)$ is a basic action term:

$$p ::= a(t,\ldots,t) \mid \pi(t,\ldots,t) \mid a(t,\ldots,t); p \mid p + p.$$

Here, ';' is the *sequential composition* operator and '+' is the *choice* operator, with a lower priority than ';'. The construct $\pi(t,\ldots,t)$ is called *abstract plan*. Abstract plans should be understood as procedure calls in imperative languages, with corresponding procedures of the form $\pi(x_1,\ldots,x_n) = p$. The set of procedures is denoted as $\mathcal{P}$.

Repair rules have the form $\varphi \leftarrow p$, and can be applied if a plan has failed and φ matches the belief base. Then the failed plan is substituted by p. The set of repair rules is denoted as $\mathcal{R}$.

A BUpL agent is a tuple $(\mathcal{B}_0, p_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{B}_0$ is the initial belief base, p_0 is the initial plan, $\mathcal{A}$ are the actions, $\mathcal{P}$ are the procedures, and $\mathcal{R}$ are the repair rules. The initial belief base and plan form the initial *mental state* of the agent.

To illustrate the above syntax, we take as an example a BUpL agent that solves the *tower of blocks* problem, i.e., the agent has to build towers of blocks. We represent blocks by natural numbers. Assume the following initial arrangement of three blocks 1,2, and 3: blocks 1 and 2 are on the table (denoted as block 0), and 3 is on top of 1. The agent has to rearrange them such that they form the tower 321 (1 is on 0, 2 on top of 1 and 3 on top of 2). The only action an agent can execute is $move(x,y,z)$ to move a block x from another block y onto z, if both x and z are clear (i.e., have no blocks on top of them). Blocks can always be moved onto the table, i.e., the table is always clear.

$$\mathcal{B}_0 = \{\ on(3,1),\ on(1,0),\ on(2,0),\ clear(2),\ clear(3),\ clear(0)\ \}$$

$$p_0 = build$$

$$\mathcal{A} = \{\ move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z), \{on(x,z), \neg on(x,y), \neg clear(z), clear(0)\}\ )\ \}$$

$$\mathcal{P} = \{build = move(2,0,1); move(3,0,2)\}$$

$$\mathcal{R} = \{\ on(x,y) \leftarrow move(x,y,0); build\ \}$$

Fig. 9.1 A BUpL Blocks World Agent

The BUpL agent from Figure 9.1 is modeled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally telling the agent to move

in Maude. It may be relaxed, but for simplicity, we do not do it here. In other languages such as 2APL [122], variables may occur in the precondition that are not in the parameters of the action. These variables are instantiated when matching the precondition against the belief base.

block 2 onto 1. This is not possible, since block 3 is already on top of 1. Similar scenarios can easily arise in multi-agent systems: imagine that initially 3 is on the table, and the agent decides to move 2 onto 1; imagine also that another agent comes and moves 3 on top of 1, thus moving 2 onto 1 will fail. The failure is handled by the repair rule $on(x,y) \leftarrow move(x,y,0); p$. Choosing $[x/3][y/1]$ as a substitution, this enables the agent to move block 3 onto the table and then the initial plan can be restarted.

9.2.2 Semantics

First, we define the satisfaction relation of formulas φ with respect to a belief base $\mathcal{B}$. For this, we consider the usual notion of *substitution* as a set that defines how to replace variables with terms. A substitution is denoted by $[x_0/t_0]\ldots[x_n/t_n]$, which expresses that x_i is replaced by t_i for $0 \leq i \leq n$. A substitution θ can be applied to a formula φ, written as $\varphi\theta$, which yields the formula φ in which variables are simultaneously replaced by terms as specified by θ. If θ and θ' are substitutions and φ is a formula, we use $\varphi\theta\theta'$ to denote $(\varphi\theta)\theta'$. A *ground substitution* is a substitution in which all t_i are ground terms. In the sequel, we will assume all substitutions to be ground, unless indicated otherwise. For technical convenience, we assume any conjunction c has the form $l_0 \wedge \ldots \wedge l_m \wedge l_{m+1} \wedge \ldots \wedge l_n$ where $l_0,\ldots,l_m$ are positive literals and $l_{m+1},\ldots,l_n$ are negative literals. We use $Var(\varphi)$ to denote the variables occurring in φ and $dom(\theta)$ to denote the set of variables forming the domain of θ. The satisfaction relation of a formula φ with respect to a belief base is defined relative to a substitution θ, denoted as $\models_\theta$, and is defined as follows, where $\models$ is the usual entailment relation for ground formulas:

$\mathcal{B} \models_\emptyset P(t_0,\ldots,t_n)$ iff $P(t_0,\ldots,t_n)$ is ground and $\mathcal{B} \models P(t_0,\ldots,t_n)$
$\mathcal{B} \models_\emptyset c$ iff c is ground and $\mathcal{B} \models c$
$\mathcal{B} \models_\theta P(t_0,\ldots,t_n)$ iff $\mathcal{B} \models_\emptyset P(t_0,\ldots,t_n)\theta$ and $Var(P(t_0,\ldots,t_n)) = dom(\theta)$
$\mathcal{B} \models_\theta c$ iff $\mathcal{B} \models_\emptyset (l_0 \wedge \ldots \wedge l_m)\theta$ and $Var(l_0 \wedge \ldots \wedge l_m) = dom(\theta)$ and
$\neg\exists\theta' : (\mathcal{B} \models_{\theta'} \neg l_{m+1}\theta$ and $Var(l_{m+1}\theta) = dom(\theta'))$
…
$\neg\exists\theta' : (\mathcal{B} \models_{\theta'} \neg l_n\theta$ and $Var(l_n\theta) = dom(\theta'))$
$\mathcal{B} \models_\theta c \vee c'$ iff $\mathcal{B} \models_\theta c$ or $\mathcal{B} \models_\theta c'$

We use $Sols(\mathcal{B},\varphi) = \{\theta \mid \mathcal{B} \models_\theta \varphi\}$ to denote the set of all substitutions for which φ follows from the belief base. Note that if $Sols(\mathcal{B},\varphi) = \emptyset$, φ does not follow from $\mathcal{B}$. If φ follows from $\mathcal{B}$ under the empty substitution, we have $Sols(\mathcal{B},\varphi) = \{\emptyset\}$.

We now continue to define what it means to execute an action. Let $a(x_1,\ldots,x_n) =_{def} (\varphi,\xi) \in \mathcal{A}$ be a basic action definition. A function call $a(t_1,\ldots,t_n)$ yields the pair $(\varphi,\xi)\theta$ where $\theta = [x_1/t_1]\ldots[x_n/t_n]$, which does not have to be ground. Let $a(t_1,\ldots,t_n) = (\varphi',\xi')$ be the result of applying the function to the terms $t_1,\ldots,t_n$, and let $\theta' \in Sols(\mathcal{B},\varphi')$. Then the effect on $\mathcal{B}$ of executing $a(t_1,\ldots,t_n)$ is that $\mathcal{B}$ is

updated by adding or removing (ground) atoms occurring in the set $\xi'\theta'$:

$$\begin{aligned} \mathcal{B} \uplus l\theta' &= \mathcal{B} \cup l\theta' \quad \text{if } l \in \xi \text{ and } l \text{ a positive literal} \\ \mathcal{B} \uplus l\theta' &= \mathcal{B} \setminus \neg l\theta' \quad \text{if } l \in \xi \text{ and } l \text{ a negative literal} \end{aligned}$$

We write $\mathcal{B} \uplus \xi'\theta'$ to represent the result of updating $\mathcal{B}$ with the instantiated effect of an action $\xi'\theta'$, which performs the update operation as specified above on $\mathcal{B}$ for each literal in ξ'. This update is guaranteed to yield a consistent belief base since we add only positive literals.

The operational semantics of a language is usually defined in terms of labeled transition systems [340]. A *labeled transition system* (LTS) is a tuple $(\Sigma, s_0, L, \rightarrow)$, where Σ is a set of states, s_0 is an initial state, L is a set of labels, and $\rightarrow \subseteq \Sigma \times L \times \Sigma$ describes all possible transitions between states, and associates a label to the transition. The notation $s \xrightarrow{\alpha} s'$ expresses that $(s, \alpha, s') \in \rightarrow$, and it intuitively means that "s becomes s' by performing action α". Invisible transitions are denoted by the label τ.

The operational semantics for a BUpL agent is defined as follows. Let $(\mathcal{B}_0, p_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$ be a BUpL agent. Then the associated LTS is $(\Sigma, (\mathcal{B}_0, p_0), L, \rightarrow)$, where:

- Σ is the set of states, which are BUpL mental states
- $(\mathcal{B}_0, p_0)$ is the initial state
- L is a set of labels, which are either ground basic action terms or τ
- $\rightarrow$ is the transition relation induced by the transition rules given in Table 9.1.

$$\frac{a(x_1,\ldots,x_n) =_{def} (\varphi,\xi) \in \mathcal{A} \qquad a(t_1,\ldots,t_n) = (\varphi',\xi') \qquad \theta \in Sols(\mathcal{B},\varphi')}{(\mathcal{B}, a(t_1,\ldots,t_n); p') \xrightarrow{a(t_1,\ldots,t_n)\theta} (\mathcal{B} \uplus \xi'\theta, p'\theta)} \ (act)$$

$$\frac{(\mathcal{B}, p_i) \xrightarrow{\mu} (\mathcal{B}', p')}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\mu} (\mathcal{B}', p')} \ (sum_i)$$

$$\frac{(\mathcal{B}, \alpha; p) \not/ \alpha\theta' \rightarrow \qquad \varphi \leftarrow p' \in \mathcal{R} \qquad \theta \in Sols(\mathcal{B},\varphi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \ (fail)$$

$$\frac{\pi(x_1,\ldots,x_n) = p \in \mathcal{P}}{(\mathcal{B}, \pi(t_1,\ldots,t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1,\ldots,t_n))} \ (\pi)$$

Table 9.1 BUpL transition rules

In rule (sum_i), p_i is either p_1 or p_2, and μ can be either a ground basic action term or a silent transition τ, in which case $\mathcal{B}' = \mathcal{B}$, and p' is a valid repair plan. In rule (π), $p(t_1,\ldots,t_n)$ stands for $p[x_1/t_1]\ldots[x_n/t_n]$.

9.3 Prototyping

In this section, we describe how the operational semantics of agent programming languages can be implemented in Maude. The main advantage of using Maude for this is that the translation of operational semantics into Maude is direct [390], ensuring a *faithful implementation*. Because of this, it is relatively easy to experiment with different kinds of semantics, making Maude suitable for rapid *prototyping* of agent programming languages. This is also facilitated by the fact that Maude supports user-definable syntax, offering prototype parsers for free. Another advantage of using Maude for prototyping specifically *logic-based* agent programming languages is that Maude has been shown to be suitable not only as a semantic framework, but also as a logical framework in which many other logics can be represented.

We use BUpL to illustrate the implementation of agent programming languages in Maude. BUpL has beliefs and plan revision features, but no goals. We refer to [370] for a description of the Maude implementation of a similar agent programming language that does have goals. While the language of [370] is based on propositional logic, BUpL allows the use of variables, facilitating experimentation with more realistic programming examples. An implementation of the agent programming language AgentSpeak in Maude is briefly described in [170].

9.3.1 Introduction to Maude

A rewriting logic specification or rewrite theory is a tuple $\langle \Sigma, E, R \rangle$, where Σ is a signature consisting of types and function symbols, E is a set of equations and R is a set of rewrite rules. The signature describes the *terms* that form the state of the system. These terms can be rewritten using equations and rewrite rules. Rewrite rules are used to model the dynamics of the system, i.e., they describe transitions between states. Equations form the functional part of a rewrite theory, and are used to reduce terms to their "normal form" before they are rewritten using rewrite rules. The application of rewrite rules is intrinsically non-deterministic, which makes rewriting logic a good candidate for modeling concurrency.

In what follows, we briefly present the basic syntax of Maude, as needed for understanding the remainder of this section. Please refer to [104] for complete information. Maude programs are built from *modules*. A module consists of a *syntax declaration* and *statements*. The syntax declaration forms the signature and consists of declarations for *sorts*, which give names for the types of data, *subsorts*, which impose orderings on data types, and *operators*, which provide names for the operations acting upon the data. Statements are either equations or rewrite rules. Modules containing no rewrite rules but only equations are called *functional modules*, and they define equational theories $\langle \Sigma, E \rangle$. Modules that contain also rules are called *system modules* and they define rewrite theories $\langle \Sigma, E, R \rangle$. Functional modules (system modules) are declared as follows:

```
fmod (mod) <ModuleName> is

          <DeclarationsAndStatements>

endfm (endm)
```

Modules can import other modules, which helps in building up modular applications from short modules, making it easy to debug, maintain or extend.

One or multiple sorts are declared using the keywords `sort` and `sorts`, respectively, and subsorts are similarly declared using `subsort` and `subsorts`. The following defines the sorts `Action` and `Plan` and their subsort relation, which is used for specifying the BUpL syntax.

```
sorts Action Plan . subsort Action < Plan .
```

We can further declare operators (functions) defined on sorts (types) as follows:

```
op <OpName> : <Sort-1> ... <Sort-k>  -> <Sort>
                                    [<OperatorAttributes>] .
```

where `k` is the arity of the operator. For example, the operator declaration below is used to define the BUpL construct plan repair rule. The operator `((_<-_))` takes a query of sort `Query` that should be tested on the belief base, and a plan, and yields a term of sort `PRrule`. The operator is in mixfix form, where the underscores indicate the positions of its parameters. This also illustrates how Maude can be used to define the syntax of a BUpL language construct.

```
op ((_<-_)) : Query Plan -> PRrule .
```

Equations and rewrite rules specify how to transform terms. Terms are variables, constants, or the result of the application of an operator to a list of argument terms. Variables are declared using the keywords `var` and `vars`. For example, `var R : PRrule` declares a variable `R` of sort `PRrule`. Equations can be unconditional or conditional and are declared as follows, respectively:

```
eq  [<Label>] : <Term-1> = <Term-2> .
ceq [<Label>] : <Term-1> = <Term-2>
                         if <Cond-1> /\ ... /\ <Cond-k> .
```

where `Cond-i` is a condition which can be an ordinary equation `t = t'`, a matching equation `t := t'` (which is true only if the two terms match), a Boolean equation (which contains, e.g., the built-in (in)equality =/=, ==, and/or logical combinators such as `not`, `and`, `or`), or a membership equation `t : S` (which means that `t` is a member of sort `S`).

For example, the following conditional equation is part of a module for specifying when a formula logically follows from the belief base. The belief base is defined as a commutative sequence of ground belief atoms of sort `Belief`, separated by `#`. The conditional equation specifies that matching term `T` against a belief base containing belief `B` yields substitution `S`, if `match(T, B)` yields a substitution `S` that is different from `noMatch`, the built-in Maude constant to indicate that no substitution has been found.

```
var B : Belief .
var BB : BeliefBase .
var T : Term .
var S : Substitution .

ceq match(T, B # BB) = S if S := match(T, B) /\ S =/= noMatch .
```

Operationally, equations can be applied to a term from left to right. Equations in Maude are assumed to be terminating and confluent,[3] i.e., there is no infinite derivation from a term `t` using the equations, and if `t` can be reduced to different terms `t1` and `t2`, there is always a term `u` to which both `t1` and `t2` can be reduced. This means that any term has a *unique normal form*, to which it can be reduced using equations in a finite number of steps.

Finally, we introduce rewrite rules. Like equations, rewrite rules can also be unconditional or conditional, and are declared as follows:

```
rl  [<Label>] : <Term-1> => <Term-2> .
crl [<Label>] : <Term-1> => <Term-2>
                          if <Cond-1> /\ ... /\ <Cond-k>  .
```

where `Cond-i` can involve equations, memberships (which specify terms as having a given sort) and other rewrites. We will present several examples in the next section.

9.3.2 Implementing BUpL: Syntax

In this section, we use BUpL to illustrate how the syntax of agent programming languages can be implemented in Maude. We make a distinction between the logical parts of the language and the non-logical parts.

9.3.2.1 Logical Part

First, we have to define the logical language on which BUpL is based. Logical formulas occur in the belief base (ground atoms), in actions specifications (a formula as precondition, and a set of literals as effects), and in repair rules (a formula as the application condition). For the representation of atoms, the Maude built-in sorts `GroundTerm` and `Term` are used. That is, any Maude (ground) term can be used as an atom of our logical base language. In addition, we define the following sorts to represent also negated (ground) terms and (ground) sets of literals.

```
sorts NegGroundTerm NegTerm GroundLitSet LitSet .
```

[3] If this is not the case, the operational semantics of Maude does not correspond with its mathematical semantics.

The following subsort relations are defined on these sorts. Note that `GroundTerm < GroundLitSet` specifies that any Maude ground term can be a (set of) ground literals, and similarly for `Term < LitSet`.

```
subsorts GroundTerm GroundTermList < GroundLitSet .
subsorts Term NegTerm GroundLitSet < LitSet .
subsort  NegGroundTerm < NegTerm .
```

`GroundLitSet` is defined as a supersort of the Maude built-in sort `GroundTermList`, since we use its constant `empty` to represent an empty set of ground literals. The sorts `Belief` and `BeliefBase` are introduced with the subsort relations

```
subsorts Belief < GroundTerm GroundTermList < BeliefBase
                                            < GroundLitSet .
```

to represent beliefs. The following operators are introduced to syntactically represented (ground) literal sets, belief bases, and negated (ground) terms. The attributes `assoc comm id: empty` declare that the operator is associative and commutative with identity the empty set. The attribute `ctor` declares that the operator is a constructor, which means that it is used to construct terms rather than to apply it as a function and calculate the result. We overload the operator `#`, using it for representing both (ground) literal sets and belief bases. The attribute `ditto` specifies that an overloaded operator has the same attributes as the first declaration of the operator (excluding `ctor`).

```
op _#_ : LitSet LitSet -> LitSet [ctor assoc comm id: empty] .
op _#_ : GroundLitSet GroundLitSet -> GroundLitSet [ctor ditto] .
op _#_ : BeliefBase BeliefBase -> BeliefBase [ctor ditto] .

op neg_ : Term -> NegTerm [ctor] .
op neg_ : GroundTerm -> NegGroundTerm [ctor] .
```

We call formulas that are evaluated on the belief base *queries*. The query language is defined over terms as follows. The definition is more general than the DNF of Section 9.2.1. However, when defining the semantics, formulas are first transformed into DNF.

```
sort Query .
subsort Term < Query .

ops top bot : -> GroundTerm .
op ~_ : Query -> Query [ctor] .
op _/\_ : Query Query -> Query [assoc] .
op _\/_ : Query Query -> Query [assoc] .
```

This completes the specification of the syntax of the logical part of BUpL.

It is important to note that Maude is suitable as a framework in which many logics can be represented, using equations to axiomatize the logic and using rewrite rules as inference rules. This facilitates experimentation with different logics for representing agent beliefs, making the framework flexible.

9.3.2.2 Non-Logical Part

The non-logical part consists of the specification of actions, plans, procedures, and repair rules. We distinguish between internal and observable actions. This is useful for testing. Actions are specified as functions using equations. The action name is the function name specified as an operator, and applying the equation yields the precondition and effect of the action. Preconditions and effects are defined using the operators `o[_,_]` and `i[_,_]` for observable and internal actions, respectively. `nilA` is the "empty" action, used to define an empty plan. The code below shows an example specification of the `move` action from the tower of blocks example of Figure 9.1.[4] The sort `Nat` represents natural numbers.

```
sorts Action I-Action O-Action .
subsorts I-Action O-Action < Action .

ops nilA : -> Action .
op o[_,_] : Query LitSet -> O-Action .
op i[_,_] : Query LitSet -> I-Action .

op on : Nat Nat -> Belief .
op clear : Nat -> Belief .

op move : Nat Nat Nat -> O-Action .

ceq [act] : move(X, Y, Z) = o[on(X, Y) /\ clear(X) /\ clear(Z),
                              neg on(X, Y) # on(X, Z) # clear(Y)
                              # neg clear(Z) # clear(0)]
            if X =/= Z .
```

Plans are built from actions, procedure calls (at the end of a plan), sequential composition (`pre`), and non-deterministic choice (`sum`). The operators `pre` and `sum` are declared to be constructors, reflecting the fact that they are used to construct plans. Procedure names are introduced as operators, and a procedure is defined as an equation that yields the plan forming the body of the procedure. For example, the procedure `build` as declared below is used for building a tower of three blocks (`321`).

```
sort Plan .
subsort Action < Plan .

op pre : Action Plan -> Plan [ctor id: nilA strat (1 0)] .
op sum : Plan Plan -> Plan [ctor comm] .

op build : -> Plan .
eq build = pre(move(2, 0, 1), move(3, 0, 2)) .
```

Note that the operator `pre` has the attribute `strat (1 0)`. This specifies that only its first argument (an action) can be normalized using equations (expressed by the

[4] Note that in the specification of the `move` action in Maude, we have added the condition `X =/= Z`, which is easily done using conditional equations.

1), before any equations are applied on the operator `pre` itself (expressed by placing `1` before `0`).[5] The second argument (a plan) is not normalized using equations. Using this attribute thus changes what a normal form is for the operator `pre`: the normal form is obtained by normalizing the operator's first argument and then normalizing the operator itself at top level, while leaving the second argument intact. This prevents the continuous application of equations, which would lead to a stack overflow in case a non-terminating procedure is specified. For example, if we would specify a recursive procedure `build` using the equation

```
eq build = pre(move(2, 0, 1), pre(move(2, 1, 0), build)) .
```

without using `strat` in the declaration of `pre`, the continuous application of the equation to normalize `build` as occurring in the right-hand side of the equation would lead to a stack overflow.

Repair rules are defined similarly to procedures, using equations. An operator is introduced to define the name and parameters of the repair rule, and the equation yields the repair rule itself. On the basis of the equations, repair rules can be collected into a repair rule base (of sort `PRbase`). The example repair rule `pr` shown below can be used to deal with a failing `move(X,Y,Z)` action. The action fails if `Y` or `Z` are not clear. In this case the repair rule can be applied to move a block to the table (clearing the block on which it was placed), after which it is tried again to build the tower.

```
sorts PRrule PRbase .
subsort PRrule < PRbase .

op ((_<-_)) : Query Plan -> PRrule .
op empty-prb : -> PRbase .
op __ : PRbase PRbase -> PRbase [assoc comm id: empty-prb] .

ops pr : Nat Nat -> PRrule .
eq [pr] : pr(X, Y) = ((on(X, Y) /\ Y > 0 <-
                                    pre(move(X, Y, 0), build))) .
```

Finally, we define an operator for representing BUpL mental states. The operator takes a label, belief base and plan, and yields a term of sort `LBpMentalState`. The label represents the label of the transitions in the transition system, i.e., it represents which actions have been executed.

```
op <<_,_,_>> : Label BeliefBase Plan -> LBpMentalState .
```

9.3.3 Example BUpL Program

Using the implementation of the BUpL syntax in Maude, one can easily specify BUpL programs in Maude. An example is the following tower building agent, which

[5] In our implementation, no equations are specified for normalizing `pre` itself.

represents the example agent from Figure 9.1 in Maude. The `move` action and the procedure and plan repair rule have already been introduced above. In addition, the program specifies the initial belief base `bb`, which expresses where blocks are positioned initially and which blocks are clear. Moreover, the initial mental state of the builder agent is specified using the operator `builder`. The initial plan is `build`. Since no actions have been executed yet in the initial mental state, its label is empty. The equation `module-name` is specified to obtain a reference to the module in which the BUpL program is written. This will be used when implementing the semantics.

```
mod AGENT-DATA
    protecting BUPL-SYNTAX .
    protecting NAT .

    eq module-name = 'AGENT-DATA .

    op on : Nat Nat -> Belief .
    op clear : Nat -> Belief .

    op bb : -> BeliefBase .
    eq bb = on(3, 1) # on(1, 0) # on(2, 0) # clear(0) #
                                        clear(3) # clear(2) .

    op move : Nat Nat Nat -> O-Action .
    vars X Y Z : Nat .
    ceq [act] : move(X, Y, Z) =
                          o[on(X, Y) /\ clear(X) /\ clear(Z),
                            neg on(X, Y) # on(X, Z) # clear(Y)
                            # neg clear(Z) # clear(0)]
              if X =/= Z .

    op build : -> Plan .
    eq build = pre(move(2, 0, 1), move(3, 0, 2)) .

    ops pr : Nat Nat -> PRrule .
    eq [pr] : pr(X, Y) = ((on(X, Y) /\ Y > 0 <-
                                  pre(move(X, Y, 0), build))) .

    op builder : -> LBpMentalState .
    eq builder = << bLabel(empty), bb, build >> .
endm
```

9.3.4 Implementing BUpL: Semantics

The implementation of the semantics of BUpL in Maude can again be divided into the implementation of the logical part and of the non-logical part.

9.3.4.1 Logical Part

Implementing the semantics of the logical part means implementing matching a query against a belief base. Matching takes place both to determine whether an action can be executed, as well as to determine whether a repair rule can be applied. It is defined using the operator `match : Query BeliefBase -> Substitution`, which takes a query and a belief base, and yields a substitution in case the query matches the belief base, and the special substitution `noMatch` otherwise.

This operator is defined by making use of Maude's *reflective* capabilities [103]. Maude is a reflective logic since important aspects of its meta-theory can be represented at the object level, so that the object level correctly simulates the meta-theoretic aspects. The meta-theoretic aspect that we use here, is matching two terms. Maude continually matches terms when using equations and rewrite rules. This meta-level functionality can be conveniently used to match a term against a belief.

The meta-level operator that can be used for implementing this, is `metaMatch`. This operator takes the meta-representation of a module and two terms, and tries to match these terms in the module. If the matching attempt is successful, the result is the corresponding substitution. Otherwise, `noMatch` is returned. Obtaining the meta-representation of modules and terms can be done using the operators `upModule` and `upTerm`, respectively. The module that we use for this is the module containing the BUpL program, since the belief base is defined there. The name of the module is obtained by defining an equation for the operator `module-name`, as shown in the example program of Section 9.3.3. The sort `Qid` is a predefined Maude sort for identifiers. The base case for the operator `match`, where a term is matched against a belief, is defined using `metaMatch` as follows.

```
var T : Term .
var B : Belief .

op module-name : -> Qid .

eq match(T, B) =
       metaMatch(upModule(module-name), upTerm(T), upTerm(B)) .
```

Matching a term against a belief base is then defined by making use of the former equation.

```
var S : Substitution .
var BB : BeliefBase .

ceq match(T, B # BB) = S if S := match(T, B) /\ S =/= noMatch .
eq  match(T, B # BB) = noMatch [owise] .
```

For reasons of space, we omit the additional equations for matching composite formulas against a belief base.

9.3.4.2 Non-Logical Part

As proposed in [424], the general idea of implementing transition rules of an operational semantics in Maude, is to implement them as (conditional) rewrite rules. The premises of a transition rule then form the conditions of the corresponding rewrite rule, and the conclusion forms the rewrite itself.

We illustrate the implementation of transition rules using those for action execution and repair rule application. The transition rule for action execution

$$\frac{a(x_1,\ldots,x_n) =_{def} (\varphi,\xi) \in \mathcal{A} \qquad a(t_1,\ldots,t_n) = (\varphi',\xi') \qquad \theta \in Sols(\mathcal{B},\varphi')}{(\mathcal{B}, a(t_1,\ldots,t_n); p') \xrightarrow{a(t_1,\ldots,t_n)\theta} (\mathcal{B} \uplus \xi'\theta, p'\theta)} \ (act)$$

is implemented in Maude as two rewrite rules: one for internal actions and one for observable actions. Here, we present only the rule for observable actions.

```
ops eqSC : -> EquationSet .
eq eqSC = upEqs(module-name, false) .

var OA : O-Action .

crl [exec-OA] : << L:Label, BB, pre(OA, P) >> =>
 << oLabel(getName(OA, eqSC)),
    update(BB, downTerm(substitute(upTerm(effect(OA)), S), 'err)),
    downTerm(substitute(upTerm(P), S), 'err) >>
 if S := match(prec(OA), BB) /\ S =/= noMatch .
```

Recall that equations are used to map actions to their specification in terms of preconditions and effects (expressed using the operator `o[_,_]` in case of observable actions). Before Maude applies rewrite rules to a term, it first reduces the term to its normal form using equations. This means that all actions in a plan of a mental state that is rewritten, are first replaced by their preconditions and effects. Any substitutions that are calculated while executing the plan, are therefore applied to these preconditions and effects. This implements the first two conditions of the corresponding transition rule.

In order to implement the third condition, an auxiliary operator `prec` is used, which yields the precondition of an action. The precondition is then matched against the belief base to yield a substitution. The rule can only be applied if a substitution is indeed found, i.e., if the precondition matches the belief base.

Updating the belief base according to the effect of the action is done using the operator `update : BeliefBase GroundLitSet -> BeliefBase`. The ground set of literals, which forms a parameter of this operator, is obtained from applying the calculated substitution `S` to the effect of the action using the operator `substitute : Term Substitution -> Term`. This operator is general in that it applies a substitution to any term of sort `Term`. In this case, we want to apply the substitution to the effect of an action. This can be done using the operator `upTerm` to obtain the meta-representation of the effect of the action, which is of sort `Term`,

and after applying the substitution transforming the term again into its object-level variant using `downTerm`. In a similar way, the calculated substitution is applied to the rest of the plan, according to the transition rule. The operator `getName`, which is used for obtaining the label of the new mental state, retrieves the name of the action (including instantiated parameters) from its precondition/effect specification and the action equations of the BUpL program (obtained using the meta-level built-in Maude function `upEqs`).

The transition rule for applying a plan repair rule

$$\frac{(\mathcal{B}, \alpha; p) \not\downarrow\alpha\theta' \rightarrow \qquad \varphi \leftarrow p' \in \mathcal{R} \qquad \theta \in Sols(\mathcal{B}, \varphi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \ (fail)$$

is implemented in Maude as the following rewrite rule:

```
crl [exec-fail] : << L:Label, BB, pre(A, P) >> =>
    << tLabel, BB, downTerm(substitute(upTerm(P), S), 'err) >>
    if match(prec(A), BB) == noMatch /\
       (((Q <- P)) PRB) := getPR(eqSC) /\
       S := match(Q,BB) /\ S =/= noMatch .
```

The first condition of the rewrite rule checks that the action that is to be executed, cannot be executed (which is the case if no substitution can be found when the precondition of the action is matched against the belief base). This implements the first condition of the transition rule.

The second condition of the rewrite rule implements the second condition of the transition rule as follows. Since repair rules are implemented as equations that yield a repair rule (see Section 9.3.2.2), we need an operator to collect the rules into a repair rule base. This is done by `getPR : EquationSet -> PRbase`, which takes the equations corresponding to the repair rules and yields a repair rule base consisting of the rules as defined by the equations.

The third and fourth conditions of the rewrite rule implement matching the condition of the repair rule to the belief base, corresponding to the third condition of the transition rule. The resulting substitution is applied to the plan of the repair rule, which becomes the plan of the next mental state.

9.3.5 Executing an Agent Program

The BUpL example agent from Section 9.3.3 can be executed in Maude using the command `rew builder`. Maude then uses the implemented BUpL semantics to rewrite the term `builder`, which is first reduced to the initial mental state of the builder agent using the equation `eq builder = << bLabel(empty), bb, build >>`, after which other equations and rewrite rules are applied that specify the semantics of BUpl. The Maude output looks as follows.

```
Maude> rew builder .
rewrite in AGENT-DATA : builder .
rewrites: 4722 in 202ms cpu (252ms real) (23264 rewrites/second)
result LBpMentalState:
<< oLabel('move['s_^3['0.Zero],'0.Zero,'s_^2['0.Zero]]),
clear(0) # clear(3) # on(1, 0) # on(2, 1) # on(3, 2), nilA >>
```

This says that the builder finished its execution after moving block 3 onto 2 (the current plan is empty), and that the belief base reflects the current configuration of the blocks, namely the tower 321. The output `'move[...]` is the meta-representation of `move(3, 0, 2)`. For example, `'s_^3['0.Zero]` represents the third successor of zero, i.e., 3.

One can also rewrite the builder step by step. For example, the following shows the resulting mental state after one step of rewriting, namely, a τ transition corresponding to handling the failure of action `move(2, 0, 1)` which cannot be executed since block 3 is on top of 1. We can see that the belief base remains unchanged, and the only change is in the current plan. The application of the repair rule `pr` replaces the failing plan by a plan which consists of first executing the action of moving a block (in our case block 3) onto the floor and then trying `build` again. Note that the action is represented by its precondition and effect in the form `o[precondition,effect]`.

```
Maude> rew [1] builder .
rewrite [1] in AGENT-DATA : builder .
rewrites: 4141 in 181ms cpu (228ms real) (22756 rewrites/second)
result LBpMentalState:
<< tLabel,
clear(0) # clear(2) # clear(3) # on(1, 0) # on(2, 0) # on(3, 1),
pre(o[clear(0) /\ (clear(3) /\ on(3, 1)),
neg clear(0) # neg on(3, 1) # clear(0) # clear(1) # on(3, 0)],
build) >>
```

9.4 Model-Checking

In Section 9.3, we have shown how the syntax and semantics of BUpL can be implemented in Maude, and how an example BUpL program can be defined and executed. One of the main advantages of using Maude for agent development is that it supports software development using formal methods. In this section, we show how the Maude LTL model-checker [156] can be used for verifying agent programs. Verification is important in order to ensure that the final agent program is correct with respect to a given specification or that it satisfies certain properties. Properties are specified in *linear temporal logic* (LTL) [296] and are verified using a model-checking algorithm. Model-checking *only* works for finite state systems.

We briefly recall some of the LTL concepts which we will refer to in the following sections. The basic LTL formulas are the booleans *true* ($\top$) and *false* ($\bot$)

and atomic propositions. Inductively, LTL formulas are built on top of the usual boolean connectives like negation and conjunction. Typical LTL operators are *next* ($\bigcirc$) and *until* ($\mathcal{U}$). The operator $\mathcal{U}$ can be used to define the connective *eventually*, $\Diamond\phi = \top\mathcal{U}\phi$. The connective $\Diamond$ can be used to further define the connective *always*, $\Box\phi = \neg\Diamond\neg\phi$.

The semantics of LTL formulas is defined in the usual way. The satisfaction of an LTL formula ϕ in a finite transition system S with an initial state s is defined as follows:

$$S, s \models \phi \text{ iff } (\forall \pi \in Paths(s))(S, \pi \models \phi)$$

which means that the LTL formula ϕ holds in the state s if and only if ϕ holds for any path in $Paths(s)$, the set of paths in S starting at s. Given a path π, the satisfaction relation for a formula ϕ is defined inductively on the structure of ϕ. We present, as an example, the semantics of the operator "next" and of the connective "until":

$$S, \pi \models_{LTL} \bigcirc\phi \quad \text{iff } S, \pi(1) \models_{LTL} \phi$$
$$S, \pi \models_{LTL} \phi\mathcal{U}\psi \text{ iff } (\exists n)(S, \pi(n) \models_{LTL} \psi) \wedge (\forall m < n)(S, \pi(m) \models_{LTL} \phi)$$

where n, m are natural numbers and $\pi(n)$ denotes the subpath of π starting in the "n"-th state on π. Basically, $\bigcirc\phi$ is satisfied in a state if and only if ϕ is satisfied in the successor state. The formula $\phi\mathcal{U}\psi$ holds on a path π if and only if there is a state which makes ψ true and in all the previous states ϕ was true.

Intuitively, a given path π satisfies the temporal formula $\Diamond\phi$ if there exists a state on π which satisfies ϕ. Similarly, π satisfies the temporal formula $\Box\phi$ if there does not exist a state on π which does not satisfy ϕ. By means of these operators, LTL allows specification of properties such as *safety* properties (something "bad" never happens) or *liveness* properties (something "good" eventually happens). These properties relate to the infinite behavior of a system. We will provide concrete examples in the next sections.

9.4.1 Connecting BUpL Agents and Model-Checker

Maude system modules can be seen as specifications at different levels. On the one hand they can specify *systems* (in our case, BUpL agents), on the other hand they can specify *properties* that we want to prove about a given system. The syntax of LTL is defined in the functional module `LTL` (in the file `model-checker.maude`). The following code, which is a part of the module `LTL`, shows the declaration of the temporal operators "until" (`U`), "release" (`R`), "eventually" (`<>`) and "always" (`[]`). It further shows the definitions of `<> f` (resp. `[] f`).

```
fmod LTL is
     protecting Bool .
     sort Formula .

     *** primitive LTL operators
     ops True False : -> Formula [ctor ...] .
     op _U_ : Formula Formula -> Formula [ctor ...] .
     op _R_ : Formula Formula -> Formula [ctor ...] .
     ...
     *** defined LTL operators
     op <>_ : Formula -> Formula [...] .
     op []_ : Formula -> Formula [...] .
     ...
     var f : Formula .
     eq <> f = True U f .
     eq [] f = False R f .
     ...
endfm
```

In order to use the Maude model checker, one needs to do two main things: (i) define which sort represents the states of the system that is to be model-checked, and (ii) define the atomic predicates that can be checked on these states. LTL formulas defined over these atomic predicates are then used to specify the property that is to be model-checked.

In our case, the states are the BUpL mental states of sort `LBpMentalState`. In order to express that these are the states of our system, we need the Maude model-checker module `SATISFACTION`, which is defined as follows.

```
fmod SATISFACTION is
    protecting BOOL .

        sorts State Prop .
        op _|=_ : State Prop -> Bool [frozen] .
endfm
```

We import this module into our own module `BUPL-PREDS` for defining the BUpL atomic predicates, and declare `subsort LBpMentalState < State` to express that BUpL mental states are to be considered the states of the system that is to be model-checked. Moreover, we use the operator `_|=_` for defining the semantics of the atomic state predicates, which are declared as predicates of sort `Prop`. We define the state predicate `fact(B)` to express that ground atom `B` is believed by the BUpL agent.

```
mod BUPL-PREDS is
    including BUPL-SEMANTICS .
    including SATISFACTION .
    including MODEL-CHECKER .
    including LTL-SIMPLIFIER .

    subsort LBpMentalState < State .
    op fact : Belief -> Prop .
```

```
    var B : Belief .
    eq << L:Label, B # BB:BeliefBase, P:Plan >> |= fact(B) = true .
endm
```

In the sequel, we will introduce additional state predicates to specify properties of BUpL agents.

9.4.2 Examples

To run the model-checking procedure we need, after loading in the system the file `model-checker.maude`, to call the operator `modelCheck` with an initial state and a formula, specifying the property that is to be checked, as arguments. The result of the algorithm is either the boolean **`true`** (if the property holds) or a counterexample. The operator `modelCheck` is declared in the system module `MODEL-CHECKER` which is defined in the file `model-checker.maude`.

```
fmod MODEL-CHECKER is
     including SATISFACTION .
     including LTL .
     subsort Prop < Formula .
     ...
     subsort Bool < ModelCheckResult .
     op modelCheck : State Formula ~> ModelCheckResult [...] .
endfm
```

Recall that `State` and `Formula` are sorts we have already seen declared in the modules `Statisfaction` and `LTL`, respectively (Section 9.4.1).

We can use the predicate `fact` (defined in Section 9.4.1) in order to define safety properties. As an example, we model-check that it is never the case that the agent believes the table is on block `3`. The following Maude output shows that the result is the boolean **`true`**.

```
Maude> red  modelCheck(builder, []~ fact(on(0, 3))) .
reduce in AGENT-DATA : modelCheck(builder, []~ fact(on(0, 3))) .
rewrites: 4811 in 196ms cpu (241ms real) (24425 rewrites/second)
result Bool: true
```

The predicate `fact` enables us to express properties of the beliefs of a BUpL agent. In order to express properties of actions, we define another state predicate `taken` using the label of a BUpL state. Recall that the label specifies which action has been executed.

```
mod BUPL-PREDS is
...
op taken : Action -> Prop .
ceq << oLabel(T), BB:BeliefBase, P:Plan >> |= taken(A) = true
    if T := getName(A, eqSC) .
```

The predicate `taken(A)` is true in a state if the label `T` matches `A`. Note that we cannot match `A` and `T` directly, since `T` is an action name with instantiated parameters, while `A` is an action specified by means of a precondition and effect (of the form `o[precondition,effect]`). The operator `getName` is used to obtain the name and instantiated parameters of `A` (see Section 9.3.4.2).

We can use the predicate `taken` to verify that a certain sequence of actions has been executed. For instance, the following Maude output shows that eventually, if block `2` is moved onto block `1` then moving block `3` onto block `2` takes place after this. This is an example of a liveness property.

```
Maude> red  modelCheck(builder,
        <> (taken(move(2, 0, 1)) -> O taken(move(3, 0, 2)))) .
reduce in AGENT-DATA : modelCheck(builder,
        <> (taken(move(2, 0, 1)) -> O taken(move(3, 0, 2)))) .
rewrites: 30 in 1ms cpu (0ms real) (30000 rewrites/second)
result Bool: true
```

We can define more meaningful liveness properties such as *goals* that should be reached from an initial configuration. The equation `g1` defines the predicate `goal321` as being true if the agent believes that block `3` is on block `2` and block `2` is on block `1`, expressing that the agent built the tower `321`.

```
mod AGENT-DATA-PREDS is
    including BUPL-PREDS .
    including AGENT-DATA .

    op goal321 : -> Prop .
    eq [g1] : goal321 = fact(on(3,2)) /\ fact(on(2,1)) .
endm
```

While the generic BUpL predicates `fact` and `taken` were specified in the module `BUPL-PREDS`, the predicate `goal321` is specific to the tower building agent and is consequently specified in the module `AGENT-DATA-PREDS`.

The following Maude output shows that the result of model-checking `[]<>goal321` is **`true`**, meaning that the BUpL agent will always eventually build the tower `321` from the initial configuration.

```
Maude> red  modelCheck(builder, []<> goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []<> goal321) .
rewrites: 4816 in 245ms cpu (292ms real) (19580 rewrites/second)
result Bool: true
```

We might be interested in knowing not only that `goal321` is reachable from the initial state, but also in the corresponding trace. For this, it suffices to model-check the negation of `goal321`. This returns a counterexample representing the trace that we want.

```
Maude> red  modelCheck(builder, []~ goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []~ goal321) .
rewrites: 4568 in 188ms cpu (249ms real) (24173 rewrites/second)
result ModelCheckResult: counterexample(
{<< empty-l,..., ... >>,'exec-fail}
{<< tLabel,..., ... >>,'exec-OA}
{<< oLabel('move['s_^3['0.Zero],'s_['0.Zero],'0.Zero]),
                                              ..., ... >>,'exec-OA}
{<< oLabel('move['s_^2['0.Zero],'0.Zero,'s_['0.Zero]]),
                                             ..., ... >>,'exec-OA},

{<< oLabel('move['s_^3['0.Zero],'0.Zero,'s_^2['0.Zero]]),
    clear(0) # clear(3) # on(1, 0) # on(2, 1) # on(3, 2),
                                                nilA >>, deadlock}
)
```

This counterexample should be read as follows. The declaration of the operator `counterexample` is in the predefined module `MODEL-CHECKER`. It is formed by a pair of transition lists:

```
op counterexample : TransitionList TransitionList ->
                                       ModelCheckResult [ctor] .
```

A transition list is composed of transitions, and a transition records a state and the name of the rule which has been applied from that state.

```
subsort Transition < TransitionList .
op {_,_} : State RuleName -> Transition [ctor] .
op __ : TransitionList TransitionList ->
                          TransitionList [ctor assoc id: nil] .
```

The first list of `counterexample` represents the shortest sequence of transitions (which record the states being visited) that leads to the first state of a loop. This loop is represented by the second list from `counterexample`. In our example, the first list consists of four transitions. It shows that first the rewrite rule `exec-fail` has been applied from the initial state (for readability, the belief base and plan are omitted), and consequently the label of the next state denotes a τ step. Then, the rule `exec-OA` is applied, which changes the label of the next state into the meta-representation of the action `move(3,1,0)`. A similar reasoning applies for the next transition.

The second list of the counterexample (after the white line) consists of only one transition. The initial plan has terminated (the action `nilA` is reached) and the belief base reflects that tower `321` is built. The rule name from this last transition is `deadlock`, a predefined constant which is declared in `MODEL-CHECKER`. It means that from the state that the agent reached, no further rewrite rule is applicable. Thus, the system "cycles" in a deadlock state and this is the loop represented by the second transition list. We note that a Maude deadlock state is, in our case, a termination BUpL state.

9.4.3 Fairness

The BUpL agent we have described always terminates, i.e., all execution paths are finite. Infinite behavior can occur due to recursive abstract plans, and because of the non-determinism of the operator `sum`. The reason in the latter case is that it is possible that the choice between a failing and a terminating action goes always in favor of the failing one. We call such behavior *unfair*.

In practice, unfair traces are generally prevented from occurring through scheduling algorithms such as round-robin. However, at the level of prototyping BUpL in Maude we would like to abstract from controlling the non-deterministic choices. Rather, non-determinism is reduced at a later phase of design, at a more concrete implementation level. We stress that it is important to abstract from control issues at the prototype level, since the main concern is to experiment with language definitions rather than scheduling algorithms.

Nevertheless, when model-checking BUpL agents one may want to ignore unfair traces and show that the agent satisfies certain properties assuming fairness. Since we work in a declarative framework, our solution is to model-check only the traces that satisfy certain fairness constraints and to define fairness using LTL. To illustrate this, we first introduce the predicate `enabled`. The proposition `enabled(A)` holds in a state if the action `A` can be executed in that state, i.e., if the action's precondition holds.

```
op enabled : Action -> Prop .
ceq << L:Label, BB, P >> |= enabled(A) = true
    if match(prec(A), BB) =/= noMatch .
```

Following [296], we then define fairness with respect to an action as follows.

```
op fair : Action -> Prop .
eq fair(A) = <>[] enabled(A) -> []<> taken(A) .
```

This says that if an action is continuously enabled it should be infinitely often taken. This requirement casts aside traces where the failing action is always chosen in spite of a terminating action a since such traces are unfair with respect to a.

For a concrete example where fairness is useful, we modify the BUpL example from Section 9.3.3 such that the initial plan of the agent is `p1`, which is defined as a non-deterministic choice (`sum`) between an always failing action and the plan `build`. We further add an always enabled repair rule `pr1` to handle the case where the failing action has been chosen in `p1`.

```
eq p1 = sum(i[bot, empty], build) .
ops pr1 : -> PRrule .
eq [pr1] : pr1 = (( top <- p1 )) .
...
eq builder = << bLabel(empty), bb, p1 >> .
```

It is now the case that achieving `goal321` is no longer always possible, demonstrated by the following counterexample, which is generated when model-checking the property `[]<> goal321`.

```
Maude> red  modelCheck(builder, []<> goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []<> goal321) .
rewrites: 4875 in 209ms cpu (254ms real) (23217 rewrites/second)
result ModelCheckResult: counterexample(
    {<< empty-l, ..., ... >>,'sum}
    {<< tLabel, ..., ... >>,'exec-fail},

    {<< tLabel, ..., ... >>,'sum}
    {<< tLabel, ..., ... >>,'exec-fail})
```

The counterexample shows that first the failing action was chosen to be executed, which is then handled by the repair rule `pr1`. In this counterexample, this leads to a loop in which over and over the failing action is chosen and then the repair rule is applied. This loop is represented in the second parameter of `counterexample` (below the white line).

However, if we consider the paths which are fair with respect to `move(3,1,0)` then we have that `goal321` is always achieved.

```
Maude > reduce in AGENT-DATA-PREDS :
modelCheck(builder, fair(move(3, 1, 0)) -> []<> goal321) .
rewrites: 9097 in 196ms cpu (231ms real) (46184 rewrites/second)
result Bool: true
```

9.5 Testing

In the previous section, we have illustrated how Maude can be used for model-checking BUpL agents, using the tower builder of Section 9.3.3 as an example. Since the tower builder has a finite number of mental states, verification by model-checking is in principle feasible. However, the state space of agents can also be infinite, making direct model-checking impossible. This issue may be addressed within the context of model-checking, e.g., by investigating abstractions techniques for reducing the state space. In this section, however, we are concerned with a different technique than model-checking, namely *testing*. Testing can be used for identifying failures in infinite state systems or in finite state systems where the state space becomes too large for model-checking. The basic idea behind testing is that it aims at finding failures by showing that the intended and the actual behavior of a system differ through generating and checking individual executions.

In this section, we present two kinds of testing that fit Maude very well. The first is testing for satisfaction of invariants by means of search (Section 9.5.1), and the second is testing through the specification of test cases that express properties of an execution trace of an agent (Sections 9.5.2 to 9.5.4). The latter is implemented by

means of Maude strategies, which are used to control the application of rewrite rules on a meta-level. We refer to Chapter [126] for a related approach to testing agent programs. It is similar in that it also uses a formal specification of test cases. The main differences concern the language used for specifying test cases, and we show how our approach fits into the rewriting framework of this chapter.

The running example that we use in this section is a variant of the tower builder introduced previously. Here we consider a tower builder that should respect the specification "the agent should continually construct towers, the order of the blocks is not relevant, however each tower should use more blocks than the previous, and additionally, the length of the towers must be an even number". Since the agent keeps on building higher towers, its state space is infinite. We assume that the programmer decides to refine the specification and tries to implement a BUpL agent that builds towers where the constituting blocks are assigned consecutive numbers, thus `21` and `4321` are examples of "well-formed" towers.

Initially, there is one block and it is on the table. In order to indicate that the agent has finished building a tower of length `X`, it inserts a predicate `done(X)` in the belief base by means of the action `finish(X,Y)` (where `Y` is added for technical reasons that we do not further explain). For indicating that the next tower that is to be built has length `X`, the agent uses a predicate `max(X)`. The predicate `length(X)` is used to represent the current length `X` of the tower. The builder agent is executed by rewriting a term of the form `builder(X,Y)`, where `X` is the length of the tower that is to be built as the first one, and `Y` is added for technical reasons that we do not further explain. For illustration purposes, we consider two variants of this tower builder: a correct one and a faulty one that builds odd length towers. Since it is not needed for explaining the techniques presented in this section, we do not provide the code for these tower builders.[6]

9.5.1 Searching

Maude provides a `search` command that can be used, among other things, to test for the satisfaction of invariants. Invariants are defined as properties of states. Search is breadth-first, which means that if there is a state where the invariant does not hold, then the search terminates.

Searching in Maude for invariants can be done using the Maude search command with parameters of the following form.

```
search init =>* x:k such that I(x:k) =/= true .
```

Here, `init` is the initial state from which the search starts. It searches for states `x` of sort `k` that are reachable from this initial state through zero or more rewrite steps

[6] It can be downloaded from http://homepages.cwi.nl/~astefano/agents/bupl-strategies.php.

(represented by =>*) and for which the invariant I does not hold. This is helpful when verifying safety properties. For example, an invariant for the BUpL builder is the length of the towers, which should always be even. This invariant can be specified by means of a predicate doneEven as follows.

```
mod BUPL-BUILDER-INVARIANTS is
    including AGENT-DATA .

    op doneEven : LBpMentalState -> Bool .
    ceq doneEven(<< L:Label, done(X) # BB, P:Plan >>) = true
        if (2 divides X) .
    eq doneEven(<< L:Label, done(X) # BB, P:Plan >>) = false
       [owise] .
    var MS : LBpMentalState .
endm
```

When we take the faulty implementation and search for doneEven(MS) =/= **true** with MS being a variable of sort LBpMentalState, we obtain a solution, i.e., a state where the invariant does not hold (done(3) appears in the belief base):

```
search in BUPL-BUILDER-INVARIANTS :
       builder(3, 0) =>* MS such that doneEven(MS) =/= true .

Solution 1 (state 11)
states: 12  rewrites: 21030 in 1220ms cpu (1301ms real)
(17226 rewrites/second)
MS --> << ..., clear(0) # clear(3) # length(3) # max(3) #
               done(3) # on(1, 0) # on(2, 1) # on(3, 2),
          ... >>
```

However, this procedure terminates only when the implementation is faulty, since in the correct implementation no state would be found where the invariant does not hold. A possible solution is to bound the search. This can be done by explicitly giving a depth bound, for example 100, as in the following example where the correct implementation is searched.

```
search [1, 100] in BUPL-BUILDER-INVARIANTS :
       builder(3, 0) =>* MS such that doneEven(MS) =/= true .

No solution.
states: 10  rewrites: 15266 in 779ms cpu (821ms real)
(19574 rewrites/second)
```

9.5.2 Formalizing Test Cases

Searching as treated in the previous section can be viewed as an ad hoc way of testing. While it may work for certain cases, it has several drawbacks. As for model-checking, state space explosion may be a problem since the whole state space is

considered (if no bound is used on the search). Moreover, it works with invariants expressed over the states of the system, while one may also want to test other properties such as the execution of certain sequences of actions. In this section, we present a formal language for the specification of test cases that does allow to specify this. We use so-called *rewriting strategies* [154] for implementing these tests in Maude. In Section 9.5.3, we introduce rewriting strategies and in Section 9.5.4 we show how these are used to implement a mechanism in Maude for checking whether a BUpL agent passes the tests.

Our test case format is based on one of the main BUpL concepts, namely actions. Our test case format is a kind of black box testing, aimed at testing the *observable behavior* of agents. For this reason, we have made a distinction between *internal and observable actions*. The idea is that the execution of observable actions is visible from outside the agent. Observable actions can be actions the agent executes in the environment in which it operates. In the sequel, we will sometimes omit the adjective "observable" if it is clear from the context. Black box testing as we do in this section can be contrasted with searching (Section 9.5.1), which focuses on testing properties of the belief base of agents and consequently can be viewed as a kind of white box testing.

We introduce a general test case format that allows to test whether certain sequences of observable actions can be executed. Sequences of actions are defined as regular expressions. The idea is that the action expression of a test is used to generate execution traces satisfying the action expression.[7] The action expression thus controls the execution of the agent in the sense that only those actions are executed that are in conformance with the action expression. This is crucial for reducing the state space, and makes this approach essentially different from searching.

In order to distinguish between internal and observable actions, we adapt the BUpL syntax slightly and distinguish internal and observable actions names Act_{int} and Act_{obs}, respectively, where $Act = Act_{int} \cup Act_{obs}$ and $Act_{int} \cap Act_{obs} = \emptyset$. The following BNF grammar defines the language $\mathcal{T}$ of test cases, where $a \in Act_{obs}$ denotes a ground observable action. $\mathcal{T}$ defines regular expressions over actions.

$$\mathcal{T} ::= a \mid \mathcal{T};\mathcal{T} \mid \mathcal{T} + \mathcal{T} \mid \mathcal{T}^*$$

We now define formally what it means to apply a test to a BUpL agent. For this, we adapt the operational semantics of Section 9.2.2 slightly to account for the distinction we make between internal and observable actions. In particular, instead of one transition rule for actions (*act*) we need two: one for internal actions and one for observable actions. The transition rule for internal actions is as the rule (*act*) of Section 9.2.2, except that it becomes a τ transition. This accounts for the fact that

[7] The formalism can be extended to include tests on the belief base of the agent that can be expressed using temporal logic (see [20]). These can be checked on the traces generated by testing for the execution of sequences of observable actions. However, for reasons of simplicity, we do not elaborate on this here. We refer to Chapter [126] for an approach that also uses temporal logic for expressing tests on agent behavior.

these actions are not observable. The transition rule for observable actions is as the rule (*act*) of Section 9.2.2, except that the action is an observable action.

We denote the application of a test $\mathcal{T}$ to an initial BUpL mental state ms_0 as $\mathcal{T}@ms_0$. The semantics is defined such that it yields the set of final states reachable through executing the agent restricted by the test, i.e., only those actions are executed that comply with the test. This means that an agent with initial mental state ms_0 satisfies a test $\mathcal{T}$ if $\mathcal{T}@ms \neq \emptyset$, in which case we say that a test $\mathcal{T}$ is *successful*. Since one usually tests for the absence of "bad" execution paths, we say that a BUpL agent with initial mental state ms_0 is *safe with respect to a test* $\mathcal{T}$ if the application of the test fails, i.e., $\mathcal{T}@ms_0 = \emptyset$. The operator @ (which applies a test to a single mental state) is lifted to its application to a set of mental states in the usual way, by taking the union of its application to each mental state in the set. Note that this means that $\mathcal{T}@\emptyset = \emptyset$. We define the semantics of tests as follows.

$$\mathcal{T}@ms_0 = \begin{cases} \{ms \mid ms_0 \overset{a}{\Rightarrow} ms\}, & \mathcal{T} = a \\ \mathcal{T}^1@ms_0 \cup \mathcal{T}^2@ms_0, & \mathcal{T} = \mathcal{T}^1 + \mathcal{T}^2 \\ \mathcal{T}^2@(\mathcal{T}^1@ms_0), & \mathcal{T} = \mathcal{T}^1;\mathcal{T}^2 \\ \{ms_0\} \cup \bigcup_{i\geq 1}(\mathcal{T}')^i@ms_0, & \mathcal{T} = (\mathcal{T}')^* \end{cases}$$

The arrow $\overset{a}{\Rightarrow}$ stands for $\Rightarrow\overset{a}{\rightarrow}$, where $\Rightarrow$ denotes the reflexive and transitive closure of $\overset{\tau}{\rightarrow}$.

We explain the semantics of $a@ms_0$ in some more detail. The idea is that the test should be successful for ms_0 if action a can be executed in ms_0. The result is then the set of mental states resulting from the execution of a, as defined by $\{ms \mid ms_0 \overset{a}{\Rightarrow} ms\}$. We need to keep those mental states to allow a compositional definition of the semantics. In particular, when defining the semantics of $\mathcal{T}^1;\mathcal{T}^2$ we need the mental states resulting from applying the test $\mathcal{T}^1$, since those are the mental states in which we then apply the test $\mathcal{T}^2$, as defined by $\mathcal{T}^2@(\mathcal{T}^1@ms_0)$.

9.5.3 Introduction to Maude Strategies

We choose to implement tests in Maude using rewriting strategies [154]. In this section we motivate this choice and introduce Maude rewriting strategies. Rewriting strategies are understood as a way to reduce the non-determinism of rewrite theories. Non-determinism is reduced since a strategy controls the application of rewrite rules. Strategies are related to tests as defined in the previous section, by viewing tests as a kind of strategies. Executing a BUpL agent under a test should restrict its execution such that only those actions are executed that are in conformance with the test. Take, for example, the test a. As we have previously defined it, the application of this test to a mental state ms is the set of all mental states which can be reached from ms by executing the observable action a (after possibly executing τ steps corresponding to internal actions, applying repair rules or making choices).

We are only interested in those rewritings that finally make it possible to execute a. Using strategies has the advantage of a clear separation between *execution* (by rewriting) at the object level and *control* (of rewriting) at the meta-level. In our case, we can add strategies to control the execution of the agent without making changes to the operational semantics of BUpL.

A strategy can be specified in a strategy language. Maude comes with such a strategy language, which we briefly describe now. For further details, please see [154] which introduces strategies as a language in Maude. A strategy language $\mathcal{S}$ can be viewed as a transformation of a rewrite theory $\mathcal{R}$ into $\mathcal{S}(\mathcal{R})$ such that the latter represents the execution of $\mathcal{R}$ in a controlled way. Given a term t in a rewrite theory $\mathcal{R}$ and a strategy s in the theory $\mathcal{S}(\mathcal{R})$, the application of s to t is denoted by $s@t$. The semantics of $s@t$ is the set of successors which result by rewriting t in $\mathcal{S}(\mathcal{R})$.

The simplest strategies are the constants `idle` and `fail`: `idle` @ $t = \{t\}$, `fail` @ $t = \emptyset$. Basic strategies consist of applying to a term t a rule (identified by a label) possibly with instantiating some variables appearing in the rule. The semantics of $l@t$, where l is a rule label, is the set of all terms to which t rewrites in one step using the rule labeled l anywhere it matches and satisfies the rule's condition.

Strategies can be combined under typical regular expression constructions: concatenation (`;`), union (`|`), and iteration of zero or more, or one or more steps (* or +). If E, E' are strategies, then $(E\,;\,E')@t = E'@(E@t)$, $(E \mid E')@t = (E@t) \cup (E'@t)$, $E+\ @t = \bigcup_{i \geq 1}(E^i@t)$ with $E^1 = E$ and $E^n = E^{n-1}\ ;\ E$, and $E\ * = \text{idle} \mid E+$.

It is also possible to define *if-then-else* strategies of the form E `?` E' `:` E'', which means that if the strategy E is successful when evaluated in a given state term, then the strategy E' is evaluated in the resulting states, otherwise E'' is evaluated in the *initial* state:

$$(E \texttt{ ? } E' \texttt{ : } E'')\ @\ t = \text{if } (E@t) \neq \emptyset \text{ then } E'@(E@t) \text{ else } E''@t \text{ fi}.$$

The if-then-else combinator is used to define strategies like `not`(E), which is defined as E `? fail : idle` , meaning that it reverses the result of applying E. A useful strategy is E`!`, which means "repeat until the end" and is defined as E`* ; not`(E).

In our case, state terms t are BUpL mental states. In order to rewrite `builder(3, 0)` using a strategy `E`, we only need to input the command `srew builder(3,0) using E` after loading the Maude file where the strategy language is defined (usually this is `maude-strat.maude`). If `E` is a rule name, for example, `exec-IA`, then the result is the mental state after performing an internal action, in this case setting `max(3)` which corresponds to the first parameter of `builder(3, 0)`.

```
Maude> (srew builder(3, 0) using exec-IA .)
rewrites: 1384 in 30ms cpu (55ms real) (44652 rewrites/second)
rewrite with strategy :
result LBpMentalState :
        << iLabel('set-max['s_^3['0.Zero],'0.Zero]),
        clear(0) # done(0) # length(1) # max(3) # on(1,0),
        ... >>
```

Strategies are declared and defined only in strategy modules. Strategy modules have the following syntax:

```
smod <STRAT-MODULE-NAME> is
     protecting <M> .
     including <STRAT-MODULE-NAME1> . ...
     including <STRAT-MODULE-NAMEk> .
     <DeclarationsAndDefinitionOfStrategies>
endsm
```

where `M` is the module containing the terms we want to rewrite using strategies and `STRAT-MODULE-NAME1`, ..., `STRAT-MODULE-NAMEk` are imported strategy modules.

Similarly to the declaration of operators, strategies are declared using the following format:

```
strat <STRAT-NAME> : <Sort-1> ... <Sort-m> @ <Sort> .
```

where `Sort` is the sort of the term which will be rewritten using the strategy `STRAT-NAME`. Like equations, strategies can be unconditional or conditional and are defined using the following syntax:

```
sd <STRAT-NAME>(<P1>, ..., <Pm>) := <Exp> .
csd <STRAT-NAME>(<P1>, ..., <Pm>) := <Exp> if <Cond> .
```

with `Pi` being the parameters of the strategy `STRAT-NAME` and `Exp` being a strategy expression.

9.5.4 Using Maude Strategies for Implementing Test Cases

We now illustrate how the test definitions of Section 9.5.2 can be implemented by means of Maude strategies. First, we show how the syntax of tests can be specified as a Maude functional module. We then describe a generic strategy `test2Strat` which associates to each test a corresponding strategy that implements the test. Finally, we focus on the implementation of the basic test a.

The following module defines the syntax of tests, in correspondence with the BNF grammar for tests of Section 9.5.2.

```
fmod TEST-SYNTAX is
        protecting SYNTACTICAL-DEFS .
        sort TestA .
        subsort 0-Action < TestA .
        op _;a_ : TestA TestA -> TestA .
        op _+a_ : TestA TestA -> TestA .
        op _*a : TestA -> TestA .
endfm
```

The code shows that we first declare a sort `TestA` for denoting tests. In order to express that any observable action is a test we use the subsort relation `subsort O-Action < TestA`. Further, we declare regular expression operators to construct new tests. We use the index `a` in their declaration in order to distinguish them from the regular expression operators defined for Maude strategies.

Now that we have defined the syntax of tests as above, we can define the strategy `test2Strat` inductively on the structure of tests:

```
strat test2Strat : Test @ LBpMentalState .
var Oa : O-Action . vars Ta1 Ta2 : TestA .
sd test2Strat(Oa) := do(Oa) .
sd test2Strat(Ta1 ;a Ta2) := test2Strat(Ta1) ; test2Strat(Ta2) .
sd test2Strat(Ta1 +a Ta2) := test2Strat(Ta1) | test2Strat(Ta2) .
sd test2Strat(Ta1 *a) := test2Strat(Ta1) * .
```

The strategy `do` is meant to implement the basic test a. Note the natural mapping from tests to the corresponding strategy.

We now focus on describing how to implement the basic test a, i.e., the strategy `do`. We recall that, when applied to a mental state ms, this test succeeds only if after performing some internal steps (corresponding to internal actions, repair rules, and choices among plans) the agent reaches a state where a is enabled. This means that we need to implement a strategy, `tauClosure`, for computing the transitive closure of τ steps. A simple[8] way to do this is as follows:

```
strat tauClosure : @ LBpMentalState .
sd tauClosure := (sum | exec-fail | exec-IA)! .
```

that is, by non-deterministically applying one of the rules which correspond to τ steps until no longer possible. Given that we have the strategy `tauClosure`, the implementation of the test a is straightforward:

```
strat do : O-Action @ LBpMentalState .
sd do(Oa) := tauClosure ; exec-OA[OA <- Oa] .
```

where `exec-OA[OA <- Oa]` applies `exec-OA` with the variable `OA` from the definition of the rewrite rule being instantiated by the argument `Oa` of the strategy. Note that the strategy `tauClosure` returns precisely those states from which no τ steps are possible, that is, the states where the head of the current plan is an observable action. If this observable action is the one given as argument to the strategy `do` then it succeeds and computes again the transitive closure. Otherwise, it fails. To see how this strategy works in practice, we strategically execute `builder(3, X:Nat)` using `do(move(2, 0, 1))`. This means that we test whether the agent executes `move(2, 0, 1)` as the first observable action.

[8] The strategy described here does not always terminate. One immediate solution is to bind the number of iterations. For a more detailed discussion, we refer to [20].

```
Maude> (srew builder(3,X:Nat) using do(move(2,0,1)) . )
rewrites: 18463 in 1415ms cpu (1417ms real) (13040 rewrites/second)
rewrite with strategy :
result LBpMentalState :
 << oLabel('move['s_^2['0.Zero],'0.Zero,'s_['0.Zero]]),
    clear(0)# clear(2)# clear(3)# done(0)# length(1)# max(3)#
    on(1,0)# on(2,1)# on(3,0), ...>>
Maude> (next .)
rewrites: 1210 in 10ms cpu (11ms real) (110020 rewrites/second)
next solution rewriting with strategy :
No more solutions .
```

What we obtain is a state reflecting that the agent moved block 2 onto block 1. This can be seen either from the label of the resulting mental state, or from the fact that `on(2,1)` is in the current belief base. Furthermore, we can also notice that this is the only possible resulting mental state since the command `(next .)` for obtaining other solutions returns `No more solutions`.

We recall that our purpose is to test whether "bad" states are reachable from the initial configuration of `builder` and that "bad" means odd length towers in our case. Thus, a suitable test is $move(2,0,1);move(3,0,2);finish(3,0)$, meaning that we test whether the agent (in its faulty variant) executes the action `finish(3,0)` after moving block 2 onto 1 and block 3 onto 2:

```
Maude> (srew builder(3,X:Nat) using
        test2Strat(move(2,0,1) ;a move(3,0,2) ;a finish(3, 0)) .)
rewrites: 50421 in 2069ms cpu (2082ms real) (24361 rewrites/second)
rewrite with strategy :
result LBpMentalState :
 << oLabel('finish['s_^3['0.Zero],'0.Zero]),
    clear(0)# clear(3)# done(3)# length(3)# max(3)#
    on(1,0)# on(2,1)# on(3,2), ...>>
```

The output shows that this is indeed the case, meaning that the agent is not safe to this test. Performing the same test on the correct builder yields no possible rewriting, and from this we can conclude that the correct builder agent is safe with respect to the test.

9.6 Conclusion

In this chapter, we have shown how the Maude term rewriting language can be used for agent development with formal foundations. We have shown how agent programming languages can be prototyped, and how agent programs can be executed, model-checked and tested using Maude and its accompanying tools. We maintain that one of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. This means that the implementation of the semantics of an agent programming language in Maude can

be used for executing agent programs, as well as for model-checking and testing them.

We see several main areas for future research. First, model-checking as described in this chapter applies the model-checker that comes with Maude. This means that it does not include state space abstraction techniques that are specific to agent programming languages. We see the investigation of such techniques and how they can be used in Maude as an important area for future research. Moreover, with respect to testing, the definition of the language to express test cases needs to be further investigated and experimented with to identify exactly which features are useful in practice. Another aspect related to the use of our testing framework in practice is the issue of how to come up with suitable test cases. It will need to be investigated, for example, whether it would be possible to do automatic test case generation.

Acknowledgements

We would like to thank Mehdi Dastani and John-Jules Ch. Meyer for their contributions to work on which this chapter is partly based.

Chapter 10

The Cognitive Agents Specification Language and Verification Environment

S. Shapiro, Y. Lespérance, and H.J. Levesque

Abstract The Cognitive Agents Specification Language (CASL) is a framework for specifying multiagent systems. It has a mix of declarative and procedural components to facilitate the specification and verification of *complex* multiagent systems. In this chapter, we describe CASL and a verification environment (CASLve) for it based on the PVS verification system. We give an example of a multiagent meeting scheduler application specified with CASL. To illustrate the verification system, we discuss a proof we carried out in it, namely, that all bounded-loop CASL specifications terminate.

S. Shapiro
Department of Computer Science, University of Toronto, Canada e-mail: steven@cs.toronto.edu

Y. Lespérance
Department of Computer Science and Engineering, York University, Canada e-mail: lesperan@cse.yorku.ca

H.J. Levesque
Department of Computer Science, University of Toronto, Canada e-mail: hector@ai.toronto.edu

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_10,

10.1 Introduction

The Cognitive Agents Specification Language (CASL) is a framework for specifying multiagent systems, which allows the specifier to view agents as entities with mental states, such as knowledge, beliefs, and goals, and to define the behaviour of the agents in terms of their mental states. It combines a declarative action theory defined in the situation calculus [365, 383, 391] — which allows the specifier to methodically and concisely describe the effects of actions on the world and the mental states of agents — with a rich programming/process language with constructs for concurrency and non-determinism to facilitate the specification and verification of *complex* multiagent systems.

We are also developing a verification environment (CASLve) for CASL based on the Prototype Verification System (PVS) [331] to make it easier to verify properties of CASL specifications. CASLve uses a representation of the CASL formalism within PVS. The verification environment should provide the user with a comprehensive library of proof methods, or lemmas, to facilitate the proof of various types of results (safety, liveness, termination, etc.). We are in the process of building such a library. For establishing termination, one important such lemma is that bounded-loop programs or subprograms terminate. We have proven such a lemma and discuss it in detail in Sec. 10.8, as an example of what is involved in the process of verification and building a library for this purpose. We show part of the proof in Sec. 10.9 to illustrate how CASLve is used. The environment should also provide specialized proof strategies for reasoning about CASL multiagent system specifications (e.g., regression) and customized tools for displaying CASL specifications and proofs, and these are planned for future work.

In Sec. 10.2, we summarize the PVS verification system, which forms the basis of CASLve. Then, we give a presentation of CASL spanning several sections. In Sec. 10.3, we discuss situation calculus action theories [365] and how we represent them in PVS. In Sections 10.4 and 10.5, we introduce our formalizations of knowledge and goals (resp.) and their representations in PVS. Next, in Sec. 10.6, we discuss the specification of the behaviour of the agents using ConGolog and its encoding in PVS. We present an example of a CASL specification in Sec. 10.7: a meeting scheduler example. In Sec. 10.8, we introduce CASLve and a useful lemma for verifying CASL specifications: that all bounded loop programs terminate. In Sec. 10.9, we illustrate some of the steps of a CASLve proof to give an idea of the proof process in CASLve. Finally, we conclude and discuss related work in Sec. 10.10.

10.2 PVS

PVS [331] is a typed, higher-order logic together with a proof system to facilitate theorem proving. The language has useful features, such as abstract datatypes and

recursive definitions of functions and relations. PVS also has an extensive library of theories of mathematics, and datatypes such as lists, infinite sequences, arrays, records, etc. The proof system has built-in proof strategies, including ones for inductive proofs, and facilities for adding new strategies. PVS also features a convenient Emacs-based user-interface, a facility for displaying proof trees graphically, and proof-management functionality.

A PVS specification is a collection of *theories* that are similar to logical theories except that they contain extra syntax for such purposes as declaring new types and declaring types of variables and constants. Theories can be parameterized, yielding a limited form of polymorphism. We will, as much as possible, omit the extra-logical syntax of the PVS language to make the theories look more like classical higher-order logic.

The PVS proof system is a standard sequent calculus for higher-order logic with high-level proof strategies and decision procedures to facilitate equational and mathematical reasoning. As mentioned above, PVS comes with an extensive library of theories, some of which are built in to the standard proof strategies and decision procedures. We will use $\Gamma \vdash_{\text{pvs}} \alpha$ to denote that the sentence α can be derived from the theory Γ and the built-in library of theories using the PVS proof system. $\Gamma \models \alpha$ will be used as usual to denote that Γ semantically entails α.

10.3 Action Theory

The situation calculus is a sorted predicate-calculus language for representing dynamically changing domains. The language has sorts for actions and situations. A situation represents a snapshot of the domain. There is a set of initial situations corresponding to the ways an agent thinks the domain might be initially. The actual initial state of the domain is represented by the distinguished initial situation constant, S_0. The term $do(a, s)$ denotes the unique situation that results from an agent performing action a in situation s. Thus, the situations can be structured into a set of trees, where the root of each tree is an initial situation and the arcs are actions. $Poss(a, s)$ denotes that it is physically possible to execute action a in situation s.

Predicates and functions that have a situation argument (which by convention is placed last) are called *fluents*. Fluents are used to talk about the dynamic aspects of the domain. For example, $\text{IN}(agt, r, s)$ could be used to specify that agt is in room r in situation s. The effects of actions on fluents are defined using successor state axioms [365], which provide a succinct representation for both effect axioms and frame axioms [308].

For example, assume that there are only two rooms, R_1 and R_2, and that the action LEAVE takes the agent from the current room to the other room. Then, the successor state axiom for INROOM_1 is:[1]

[1] We adopt the convention that unbound variables are universally quantified in the widest scope.

$$\begin{aligned}&\textsc{InRoom}_1(do(a,s)) \equiv\\ &\quad((\neg\textsc{InRoom}_1(s) \wedge a = \textsc{leave}) \vee (\textsc{InRoom}_1(s) \wedge a \neq \textsc{leave})).\end{aligned}$$

This axiom asserts that the agent will be in R_1 after doing some action iff either the agent is in R_2 ($\neg\textsc{InRoom}_1(s)$) and leaves it or the agent is currently in R_1 and the action is anything other than leaving it.

To completely specify the dynamics of an application domain, we use a theory with the following kinds of axioms: (1) successor state axioms for the fluents, which describe how they are affected by actions (2) action precondition axioms, which specify the circumstances under which an action can be performed, (3) initial state axioms, which describe the initial state of the domain and the initial mental state of the agent, (4) unique names axioms for the actions, and (5) domain-independent foundational axioms (discussed below).

The axioms which define the structure of the situations (including an induction axiom) are called the *foundational axioms*. Reiter [365] formulated foundational axioms for the case where there is only a single initial situation, S_0. Since we will need multiple initial situations to model knowledge and goals, we use the axiomatization provided by Lakemeyer and Levesque (L&L) [276].

L&L first define the initial situations to be those that have no predecessors: $Init_{LL}(s') \stackrel{\text{def}}{=} \neg\exists a, s.s' = do(a,s)$. Then, they define a relation on situations $s \preceq_{LL} s'$ that holds if s' can be reached from s by a (possibly empty) sequence of actions. $\preceq_{LL}$ is defined to be the smallest relation that is reflexive and transitive and contains $(s, do(a,s))$ for any s and a:

$$\begin{aligned}s \preceq_{LL} s' \stackrel{\text{def}}{=} \forall P[&(\forall s_1.P(s_1,s_1)) \wedge (\forall a, s_1.P(s_1,do(a,s_1))) \wedge\\ &(\forall s_1,s_2,s_3.P(s_1,s_2) \wedge P(s_2,s_3) \supset P(s_1,s_3)) \supset\\ &P(s,s')]\end{aligned}$$

The foundational axioms are as follows:

F1. $Init_{LL}(S_0)$.

F2. $\forall a_1,a_2,s_1,s_2.do(a_1,s_1) = do(a_2,s_2) \supset (a_1 = a_2 \wedge s_1 = s_2)$.

F3. $\forall P(\forall s,s'.Init_{LL}(s) \wedge s \preceq_{LL} s' \supset P(s')) \supset \forall s.P(s)$.

F1 declares that S_0 is an initial situation. F2 states that performing different actions yields different situations, i.e., the *do* function is 1-1. F3 is an induction axiom which says that if a property holds for any situation that can be reached from an initial situation, then that property holds for all situations.

We represent situations in PVS as an abstract datatype:

```
Sit : DATATYPE
  BEGIN
    addinit(getroot : Rootset) : Init
    do(lastact : Action, undo : Sit) : Noninit
  ENDSit
```

This datatype is called *Sit*. It uses two types, *Rootset* and *Action*, so we assume that these have been previously declared as types. The *Rootset* type is a set of objects that will become the initial situations in the situation datatype. The *Action* type is the set of actions. More information about the objects in these types will be given by applications that use this datatype, but we do not need any more details about the types for the declaration.

Datatypes have three main elements: constructors, accessors, and recognizers. Constructors form the elements of datatypes, i.e., they are functions whose values are objects in the datatype. Accessors map elements of the datatype back into the objects that were used to construct them. Recognizers are predicates that identify which constructor was used to construct an element of the datatype. The *Sit* datatype has two constructors. The first one, *addinit*, maps objects of the type *Rootset* into initial situations. The second one, *do*, maps an action and a situation into a non-initial situation. There are also two recognizers. $Init(s)$ ($Noninit(s)$, resp.) will be true iff s is an initial (non-initial, resp.) situation. *getroot* is an accessor that maps an initial situation into the element of *Rootset* that was used to construct it. *lastact* (*undo*, resp.) is an accessor that maps a non-initial situation $do(a, s)$ into a (s, resp.). Note that datatype declarations can be recursive, i.e., one can use the datatype as a type in its own declaration.

The datatype declaration generates a theory that formalizes the datatype. We can infer the last two foundational axioms from the theory generated by the datatype declaration for *Sit*. The first foundational axiom has to be explicitly added to our theory. The axioms generated for the *Init* recognizer imply that for any s, $Init(s)$ holds iff $Init_{LL}(s)$ holds. Also, among the definitions generated by the datatype declaration is a relation, $Subterm(s, s')$, which holds if s is a subterm of s' (i.e., s' can be reached from s with a finite number of applications of *do*). We will substitute the infix operator $\preceq$ for *Subterm* to make it better fit its intuitive meaning for situations. $\preceq$ is equivalent to $\preceq_{LL}$.

Let Sit denote the theory that is generated by the *Sit* datatype augmented with the axiom: $Init(S_0)$. We can show that it correctly represents the theory obtained from L&L's foundational axioms:

Theorem 10.1. *1.* $\text{Sit} \vdash_{\text{pvs}} \forall s.Init(s) \equiv Init_{LL}(s)$
2. $\text{Sit} \vdash_{\text{pvs}} \forall s, s'.s \preceq s' \equiv s \preceq_{LL} s'$
3. $\text{Sit} \vdash_{\text{pvs}} \forall a_1, a_2, s_1, s_2.do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2$
4. $\text{Sit} \vdash_{\text{pvs}} \forall P(\forall s, s'.Init(s) \wedge s \preceq s' \supset P(s')) \supset \forall s P(s)$

We will need to quantify over formulae, and so we will encode formulae as terms in the language. This is needed both for the semantics of ConGolog (for tests) and for communicative actions (INFORM and REQUEST) which take formulae as arguments. Since PVS is a higher-order logic, Shapiro [391] represented formulae as predicates on situations (or predicates on pairs of situations for goal formulae, see Sec. 10.5). This would facilitate verification since substitution would be done automatically in PVS (via function application) and would not have to be axiomatized.

Unfortunately, the unique names axioms for communicative actions that take a predicate on situations as an argument are inconsistent with the foundational axioms, in particular, with axiom F2 [391]. F2 says that there is an injection from actions into situations, i.e., the cardinality of the situations is at least as large as the cardinality of the actions. Now, suppose we have an INFORM(ϕ) action, where ϕ is a predicate on situations. The unique names axiom for INFORM would be:

$$\text{INFORM}(\phi) = \text{INFORM}(\phi') \supset \phi = \phi'.$$

Since ϕ is a predicate on situations, this axiom says that there is an injection from predicates on situations into actions, i.e., the cardinality of the actions is strictly greater than the cardinality of the situations, yielding a contradiction. Note that we would not want to drop the unique names axiom for INFORM because then it would not be clear that the agents know what they are talking about.

To avoid this problem we assume that an axiomatization of formulae as terms is given, and that we have a *Holds* predicate, which is used to interpret formulae. This axiomatization can be a general one, along the lines of the one given by De Giacomo *et al.* in [135], where, e.g., variables are represented as terms and substitution is axiomatized. However, this can be difficult to work with when doing formal proofs. If, for a particular domain, the set of formulae that will be used is known in advance, then the axiomatizer of the domain can write the *Holds* axioms directly for those sentences only, and using them in proofs should be easier. Either way, we call the given axiomatization the *encoding axioms*, assume they are consistent, and cover all the formulae that will arise in the domain of interest.

We will have two types of formulae. A *fluent formula*, denoted by ϕ (possibly with decorations) can contain a distinguished situation constant, *Now*. We assume that the axiomatization of formulae as terms ensures that $Holds(\phi, s)$ is true iff ϕ is true when *Now* is replaced by s. A *goal formula*, denoted by ψ (possibly with decorations), can contain the distinguished situation constants *Now* and *Then*. We assume that the axiomatization of formulae as terms ensures that $Holds(\psi, s, s')$ is true iff ψ is true when *Now* is replaced by s and *Then* is replaced by s'. We often suppress *Now* and *Then* when the intent is clear from the context. To simplify the formulae in the following, we abbreviate $Holds(\phi, s)$ ($Holds(\psi, s, s')$, resp.) by $\phi[s]$ ($\phi[s, s']$, resp.).

10.4 Knowledge

In CASL, we want to be able to model agents in terms of their mental states. Specifically, we include operators to specify agents' information (what they know or believe) and their motivation (what their goals are). Scherl and Levesque [383] showed how to model (single-agent) knowledge and sensing actions in the situation calculus, using a possible-worlds semantics with an accessibility relation K,

where the possible worlds are situations. We adapt their approach to handle multiple agents and communicative actions between agents. $K(agt, s', s)$ will be used to denote that in situation s, agt thinks that situation s' might be the actual situation. An agent knows a formula ϕ in s if ϕ holds in all situations K-accessible from s, i.e., $\mathbf{Know}(agt, \phi, s) \stackrel{\text{def}}{=} \forall s'.K(agt, s', s) \supset \phi[s']$.

Scherl and Levesque [383] show how to obtain a successor state axiom for K that completely specifies how knowledge is affected by actions. In their framework, the knowledge-producing actions were performed by the agent itself, i.e., sensing actions. In CASL, we are interested in communication actions, which are actions that affect the mental state of the agent to whom they are addressed rather than that of the agent who performs the action. However, Scherl and Levesque's successor state axiom for K is easily adapted to handle communicative actions and multiple agents.[2]

The speech act relevant to knowledge is the INFORM($informer, agt, \phi$) action, i.e., *informer* informs *agt* that ϕ holds. Cohen and Levesque [111] argue that for an agent to inform that ϕ holds, it must be the case that the agent knows that ϕ holds. We adopt this requirement using the following precondition axiom:

Axiom 1 *[Precondition Axiom for* INFORM*]*

$$Poss(\text{INFORM}(informer, agt, \phi), s) \equiv \mathbf{Know}(informer, \phi, s).$$

We next introduce some notation to use in the successor state axiom for K below. $K^{-}(a, agt, s', s)$ states the conditions under which action a causes (the successor of) s' to be dropped from the K-relation for agent agt. In our case, it holds if a is an action to inform agt that ϕ holds, for some ϕ, and ϕ is false in s'.

Definition 10.1.

$$K^{-}(a, agt, s', s) \stackrel{\text{def}}{=} \exists informer, \phi.a = \text{INFORM}(informer, agt, \phi) \wedge \neg\phi[s'].$$

Here is the successor state axiom for K with INFORM as the only knowledge-producing action:[3]

Axiom 2 *[Successor State Axiom for K]*

$$K(agt, s'', do(a, s)) \equiv$$
$$\exists s'.K(agt, s', s) \wedge s'' = do(a, s') \wedge Poss(a, s') \wedge \neg K^{-}(a, agt, s', s)$$

This axiom states that a situation s'' is accessible from situation $do(a, s)$ iff s'' is the result of doing action a in situation s', doing a in s' is physically possible, s' is accessible from s, and $K^{-}(a, agt, s', s)$ does not hold.

We also adapt Scherl and Levesque's definition for an agent to know whether ϕ holds in s:

[2] Another approach to generalizing to the multiagent case can be found in Lespérance *et al.* [281].

[3] This axiom can easily be extended to handle sensing actions as well as inform actions.

Definition 10.2.

$$\mathbf{KWhether}(agt, \phi, s) \stackrel{\text{def}}{=} \mathbf{Know}(agt, \phi, s) \vee \mathbf{Know}(agt, \neg\phi, s).$$

That is, an agent knows whether ϕ holds in s if it either knows that ϕ holds in s or that $\neg\phi$ holds in s.

We can place constraints on the accessibility relation, but only in the initial situations, since accessibility in successor situations is governed by the successor state axiom for K. Following Scherl and Levesque, we assert that initial situations can only be accessible from other initial situations.

Axiom 3

$$K(agt, s', s) \wedge Init(s) \supset Init(s')$$

This is a simplifying assumption, which when combined with the successor state axiom for K, entails that the agents know the history of actions that have been executed. This means that each agent is aware of every INFORM action, i.e., there is no privacy. This issue is addressed in Shapiro and Lespérance [392], where encrypted speech acts are modelled.

We can also place constraints on K to get the familiar properties of knowledge. Scherl and Levesque asserted these constraints for initial situations and then showed that they continue hold over executable sequences of actions. We do the same here. In particular, we assume that K is initially reflexive, transitive and symmetric. Recall that the order of the situations is reversed from the usual convention in modal epistemic logic.

Axiom 4

$$\begin{aligned}
&Init(s) \supset K(agt, s, s)\\
&Init(s) \supset (K(agt, s', s) \supset K(agt, s, s')\\
&Init(s) \supset (K(agt, s', s) \wedge K(agt, s'', s') \supset K(agt, s'', s)).
\end{aligned}$$

We can show that these constraints are preserved over executable sequences of actions. We say that a situation is executable, if every action in its history was physically possible to execute:

Definition 10.3.

$$Executable(s) \stackrel{\text{def}}{=} \forall a, s'.do(a, s') \preceq s \supset Poss(a, s')$$

Let Σ consist of the foundational axioms (F1–F3), the formula encoding axioms, and axioms 1–4. The following theorem states that the constraints on K are preserved over executable sequences of actions.

Theorem 10.2.

$$\Sigma \models \forall agt, s.Executable(s) \supset K(agt, s, s),$$
$$\Sigma \models \forall agt, s, s'.Executable(s) \supset (K(agt, s', s) \supset K(agt, s, s')), \text{ and}$$
$$\Sigma \models \forall agt, s, s,', s''.Executable(s) \supset (K(agt, s', s) \wedge K(agt, s'', s') \supset K(agt, s'', s)).$$

The successor state axiom for K does not support belief revision since only situations whose predecessors were accessible before an action will be accessible after the action. In this sense, no 'new' situations are added to the K relation, so the agents' knowledge is monotonic. However, since our framework is based on a logic of action, it is straightforward to support knowledge update. When an action is performed, the agents know it (due to the successor state axiom for K), and they know the consequences of performing the action (because these hold for all situations), so their knowledge is updated automatically. We now state these properties formally. First, we need some notation that allows us to talk about the past. We use $\mathbf{Prev}(\phi, s)$ to denote that ϕ held in the situation immediately before s:

Definition 10.4.

$$\mathbf{Prev}(\phi, s) \stackrel{\text{def}}{=} \exists a, s'.s = do(a, s') \wedge \phi[s'].$$

Now, we have that if an agent knows that ϕ holds in s, then after any action a, the agent will remember that ϕ held before a.

Theorem 10.3.

$$\Sigma \models \forall a, agt, \phi, s.\mathbf{Know}(agt, \phi, s) \supset \mathbf{Know}(agt, \mathbf{Prev}(\phi), do(a, s)).$$

If an agent knows that ϕ held in the last situation and that the last action a causes ϕ' to hold if ϕ holds beforehand, then the agent also knows that ϕ' holds after executing a. If ϕ is equivalent to ϕ', then we have knowledge persistence. Otherwise, it is knowledge update.

Theorem 10.4.

$$\Sigma \models \forall a, agt, \phi, \phi', s.\mathbf{Know}(agt, \mathbf{Prev}(\phi), do(a, s)) \wedge \mathbf{Know}(agt, [Poss(a, Now) \wedge \phi[Now] \supset \phi'[do(a, Now)]], s) \supset \mathbf{Know}(agt, \phi', do(a, s))$$

As a corollary, we show the conditions under which an agent's knowledge persists (or is updated).

Corollary 10.1.

$$\Sigma \models \forall a, agt, \phi, \phi', s.\mathbf{Know}(agt, \phi, s) \wedge \mathbf{Know}(agt, [Poss(a, Now) \wedge \phi[Now] \supset \phi'[do(a, Now)]], s) \supset \mathbf{Know}(agt, \phi', do(a, s))$$

We now turn to the conditions under which ignorance persists. Since $\neg\mathbf{Know}(agt, \phi, s)$ holds if there is an accessible situation where ϕ is false, for ignorance to persist, such a witness must continue to be accessible after the action is executed.

Theorem 10.5.

$$\Sigma \models \forall a, agt, \phi, s.(\exists s'.K(agt, s', s) \wedge Poss(a, s') \wedge \neg K^{-}(a, agt, s', s) \wedge \neg\phi[s']) \supset \neg\mathbf{Know}(agt, \mathbf{Prev}(\phi), do(a, s)).$$

From the last theorem, it follows that we have that if a is not an INFORM action and the agent knows that a is executable and the agent does not know ϕ, then after a occurs, the agent knows it was ignorant about ϕ beforehand.

Theorem 10.6.

$$\Sigma \models \forall a, agt, \phi, s.(\neg\exists informer, \phi_1.a = \text{INFORM}(informer, agt, \phi_1)) \wedge \mathbf{Know}(agt, Poss(a), s) \wedge \neg\mathbf{Know}(agt, \phi, s) \supset \neg\mathbf{Know}(agt, \mathbf{Prev}(\phi), do(a, s)).$$

If, after executing a, the agent does not know that ϕ held in the previous situation, and the agent also knows that a causes ϕ' to be false if ϕ is false originally, then the agent also does not know ϕ' after a is executed. Note that this theorem is *not* inconsistent with Theorem 10.4.

Theorem 10.7.

$$\Sigma \models \forall a, agt, \phi, \phi', s.\neg\mathbf{Know}(agt, \mathbf{Prev}(\phi), do(a, s)) \wedge \mathbf{Know}(agt, [Poss(a, Now) \wedge \neg\phi[Now] \supset \neg\phi'[do(a, Now)]], s) \supset \neg\mathbf{Know}(agt, \phi', do(a, s)).$$

As a corollary to the last two theorems, we have the persistence or update of ignorance.

Theorem 10.8.

$$\Sigma \models \forall a, agt, \phi, \phi', s.(\neg\exists informer, \phi.a = \text{INFORM}(informer, agt, \phi)) \wedge \mathbf{Know}(agt, Poss(a), s) \wedge \neg\mathbf{Know}(agt, \phi, s) \wedge \mathbf{Know}(agt, [Poss(a, Now) \wedge \neg\phi[Now] \supset \neg\phi'[do(a, Now)]], s) \supset \neg\mathbf{Know}(agt, \phi', do(a, s)).$$

The formalization we have given handles knowledge expansion, which is a special case of belief change where an agent adds to its existing knowledge but never discovers that it was mistaken about something it believed. In [397], we show how to generalize Scherl & Levesque's framework to handle *belief* change more generally (for the single agent case with sensing actions), where the agent can discover that it was mistaken about its beliefs and be forced to revise them.

10.5 Goals

Following Cohen and Levesque [110], we formalize the goals of an agent using an accessibility relation, $W(agt, s', s)$, which holds if in situation s, agt considers that in s' everything that it wants to be true is actually true [392]. For example, if agt wants to become a millionaire in s, then in all situations W-related to s, agt is a millionaire, but these situations can be arbitrarily far in the future. Goals will be evaluated relative to two situations *now* and *then*,[4] where $now \preceq then$. We can think of *then* as defining a path of situations, namely, the sequence of situations in the history of *then*. Intuitively, *now* is the situation along that path that occurs at the current time, i.e., the situations that come before *now* are considered to be in the past, and the situations that come after *now* are considered to be in the future. Thus, goal formulae are evaluated relative to a path of situations and the current 'time'. For example, we could represent the goal of increasing one's wealth as: TOTALASSETS(*then*) > TOTALASSETS(*now*).

We use the K accessibility relation to pick out the current situation along a path, since the K-accessible situations are the ones that the agent thinks might be the current situation. Following Cohen and Levesque, we want the goals of the agent to be compatible with what it knows. The situations that the agent wants to actualize should be on a path from a situation that the agent considers possible. Therefore, the situations that will be used to determine the goals of an agent will be the W-accessible situations that are also compatible with what the agent knows, in the sense that there is K-accessible situation in their history.

We will say that s' $K_{agt,s}$-*intersects* s'' if $K(agt, s'', s)$ and $s'' \preceq s'$. We will suppress agt or s if they are understood from the context. We define the goals of agt in s to be those formulae that are true in all the situations s' that are W-accessible from s and that K-intersect some situation, s'':

$$\begin{aligned}&\mathbf{Goal}(agt, \psi, s) \stackrel{\text{def}}{=} \\ &\quad \forall now, then. W(agt, then, s) \wedge K(agt, now, s) \wedge \\ &\qquad\qquad now \preceq then \supset \psi[now, then].\end{aligned}$$

In [391], a successor-state axiom for W was formulated that handles goal expansion as a result of REQUEST actions and goal contraction as a result of CANCELREQUEST actions. This axiom, which we adopt here, has the same structure as a successor state axiom for a domain-dependent fluent. $W^+(agt, a, s', s)$ ($W^-(agt, a, s', s)$, resp.), which is defined below, denotes the conditions under which s' is added to (dropped from, resp.) W due to action a:

Axiom 5

$$W(agt, s', do(a, s)) \equiv (W^+(agt, a, s', s) \vee (W(agt, s', s) \wedge \neg W^-(agt, a, s', s))).$$

[4] Note that *now* and *then* are situation variables, but *Now* and *Then* are situation constants.

An agent's goals are expanded when it is requested to do something by another agent. After the REQUEST($requester, agt, \psi$) action occurs, agt should adopt the goal that ψ, unless it currently has a conflicting goal. Therefore, the REQUEST($requester, agt, \psi$) action should cause agt to drop any paths in W where ψ does not hold. This action is taken into account in the definition of W^-:

Definition 10.5.

$$W^-(agt, a, s', s) \stackrel{\text{def}}{=}$$
$$\exists requester, \psi, s''.a = \text{REQUEST}(requester, agt, \psi) \wedge \neg \mathbf{Goal}(agt, \neg\psi, s) \wedge$$
$$K(agt, s'', s) \wedge s'' \preceq s' \wedge \neg\psi[do(a, s''), s'].$$

According to this definition, s' will be dropped from W, if for some *requester* and ψ, a is the REQUEST($requester, agt, \psi$) action and agt does not have the goal that $\neg\psi$ in s, and s' K-intersects some s'' such that ψ does not hold in the path $(do(a, s''), s')$. The reason that we check whether $\neg\psi$ holds at $(do(a, s''), s')$ rather than at (s'', s') is to handle goals that are relative to the current time. If, for example, ψ states that the very next action should be to get some coffee, then we need to check whether the next action after the request is getting the coffee. If we checked $\neg\psi$ at (s'', s'), then the next action would instead be the REQUEST action.

If the agent gets a request for ψ and it already has the goal that $\neg\psi$ then it does not adopt the goal that ψ, otherwise its goal state would become inconsistent and it would want everything. This is a simple way of handling goal conflicts. A more interesting method would be to give more credence to requests from certain individuals, or requests of certain types. For example, if an agent gets a request from its owner that conflicts with a previous request from someone else, it should drop the previous request and adopt its owner's request instead. We reserve a more sophisticated handling of conflicting requests for future work.

We also handle a limited form of goal contraction. Suppose that the owner of an agent asks it to do ψ and later decides it no longer wants the agent to do ψ. The owner should be able to tell the agent to stop working on ψ. We use the action CANCELREQUEST($requester, agt, \psi$) for this purpose. This action causes agt to drop the goal that ψ. A CANCELREQUEST action can only be executed if a corresponding REQUEST action has occurred in the past:

Axiom 6

$$Poss(\text{CANCELREQUEST}(requester, agt, \psi), s) \equiv$$
$$\exists s'.do(\text{REQUEST}(requester, agt, \psi), s') \preceq s.$$

We handle CANCELREQUEST actions by determining what the W relation would have been if the corresponding REQUEST action had never happened. Suppose a CANCELREQUEST($Requester, Agt, \psi$) action occurs in situation S. We restore the W relation to the way it was before the corresponding REQUEST occurred. Then, starting just after the REQUEST, we look at all the situations $do(A^*, S^*)$ in the history of S and remove from W any situation S' that satisfies $W^-(Agt, A, S', S^*)$. We first

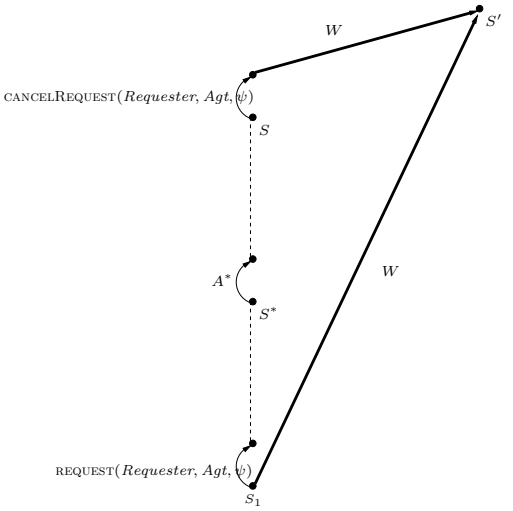

Fig. 10.1 Goal contraction.

define the predicate $\textsc{Cancels}(a, a')$, which says that action a cancels the action a'. In our case, the only cancelling action is cancelRequest, and it only cancels the corresponding request:

Definition 10.6.

$$\textsc{Cancels}(a, a') \stackrel{\text{def}}{=} \exists requester, \psi . a = \textsc{cancelRequest}(requester, agt, \psi) \wedge a' = \textsc{request}(requester, agt, \psi)$$

We now define $W^+(agt, a, s', s)$ which states the conditions under which s' is added to W^+ after a is executed in s:

Definition 10.7.

$$\begin{aligned} &W^+(agt, a, s', s) \stackrel{\text{def}}{=} \\ &\quad (\exists s_i . W(agt, s', s_i) \wedge \\ &\qquad (\exists a_1 . do(a_1, s_i) \preceq s \wedge \textsc{Cancels}(agt, a, a_1) \wedge \\ &\qquad\quad \forall a^*, s^* . do(a_1, s_i) \prec do(a^*, s^*) \preceq s \supset \neg W^-(agt, a^*, s', s^*))) \end{aligned}$$

To help explain this definition, we will refer to Fig. 10.1, where a segment of a situation forest is shown. The situation S' is not W-related to S. However, S' was

W-related to S_1, which is in the history of S. The next action after S_1 in the history of S was REQUEST($Requester, Agt, \psi$). We suppose S' was dropped from W after the REQUEST. If none of the actions between do(REQUEST($Requester, Agt, \psi$), S_1) and S also cause S' to be dropped from W, then S' is returned to W after the CANCELREQUEST($Requester, Agt, \psi$) action is executed in S. In other words, $W^+(Agt,$CANCELREQUEST$(Requester, Agt, \psi), S', S)$ holds because the following hold:

1. $W(Agt, S', S_1)$,
2. do(REQUEST($Requester, Agt, \psi$), S_1) $\preceq S$,
3. CANCELS(CANCELREQUEST($Requester, Agt, \psi$), REQUEST($Requester, Agt, \psi$)), and
4. for every situation $do(A^*, S^*)$ between do(REQUEST($Requester, Agt, \psi$), S_1) and S, $\neg W^-(Agt, A^*, S', S^*)$ holds.

Note that for our definition of W^+ to work properly, we must assume that there is only one REQUEST action in the history that is cancelled by each CANCELREQUEST, so we make that assumption here. We will relax this assumption in future work.

For the theorems in this section, we will also need unique names axioms for the INFORM, REQUEST, and CANCELREQUEST actions. We add the axioms in this section to Σ, i.e., Σ is redefined to contain: the foundational axioms (axioms F1–F3), the encoding axioms, the unique names axioms, and Axioms 1–6.

We first consider goal expansion. If an agent does not have $\neg\psi$ as a goal, then it will adopt ψ as a goal, if it receives a request for ψ.

Theorem 10.9.

$$\Sigma \models \forall agt, \psi, requester, s.\neg\mathbf{Goal}(agt, \neg\psi, s) \supset \mathbf{Goal}(agt, \psi, do(\text{REQUEST}(requester, agt, \psi), s))$$

Next, we examine the persistence of goals. We first define **Prev**($\psi, now, then$) to mean that ψ held in the last situation, i.e., the one before now:[5]

Definition 10.8.

$$\mathbf{Prev}(\psi, now, then) \stackrel{\text{def}}{=} \exists a, s''.now = do(a, s'') \wedge \psi[s'', then].$$

If an agent has the goal that ψ in S, then the only way the agent will not have **Prev**(ψ) as a goal in $do(A, S)$ is if A causes the agent to add to W a situation S' that K_S-intersects a situation S'' such that $\neg\psi[S'', S']$. Therefore, the following holds:

[5] Note that this is an overloading of the definition of **Prev** on p. 297.

Theorem 10.10.

$$\Sigma \models \forall a, agt, \psi, s.\mathbf{Goal}(agt, \psi, s) \wedge (\forall s', s''.K(agt, s'', s) \wedge s'' \preceq s' \wedge W^{+}(agt, a, s', s) \supset \psi[s'', s']) \supset \mathbf{Goal}(agt, \mathbf{Prev}(\psi), do(a, s)).$$

The following corollary says that if an agent has ψ as a goal, and a is not a CANCELREQUEST action, then after a occurs, the agent has the goal that **Prev**(ψ).

Corollary 10.2.

$$\Sigma \models \forall a, agt, \psi, s.\mathbf{Goal}(agt, \psi, s) \wedge (\forall requester, \psi'.a \neq \text{CANCELREQUEST}(requester, agt, \psi')) \supset \mathbf{Goal}(agt, \mathbf{Prev}(\psi), do(a, s)).$$

Now, if an agent has the goal that **Prev**(ψ) in $do(a, s)$, and the agent knows that the action a does not affect ψ, if it held beforehand, then the agent also has the goal that ψ in $do(a, s)$.

Theorem 10.11.

$$\Sigma \models \forall a, agt, \psi, s. \mathbf{Goal}(agt, \mathbf{Prev}(\psi), do(a, s)) \wedge \mathbf{Know}(agt, \forall s'.Poss(a, Now) \wedge \psi[Now, s'] \supset \psi[do(a, Now), s']), s) \supset \mathbf{Goal}(agt, \psi, do(a, s)).$$

We can use the last theorem to drop the **Prev** operator in Corollary 10.2. The resulting corollary says that a goal ψ persists over an action a, if a is not a CANCELREQUEST action, and the agent knows that a does not change the value of ψ.

Corollary 10.3.

$$\Sigma \models \forall a, agt, \psi, s. \mathbf{Goal}(agt, \psi, s) \wedge (\forall requester, \psi'.a \neq \text{CANCELREQUEST}(requester, agt, \psi')) \wedge \mathbf{Know}(agt, (\forall s'.Poss(a, Now) \wedge \psi[Now, s'] \supset \psi[do(a, Now), s']), s) \supset \mathbf{Goal}(agt, \psi, do(a, s)).$$

We want our agents to be able to introspect their goals, i.e., if an agent has a goal (does not have a goal, resp.) that ψ, it should know that it has (does not have, resp.) ψ as a goal. We identify constraints that yield these properties. They are constraints on K and W.

For positive introspection of goals, we need a constraint similar to transitivity. We call this constraint *KwTrans*. We define it with respect to a particular starting situation s.

Definition 10.9.

$$KwTrans(agt, s) \stackrel{\text{def}}{=} \forall s_1, s_2, s_3. K(agt, s_1, s) \wedge K(agt, s_2, s_1) \wedge W(agt, s_3, s_1) \wedge s_2 \preceq s_3 \supset W(agt, s_3, s).$$

We define what it means for K to be transitive starting in a situation s:

Definition 10.10.

$$Ktrans(agt, s) \stackrel{\text{def}}{=} \forall s_1, s_2. K(agt, s_1, s) \wedge K(agt, s_2, s_1) \supset K(agt, s_2, s).$$

If K is transitive and *KwTrans* holds starting at s then the agents have positive introspection of goals at s.

Theorem 10.12.

$$\Sigma \models \forall agt, s. Ktrans(agt, s) \wedge KwTrans(agt, s) \supset (\mathbf{Goal}(agt, \psi, s) \supset \mathbf{Know}(agt, \mathbf{Goal}(agt, \psi), s)).$$

For negative introspection of goals, we need a constraint similar to Euclideanness, which we call *KwEuc*:

Definition 10.11.

$$KwEuc(agt, s) \stackrel{\text{def}}{=} \forall s_1, s_2, s_3. K(agt, s_1, s) \wedge K(agt, s_2, s) \wedge W(agt, s_3, s) \wedge s_2 \preceq s_3 \supset W(agt, s3, s_1).$$

We define what it means for K to be Euclidean starting in a situation s:

Definition 10.12.

$$Keuc(agt, s) \stackrel{\text{def}}{=} \forall s_1, s_2. K(agt, s_1, s) \wedge K(agt, s_2, s) \supset K(agt, s_2, s_1).$$

If K is Euclidean and *KwEuc* holds starting at s, then the agents have negative introspection of goals at s.

Theorem 10.13.

$$\Sigma \models \forall agt, s. Keuc(agt, s) \wedge KwEuc(agt, s) \supset (\neg\mathbf{Goal}(agt, \psi, s) \supset \mathbf{Know}(agt, \neg\mathbf{Goal}(agt, \psi), s)).$$

We can show that *KwTrans* and *KwEuc* persist, if they hold in the initial situations, and K is initially transitive and Euclidean.

Theorem 10.14.

$$\begin{aligned}&\Sigma \models \forall agt, s.Executable(s) \wedge \\ &[\forall s'.Init(s') \supset (Ktrans(agt, s') \wedge Keuc(agt, s') \wedge \\ &\qquad KwTrans(agt, s') \wedge KwEuc(agt, s'))] \supset \\ &\quad KwTrans(agt, s) \wedge KwEuc(agt, s).\end{aligned}$$

It follows that positive and negative goal introspection persist, if the associated constraints hold initially.

10.6 Agent Behaviour

We have just presented a framework in which one can systematically and concisely describe the effects of actions on the world and on the mental states of multiple, communicating agents. In order to describe a multi-agent system, we must also specify what actions the agents perform.

We specify the behaviour of agents with the notation of the programming language ConGolog [134], the concurrent version of Golog [283]. While versions of both Golog and ConGolog have been implemented, we are mainly interested here in the potential for using ConGolog as a specification language. The language contains the following constructs:[6]

a,	primitive action
$\phi?$,	wait for a condition
$\delta_1;\delta_2$,	sequence
$\delta_1 \mid \delta_2$,	nondeterministic choice of programs
$\pi x.\delta$,	nondeterministic choice of arguments
δ^*,	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2,	conditional
for $x \in L$ **do** δ,	for loop
while ϕ **do** δ,	while loop
$\delta_1 \parallel \delta_2$,	concurrency with equal priority
$\delta_1 \rangle\rangle \delta_2$,	concurrency with δ_1 at a higher priority
$\langle\, \mathbf{x} : \phi \rightarrow \delta \,\rangle$,	interrupt

In the above, a denotes a situation calculus action; ϕ denotes a fluent formula; δ, δ_1, and δ_2 stand for complex actions; L is a finite list;[7] and $\mathbf{x}$ is a sequence of variables. These constructs are mostly self-explanatory. Intuitively, the interrupts work as follows. Whenever $\exists \mathbf{x}.\phi$ becomes true, δ is executed with the bindings of $\mathbf{x}$ that satisfied ϕ; once δ has finished executing, the interrupt can trigger again.

[6] De Giacomo *et al.* [134] allow recursive procedures in the language. To simplify matters, we omit them here. We only allow non-recursive procedures and treat them as definitions.

[7] We assume that an axiomatization of lists is included in Σ.

The semantics of ConGolog programs are defined by De Giacomo *et al.* [134] using a kind of semantics called *structural operational semantics* [212], which is based on "single steps" of computation, or *transitions*. A step here is either a primitive action or testing whether a condition holds in the current situation. They introduce two special predicates, $Final_{DG}$ and $Trans_{DG}$, where $Final_{DG}(\delta, s)$ denotes that program δ may legally terminate in situation s, and where $Trans_{DG}(\delta, s, \delta', s')$ means that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining. They then define $Do_{DG}(\delta, s, s')$ to mean that s' is a terminating situation of program δ starting in situation s:

$$Do_{DG}(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta'.Trans^*_{DG}(\delta, s, \delta', s') \wedge Final_{DG}(\delta', s'),$$

where $Trans^*_{DG}$ is the reflexive, transitive closure of $Trans_{DG}$.

In other words, $Do_{DG}(\delta, s, s')$ holds if it is possible to repeatedly single-step the program δ, obtaining a program δ' and a situation s' such that δ' can legally terminate in s'. The semantics does not handle the for-loop construct, since De Giacomo *et al.* did not have this construct. However, it is straightforward to extend their semantics to handle for-loops.

```
Program: DATATYPE
  BEGIN
    nil: Null                                                          % null program used in semantics
    prim(action: Action): Primitive                                    % primitive action
    test(testPred: Fluent): Test                                       % wait for a condition
    seqn(seqFirst:Program, seqSecond:Program): Sequence                % sequence
    nondet(ndFirst:Program, ndSecond:Program): NonDet                  % nondet. choice of actions
    pick(piPred : NP, piProg : [(piPred) → Program]): Pick             % nondet. choice of argument
    star(starProg:Program): Star                                       % nondet. iteration
    if(ifPred:Fluent, thenProg:Program, elseProg:Program): If          % conditional
    while(whilePred:Fluent, whileProg:Program): While                  % while loop
    for(forType: NP, objlist: list[(forPred)], forProg:[forPred → Program]): For   % for loop
    conc(concFirst:Program, concSecond:Program): Conc                  % concurrency with equal priority
    pri(priFirst:Program, priSecond:Program): PriConc                  % prioritized concurrency
  end Program
```

Fig. 10.2 Datatype declaration for ConGolog programs.

As seen above, De Giacomo *et al.* quantify over programs when defining Do_{DG}. To encode this semantics in PVS, we define a type for programs. We do this using the datatype declaration shown in Fig. 10.2. The declaration depends on some predefined types. As before, *Action* is the type of primitive actions. *Fluent* is the type of fluent formulae discussed earlier. $[D \rightarrow R]$ is the type of functions from D to R. *list*$[T]$ is the type of lists with elements of type T. The π, for-loop, and interrupt operators are similar to quantifiers in that they bind variables. We assume that there is a type *QuantDom* that is the quantification domain for these operators. However, we want to allow quantification over any non-empty subtype of *QuantDom*, therefore in the datatype we use the type *NP*, which we define to be the type of non-empty pred-

icates on *QuantDom*. In PVS, types cannot be passed to functions, but predicates can be. A predicate can be converted to a type (i.e., the type of objects that satisfy the predicate) by enclosing it in parenthesis. For example, the arguments to the *pick* constructor are a non-empty predicate on *QuantDom*, *piPred*, and a function from (*piPred*) to programs. This is called *dependent subtyping*, since the type of an argument of a function depends on an earlier argument. It is a very useful feature of PVS.

For each program construct in the language, we define a constructor, accessors, and a recognizer. These are mostly self-explanatory. *nil* is the null program that is used in defining *Trans* and *Final*. It is a constant constructor, so it has no accessors, but it has a recognizer *Null*. There is no constructor for interrupts because in the ConGolog semantics [134], they are defined in terms of other constructs.

As seen above, the π and for-loop operators bind variables. For example, in $\pi x.\delta$, x is introduced as a variable and δ is a program in which x can occur as a free variable. In the axioms for $Trans_{DG}$ and $Final_{DG}$, δ always appears in the scope of an existential quantifier that binds x. We saw that the *pick* operator (and the for-loop operator) in the program datatype takes a predicate and a function from objects that satisfy the predicate to programs. In other words, we are representing programs with a free variable as functions from the domain of the free variable to programs (we will refer to these functions as *program functions*). We can achieve the effect of existentially quantifying over the free variable using: $\exists y.(\lambda x.\delta)(y)$, where $(\lambda x.\delta)(y)$ denotes the application of $\lambda x.\delta$ to y and yields a program.

The axioms for $Trans_{DG}$ and $Final_{DG}$ can easily be defined in PVS using the CASES statement which handles pattern matching over datatypes. Their encodings in PVS will be denoted *Trans* and *Final*, respectively. For example, we can represent the definition of the *Final* predicate in PVS as follows:

final : AXIOM
  Final(δ^*, s) ≡
  CASES δ^* OF
    nil : *TRUE*,
    prim(a) : *FALSE*,
    test(ϕ) : *FALSE*,
    seq(δ', δ'') : Final(δ', s) ∧ Final(δ'', s),
    nondet(δ', δ'') : Final(δ', s) ∨ Final(δ'', s),
    pick($ST, \overline{\delta}$) : ∃(x : ST) : Final($\overline{\delta}(x), s$),
    star(δ) : *TRUE*,
    progif(ϕ, δ', δ'') : ($\phi(s)$ ∧ Final(δ', s)) ∨ (¬$\phi(s)$ ∧ Final(δ'', s)),
    while(ϕ, δ) : ($\phi(s)$ ∧ Final(δ, s)) ∨ ¬$\phi(s)$,
    for(*forPred*,*forList*,$\overline{\delta}$) : Null(*forList*),
    conc(δ_1, δ_2) : Final(δ_1, s) ∧ Final(δ_2, s),
    pri(δ_1, δ_2) : Final(δ_1, s) ∧ Final(δ_2, s)
  ENDCASES

The CASES statement matches its argument (represented by δ here) against each of its cases (the part to the left of the colon in each of the cases above), and returns the formula to the right of the colon for the matching case. For example, if δ^* is of the form $\text{seqn}(\delta', \delta'')$, then the CASES statement returns $\text{Final}(\delta', s) \wedge \text{Final}(\delta'', s)$. In other words, a sequence $\text{seqn}(\delta', \delta'')$ is final in situation s, if both δ' and δ'' are final in s. The overbar (e.g., $\overline{\delta}$) is used to indicate that $\overline{\delta}$ is a program function variable. The *Trans* predicate is similarly defined with a CASES operator.

trans : AXIOM
$\text{Trans}(\delta^*, s, \delta, s') \equiv$
CASES δ^* OF
$\text{nil} : \mathit{FALSE}$,
$\text{prim}(a) : \mathit{Poss}(a, s) \wedge \delta = \text{nil} \wedge s' = \mathit{do}(a, s)$,
$\text{test}(\phi) : \phi(s) \wedge \delta = \text{nil} \wedge s' = s$,
$\text{seqn}(\delta_1, \delta_2) : (\text{Final}(\delta_1, s) \wedge \text{Trans}(\delta_2, s, \delta, s')) \vee$
$(\exists \delta' : \delta = \text{seqn}(\delta', \delta_2) \wedge \text{Trans}(\delta_1, s, \delta', s'))$,
$\text{nondet}(\delta_1, \delta_2) : \text{Trans}(\delta_1, s, \delta, s') \vee \text{Trans}(\delta_2, s, \delta, s')$,
$\text{pick}(ST, \overline{\delta}) : \exists(x : ST) : \text{Trans}(\overline{\delta}(x), s, \delta, s')$,
$\text{star}(\delta_1) : \exists \delta' : \delta = \text{seqn}(\delta', \text{star}(\delta_1)) \wedge \text{Trans}(\delta_1, s, \delta', s')$,
$\text{if}(\phi, \delta_1, \delta_2) : (\phi(s) \wedge \text{Trans}(\delta_1, s, \delta, s')) \vee (\neg\phi(s) \wedge \text{Trans}(\delta_2, s, \delta, s'))$,
$\text{while}(\phi, \delta_1) : \phi(s) \wedge \exists \delta' : \delta = \text{seqn}(\delta', \text{while}(\phi, \delta_1)) \wedge \text{Trans}(\delta_1, s, \delta', s')$,
$\text{for}(\mathit{forList}, \overline{\delta}) : \neg \mathit{null}?(\mathit{forList}) \wedge \exists \delta' : \delta = \text{seqn}(\delta', \text{for}(\mathit{cdr}(\mathit{forList}), \overline{\delta}) \wedge$
$\text{Trans}(\overline{\delta}(\mathit{car}(\mathit{forList})), s, \delta', s')$,
$\text{conc}(\delta_1, \delta_2) : \exists \delta' : (\delta = \text{conc}(\delta', \delta_2) \wedge \text{Trans}(\delta_1, s, \delta', s')) \vee$
$(\delta = \text{conc}(\delta_1, \delta') \wedge \text{Trans}(\delta_2, s, \delta', s'))$,
$\text{pri}(\delta_1, \delta_2) : \exists \delta' : (\delta = \text{pri}(\delta', \delta_2) \wedge \text{Trans}(\delta_1, s, \delta', s')) \vee$
$(\delta = \text{pri}(\delta_1, \delta') \wedge \text{Trans}(\delta_2, s, \delta', s') \wedge$
$\neg \exists \delta'', s'' : \text{Trans}(\delta_1, s, \delta'', s''))$
ENDCASES

As we noted earlier, $\text{Trans}(\delta^*, s, \delta, s')$ holds if the program δ_1 can be executed one step in situation s to yield program δ in situation s'. For example, if δ^* is of the form $\text{seqn}(\delta_1, \delta_2)$, then $\text{Trans}(\delta^*, s, \delta, s')$ holds iff either δ_1 is in a final state in s and there is a transition of δ_2 in s to δ in s' or there is a transition of δ_1 in s to some δ' in s' and δ is the sequence of δ' and δ_2.

Trans^*_{DG} is the reflexive, transitive closure of Trans_{DG}. Since PVS is a higher-order logic, the definition for Trans^*_{DG}, which is second-order, could be directly encoded in PVS. However, for proofs in PVS, we find it more convenient to use a different definition of Trans^* and show it is equivalent to Trans^*_{DG}. In PVS, an infinite sequence of elements of type T can be represented by a function from *Nat* to T. We will represent a program execution by an infinite sequence of *program states*. A program state is a pair consisting of a program and a situation. We define Trans^* as follows:

$$Trans^*(\delta, s, \delta', s') \stackrel{\text{def}}{=}$$
$$\exists seq, n.seq(0) = (\delta, s) \wedge seq(n) = (\delta', s') \wedge$$
$$\forall i.i < n \supset Trans(seq(i), seq(i+1)).$$

In other words, $Trans^*(\delta, s, \delta', s')$ holds if there exists an infinite sequence, seq, and a natural number, n, such that (δ, s) is the 0-th element of seq, (δ', s') is the n-th element of seq, and for all $i < n$, $Trans$ takes the i-th element of seq to the $i+1$-th element. Note that we are using $Trans$ as a binary predicate here over pairs of program states.

This definition is equivalent to $Trans^*_{DG}$. Let ConGolog denote the theory generated by the *Program* datatype, and the axioms for *Trans* and *Final*.

Theorem 10.15.

$$\text{Sit} \cup \text{ConGolog} \vdash_{\text{pvs}} \forall \delta, s, \delta', s'.Trans^*_{DG}(\delta, s, \delta', s') \equiv Trans^*(\delta, s, \delta', s')$$

10.7 A Meeting Scheduler Example

We illustrate the use of CASL by briefly describing a specification of a meeting scheduler multi-agent system that is more fully described in [394]. In this example, there are meeting organizer agents, which are trying to schedule meetings with personal agents, which manage the schedules of their (human) owners. To schedule a meeting, an organizer agent requests of each of the personal agents of the participants in the meeting to adopt the goal that its owner attend the meeting during a given time period. If a personal agent does not have any goals that conflict with its owner attending the meeting (i.e., it has not previously scheduled a conflicting meeting), it adopts the goal that its owner attend this meeting and informs the meeting organizer that it has adopted this goal, i.e., that it accepts the meeting request. Otherwise, the personal agent informs the meeting organizer that it has not adopted the goal that its owner attend the meeting, i.e., that it declines the meeting request.

As an example of a CASL specification, the specification of the personal agents is shown in Fig. 10.3. Its arguments are the personal agent and its owner. In the specification, we use the fluent AtMeeting($user, chair, s$), which means that *user* is at a meeting chaired by *chair* in situation s. **During**($period, \phi$) means that ϕ holds throughout the time period specified by *period*. InformWhether(agt_1, agt_2, ϕ) is a complex action in which agt_1 informs agt_2 whether ϕ holds.

There are two interrupts, the first running at higher priority than the second. The first interrupt fires when the agent has the goal that the user be at a meeting that starts in less than fifteen minutes, and the agent knows that the user does not yet know that it has this goal. The agent asks the user to go the meeting (actually, the agent informs the user that it wants him to go to the meeting). The second interrupt handles meeting requests; it fires when the agent knows that an organizer agent has

$$
\begin{array}{l}
\textsc{ManageSchedule}(pa, user) \stackrel{\text{def}}{=} \\
\quad \langle period, chair : \\
\qquad \mathbf{Goal}(pa, \mathbf{During}(period, \textsc{AtMeeting}(user, chair))) \wedge \\
\qquad \mathbf{Know}[pa, \neg \mathbf{Know}(user, \mathbf{Goal}(pa, \mathbf{During}(period, \textsc{AtMeeting}(user, chair)))) \wedge \\
\qquad\qquad \textsc{Start}(period) - :15 \leq time \leq \textsc{Start}(period)] \rightarrow \\
\qquad\quad \textsc{Inform}(pa, user, \mathbf{Goal}(pa, \mathbf{During}(period, \textsc{AtMeeting}(user, chair)))) \rangle \\
\quad \rangle\!\rangle \\
\quad \langle oa, period, chair : \\
\qquad \mathbf{Know}(pa, \mathbf{Goal}(oa, \mathbf{During}(period, \textsc{AtMeeting}(user, chair))) \wedge \\
\qquad\qquad \neg \mathbf{KWhether}(oa, \mathbf{Goal}(pa, \mathbf{During}(period, \textsc{AtMeeting}(user, chair))))) \rightarrow \\
\qquad\quad \textsc{InformWhether}(pa, oa, \mathbf{Goal}(pa, \mathbf{During}(period, \textsc{AtMeeting}(user, chair)))) \rangle
\end{array}
$$

Fig. 10.3 Specification of the personal agents.

requested a meeting, and it knows that it has not yet replied to the request. The action taken is to inform the organizer whether it accepts or declines the meeting request.

A complete meeting scheduler system is defined by composing instances of the personal agents and the meeting organizer agents in parallel, thereby modelling the behaviour of several agents acting independently. We also need to compose the nondeterministic iteration of a *tick* action concurrently (at a lower priority) to allow time to pass when the agents are not acting. Here is an example of such a system:

$$
\begin{array}{l}
[\textsc{ManageSchedule}(\textsc{pa}_1, \textsc{user}_1) \parallel \\
\textsc{ManageSchedule}(\textsc{pa}_2, \textsc{user}_2) \parallel \\
\textsc{ManageSchedule}(\textsc{pa}_3, \textsc{user}_3) \parallel \\
\textsc{OrganizeMeeting}(\textsc{oa}_1, \textsc{user}_1, \{\textsc{user}_1, \textsc{user}_3\}, \\
\qquad\qquad 12{:}00\text{–}2{:}00) \parallel \\
\textsc{OrganizeMeeting}(\textsc{oa}_2, \textsc{user}_2, \{\textsc{user}_2, \textsc{user}_3\}, \\
\qquad\qquad 1{:}30\text{–}2{:}45)] \rangle\!\rangle \textsc{tick}^*
\end{array}
$$

In this example, $\textsc{oa}_1$ is trying to schedule a meeting between $\textsc{user}_1$ and $\textsc{user}_3$ from 12 to 2. $\textsc{oa}_2$ is trying to schedule a meeting between $\textsc{user}_2$ and $\textsc{user}_3$ from 1:30 to 2:45. Since both meeting organizers will try to obtain $\textsc{user}_3$'s agreement for meetings that overlap, there will be two types of execution sequences, depending on who obtains this agreement.

The meeting scheduler system is easy to represent in CASLve. The only complication is in defining the domain of quantification (*QuantDom*). In the example, we need to quantify over periods of time and agents. We represent periods of time as pairs of natural numbers, while agents are declared as a primitive type. In PVS, one cannot directly define a type to be the union of other types, so we had to define a datatype with constructors for both types. This means that when defining the system, we have to use constructors and accessors to map into the *QuantDom* and back

to the original types. See [391] for the details of the PVS encoding, and for a formal proof that a terminating execution of the meeting scheduler system exists.

10.8 Verification

We now present the PVS-based verification environment for CASL that we have developed, which we call CASLve. PVS has high-level proof-strategies and decision procedures that take care of many of the low-level details associated with computer-aided theorem proving. Some simple proofs (including some inductive proofs) can be handled using a single application of a proof strategy. Many proofs can be accomplished using only the following steps: lemma introduction (here lemma is a general term for any proposition: axioms, lemmas, theorems, etc., and includes induction axioms), definition expansion, quantifier instantiation, and simplification. In addition, PVS has useful proof-management facilities, including a graphical display of the proof tree, and proof stepping and editing.

We have used CASLve to prove many lemmas that are useful in verifying properties of programs. In the remainder of this section, we will discuss one such lemma, namely that all bounded-loop programs terminate. We define a bounded-loop program to be one without while-loops and nondeterministic iteration (but for-loops are allowed): $Bounded(\delta) \stackrel{\text{def}}{=} \forall \delta'.Subterm(\delta', \delta) \supset \neg Star(\delta') \wedge \neg While(\delta')$. If we want to prove that this property holds for some program, we run into a problem involving the use of program functions as a way of handling program operators that bind variables. We intend to use functions such as: $\lambda x.seqn(test(\text{ONTABLE}(x)), \text{REMOVE}(x))$, which when applied to an object such as BLOCK1 simulates the substitution of BLOCK1 for x. However, there is nothing to stop us from defining a function $f : \text{NAT} \rightarrow \text{PROGRAM}$ such that $f(i) = a^i$, where a^i denotes the sequential composition of a with itself i times. Then, $pick(Nat, f)$ is essentially an unbounded program because we cannot bound in advance the number of primitive actions that it will execute.

Our solution to this problem is to restrict the program functions to functions that always return programs that have the same structure. We first define a congruence relation $Congruent(\delta, \delta')$ that holds if two programs have the same structure. This relation is defined recursively and it checks that the outermost operator is the same for each program and then recursively checks that the rest of the programs are congruent. In addition, if δ and δ' are of the form $pick(NP_1, pf_1)$, and $pick(NP_2, pf_2)$, respectively, then we require that $NP_1 = NP_2$ and $\forall x : (NP_1).Congruent(pf_1(x), pf_2(x))$. We have a similar requirement for for-loops, but we also require that the two lists be of the same length. We omit the formal definition of this relation here. We say that a program δ is *suitable*, formally $SProg(\delta)$, if for any subprogram of δ that is of the form $pick(NP, pf)$ or $for(NP, l, pf)$ all the instantiations of pf are congruent, i.e., $\forall x, y : (NP).Congruent(pf(x), pf(y))$. Again, we omit the formal definition. We will

limit our attention to suitable programs. We can do this because they are closed under transitions:

Theorem 10.16.

$$\text{Sit} \cup \text{ConGolog} \vdash_{\text{pvs}} \forall \delta, \delta', s, s'. SProg(\delta) \wedge Trans(\delta, s, \delta', s') \supset SProg(\delta').$$

All future quantifications over programs will be assumed to be restricted to suitable programs.

Following Francez [184], we say that a program δ *terminates* starting in situation s if it has no infinite executions starting in s. We can use the notion of a sequence of program states introduced for our definition of $Trans^*$ in Sec. 10.6 to talk about infinite executions. We say that δ has an infinite execution starting in s, if there is a infinite sequence of program states that starts with (δ, s), such that *Trans* holds of each adjacent pair of states: $InfExec(\delta, s) \stackrel{\text{def}}{=} \exists seq. seq(0) = (\delta, s) \wedge \forall i. Trans(seq, i)$, where *seq* ranges over functions from the natural numbers to program states, and $Trans(seq, i)$ holds if there is a transition from the i-th to the $i+1$-th element of *seq*. Note that this is an overloading of the previously defined binary *Trans* predicate. Now, we can define termination as the absence of an infinite execution: $Terminates(\delta, s) \stackrel{\text{def}}{=} \neg InfExec(\delta, s)$.

We adapt a technique from Francez [184], to assist in proving termination of a program δ starting in situation s. The idea is to find a predicate $P_{seq} \subseteq \text{NAT} \times \text{NAT}$ that is intuitively a measure on an execution *seq* of δ (where the sequence is understood from the context, we drop the subscript). Intuitively, $P(i, j)$ holds if the i-th step of *seq* has measure j. We can infer that $Terminates(\delta, s)$ holds, if for any sequence *seq* such that $seq(0) = (\delta, s)$, there is a P such that:

1. $P(0, j)$ holds for some j,
2. the value of the measure strictly decreases with each transition step of the sequence, and
3. when the measure reaches 0, i.e., $P(i, 0)$ for some i, then there is no legal transition from $seq(i)$ to $seq(i+1)$,

We state this formally in the following theorem:

Theorem 10.17.

$$\begin{array}{l}\text{Sit} \cup \text{ConGolog} \vdash_{\text{pvs}} \\ \quad \forall \delta, s[\forall seq. seq(0) = (\delta, s) \supset \\ \quad\quad (\exists P.(\exists j. P(0, j)) \wedge \\ \quad\quad\quad (\forall i, j. j > 0 \wedge P(i, j) \wedge Trans(seq, i) \supset \\ \quad\quad\quad\quad \exists k. k < j \wedge P(i+1, k)) \wedge \\ \quad\quad\quad (\forall i. P(i, 0) \supset \neg Trans(seq, i)))] \supset Terminates(\delta, s)\end{array}$$

Note that we use a relation for the measure because we do not want to require that the measure be defined over all natural numbers. In particular, once the measure reaches 0, we want to allow it to be undefined from then on. If we can find such a measure for any bounded program δ and situation s, we can show that all bounded programs terminate. The measure that we use is based on the length of δ, where the length of a program is the maximum number of primitive actions and tests that will be generated in any execution of the program. We informally describe the *proglen* function which maps a bounded program to its length, but omit its formal definition. It is defined recursively. *nil* has length 0. Primitive actions and tests have length 1. The length of sequential, concurrent, and prioritized concurrent compositions are the sum of the lengths of their arguments. The length of nondeterministic choice of programs and if-then-else statements are the maximum of the lengths their program arguments. Since we are only considering bounded, suitable programs, the length of the program that results from applying the program function argument of a *pick* statement to an object will be the same for all objects. Therefore, the length of a *pick* statement is the length of the program that results from applying the program function to an arbitrary object. Similarly, the length of a for-loop is the list length of its list argument multiplied by the program length of its program function argument applied to an arbitrary object.

We will now define a relational measure, P_{seq}, that uses the program length. We want the measure to be defined initially and as long as the sequence continues to be a valid execution of the program. Where it is defined, $P_{seq}(i, j)$ will hold if j is the program length of the program component of $seq(i)$: $P_{seq} \stackrel{\text{def}}{=} \lambda i, j.(\forall k.k < i \supset Trans(seq, k)) \wedge j = proglen(pj_1(seq(i)))$, where pj_1 projects out the first element of a pair. We use this measure to show that all bounded programs terminate:

Theorem 10.18.

$$\text{Sit} \cup \text{ConGolog} \vdash_{\text{pvs}} \forall \delta, s.Bounded(\delta) \supset Terminates(\delta, s)$$

10.9 Example Proof

To illustrate CASLve, we will run through part of a proof. Since we are presenting parts of a PVS proof, we will use PVS notation, i.e., a sequent calculus with a typed, higher-order language. The proof we illustrate is a lemma that is used in the proof of Theorem 10.18, i.e., that all bounded programs terminate. The lemma says that all legal transitions of bounded programs result in programs of smaller length; it is stated formally in the first sequent below.

When the PVS prover is invoked, one enters the proof mode with a single sequent that contains only the proposition to be proved in the consequent. The antecedent formulae are numbered with negative integers, while the consequent are numbered with positive integers.

$$\{1\}\ \forall(\delta : Bounded), (\delta' : Program), (s, s' : Sit) : Trans(\delta, s, \delta', s') \supset proglen(\delta') < proglen(\delta)$$

The proof proceeds by induction over δ. The PVS command for this is: (INDUCT δ 1), which is a strategy that sets up a proof of formula {1} by induction over δ. Since δ is of type *Program*, PVS sets up the induction by creating a new sequent to prove for each program construct, and possibly some sequents to prove type correctness conditions. Since we do not have much space, we will only show the proof of one of the cases. We will show the proof for tests. Recall from Fig. 10.2 on page 306 that the program construct for tests is $test(\phi)$, where ϕ is a fluent formula. The sequent that PVS generates for this case is as follows.

$$\{1\}\ \forall(\phi : Fluent) : Bounded(test(\phi)) \supset \forall(\delta' : Program), (s, s' : Sit) : Trans(test(\phi), s, \delta', s') \supset proglen(\delta') < proglen(test(\phi))$$

Next, we simplify the sequent with the PVS command (REDUCE NIL). REDUCE is a strategy that performs various simplifications, including skolemization, propositional simplification, applying decision procedures, and equality replacement. The NIL parameter is used to prevent heuristic quantifier instantiation. Skolem constants are formed by adding subscripted numerals to variable names. The following sequent is the result of the simplification.

$$\begin{array}{l}\{\text{-}1\}\ Bounded(test(\phi_1))\\ \{\text{-}2\}\ Trans(test(\phi_1), s_1, \delta_1', s_1')\\ \hline \{1\}\ proglen(\delta_1') < proglen(test(\phi_1))\end{array}$$

In our encoding of ConGolog, we made *Trans* a PVS definition. The definition states that $Trans(test(\phi_1), s_1, \delta_1', s_1')$ holds iff $\phi_1[s_1]$ holds and $s_1 = s_1'$ and $\delta_1' = nil$. The next step of the proof is to expand the definition of *Trans*, which yields the following sequent.

$$\begin{array}{l}[\text{-}1]\ Bounded(test(\phi_1))\\ \{\text{-}2\}\ \phi_1[s_1] \wedge \delta_1' = nil \wedge s_1' = s_1\\ \hline [1]\ proglen(\delta_1') < proglen(test(\phi_1))\end{array}$$

If a formula's number is enclosed in square brackets, it means that the formula has not changed from the previous sequent. In the definition of *proglen*, the *nil* program is given length 0 and a program consisting of only a test is given length 1. Therefore, after simplifying and expanding the definition of *proglen*, we obtain the following sequent.

$$\begin{array}{l}[\text{-}1]\ Bounded(test(\phi_1))\\ [\text{-}2]\ \phi_1[s_1]\\ [\text{-}3]\ \delta_1' = nil\\ [\text{-}4]\ s_1' = s_1\\ \hline \{1\}\ 0 < 1\end{array}$$

It is easy to see that this sequent is true, and we can use the (GROUND) command, which simplifies using decision procedures, to complete the proof. We have illustrated some of the main steps used in CASLve. The other ones that are used most often are quantifier instantiation, lemma introduction, and GRIND, which is a strategy that repeatedly performs heuristic instantiation of quantifiers and simplification and can complete many simple proofs. PVS also has a facility for creating user-defined strategies, which we would like to use to develop strategies specifically for CASL to further facilitate the verification of CASL specifications.

10.10 Conclusion

We have presented the different aspects of the CASL specification language and how we encoded them in PVS to form the basis of a verification environment. We briefly described the specification of a meeting scheduler multi-agent system in CASL, and we showed that all bounded-loop programs terminate, which is a useful lemma for proving termination of CASL programs.

CASL is a very expressive language, which we believe facilitates the specification of complex multiagent systems. Of course, the expressivity of the language makes the task of verification more difficult. Other agent verification frameworks have chosen to limit the expressivity of the language in order to facilitate the verification side. Some approaches use model checking for automated verification [34, 73]. However the languages used are propositional. Other approaches use theorem proving. For example, Engelfriet *et al.* [164] consider compositional verification of multi-agent systems. Hindriks and Meyer [226] define an agent programming language and a verification logic for it. However, both these approaches use propositional logic, and neither provide a verification environment. Our approach is quite different. We believe that the complexity of multiagent systems justifies using more expressive languages, and that the full spectrum of the trade-off between language expressivity versus ease of verification should be examined.

CASLve is a work in progress. We would like to develop it further by, e.g., writing proof strategies in PVS specifically tailored to the verification of CASL specifications. Ultimately, we hope to develop a hybrid system that uses theorem proving to structure verification at a high level and generate lower level verification tasks that could be handled with automated theorem proving techniques or model checking. This could be one way to combine an expressive agent specification language with effective verification techniques.

In other future work, we plan to use CASL to verify properties of the meeting scheduler system and other multiagent specifications. In addition, we would like to extend our account of goal change by handling conflicting requests and allowing more than one REQUEST in the history of a CANCELREQUEST, as mentioned above. Finally, we will extend CASL to handle recursive procedures, using the framework of De Giacomo *et al.* [134].

Chapter 11

A Temporal Trace Language for Formal Modelling and Analysis of Agent Systems

A. Sharpanskykh and J. Treur

Abstract This chapter presents the hybrid Temporal Trace Language (TTL) for formal specification and analysis of dynamic properties of multi-agent systems. This language supports specification of both qualitative and quantitative aspects, and subsumes languages based on differential equations and temporal logics. TTL has a high expressivity and normal forms that enable automated analysis. Software environments for performing verification of TTL specifications have been developed. TTL proved its value in a number of domains.

A. Sharpanskykh
Vrije Universiteit Amsterdam, Department of Artificial Intelligence, The Netherlands e-mail: sharp@cs.vu.nl

J. Treur
Vrije Universiteit Amsterdam, Department of Artificial Intelligence, The Netherlands e-mail: treur@cs.vu.nl

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,

DOI 10.1007/978-1-4419-6984-2_11,

11.1 Introduction

Traditionally, the multi-agent paradigm has been used to improve efficiency of software computation. Languages used to specify such multi-agent systems often had limited expressive power (e.g., executable, close to (logic) programming languages), which nevertheless was sufficient to describe complex distributed algorithms. Recently many agent-based methods, techniques and methodologies have been developed to model and analyse phenomena in the real world (e.g., social, biological, and psychological structures and processes). By formally grounded multi-agent system modelling one can gain better understanding of complex real world processes, test existing theories from natural and human sciences, identify different types of problems in real systems.

Modelling dynamics of systems from the real world is not a trivial task. Currently, continuous modelling techniques based on differential and difference equations are often used in natural science to address this challenge, with limited success. In particular, for creating realistic continuous models for natural processes a great number of equations with a multitude of parameters are required. Such models are difficult to analyze, both mathematically and computationally. Further, continuous modelling approaches, such as the Dynamical Systems Theory [344], provide little help for specifying global requirements on a system being modelled and for defining high level system properties that often have a qualitative character (e.g., reasoning, coordination). Also, sometimes system components (e.g., switches, thresholds) have behaviour that is best modelled by discrete transitions. Thus, the continuous modelling techniques have limitations, which can compromise the feasibility of system modelling in different domains.

Logic-based methods have proved useful for formal qualitative modelling of processes at a high level of abstraction. For example, variants of modal temporal logic [27, 198] gained popularity in agent technology, and for modelling social phenomena. However, logic-based methods typically lack quantitative expressivity essential for modelling precise timing relations as needed in, e.g., biological and chemical processes.

Furthermore, many real world systems (e.g., a television set, a human organisation, a human brain) are hybrid in nature, i.e., are characterized by both qualitative and quantitative aspects. To represent and reason about structures and dynamics of such systems, the possibility of expressing both qualitative and quantitative aspects is required. Moreover, to tackle the issue of complexity and scalability the possibility of modelling of a system at different aggregation levels is in demand. In this case modelling languages should be able to express logical relationships between parts of a system.

To address the discussed modelling demands, the Temporal Trace Language (TTL) is proposed, which subsumes languages based on differential equations and temporal logics, and supports the specification of the system behaviour at different levels of abstraction.

Generally, the expressivity of modelling languages is limited by the possibility to perform effective and efficient analysis of models. Analysis techniques for complex systems include simulation based on system models, and verification of dynamic properties on model specifications and traces generated by simulation or obtained empirically.

For simulation it is essential to have limitations to the language. To this end, an executable language that allows specifying only direct temporal relations can be defined as a sublanguage of TTL; cf. [81]. This language allows representing the dynamics of a system by a (possible large) number of simple temporal (or causal) relations, involving both qualitative and quantitative aspects. Furthermore, using a dedicated tool, TTL formulae that describe the complex dynamics of a system specified in a certain format may be automatically translated into the executable form. Based on the operational semantics and the proof theory of the executable language, a dedicated tool has been developed that allows performing simulations of executable specifications.

To verify properties against specifications of models two types of analysis techniques are widely used: logical proof procedures and model checking [100]. By means of model checking entailment relations are justified by checking properties on the set of all theoretically possible traces generated by execution of a system model. To make such verification feasible, expressivity of both the language used for the model specification and the language used for expressing properties has to be sacrificed to a large extent. Therefore, model specification languages provided by most model checkers allow expressing only simple temporal relations in the form of transition rules with limited expressiveness (e.g., no quantifiers). For specifying a complex temporal relation a large quantity (including auxiliary) of interrelated transition rules is needed. In this chapter normal forms and a transformation procedure are introduced, which enable automatic translation of an expressive TTL specification into the executable format required for automated verification (e.g., by model checking). Furthermore, abstraction of executable specifications, as a way of generating properties of higher aggregation levels, is considered in this chapter. In particular, an approach that allows automatic generation of a behavioural specification of an agent from a cognitive process model is described.

In some situations it is required to check properties only on a limited set of traces obtained empirically or by simulation (in contrast to model checking which requires exhaustive inspection of all possible traces). Such type of analysis, which is computationally much cheaper than model checking, is described in this chapter.

The chapter is organised as follows. Section 11.2 describes the syntax of the TTL language. The semantics of the TTL language is described in Section 11.3. Multi-level modelling of multi-agent systems in TTL and a running example used throughout the chapter are described in Section 11.4. In Section 11.5 relations of TTL to other well-known formalisms are discussed. In Section 11.6 normal forms and transformation procedures for automatic translation of a TTL specification into the executable format are introduced. Furthermore, abstraction of executable specifications is considered in Section 11.6. Verification of specifications of multi-agent

systems in TTL is considered in Section 11.7. Finally, Section 11.8 concludes the chapter.

11.2 Syntax of TTL

The language TTL is a variant of an order-sorted predicate logic [299]. Whereas standard multi-sorted predicate logic is meant to represent static properties, TTL is an extension of such language with explicit facilities to represent dynamic properties of systems. To specify state properties for system components, ontologies are used which are specified by a number of sorts, sorted constants, variables, functions and predicates (i.e., a signature). State properties are specified based on such ontology using a standard multi-sorted first-order predicate language. For every system component A (e.g., agent, group of agents, environment) a number of ontologies can be distinguished used to specify state properties of different types. That is, the ontologies *IntOnt*(*A*), *InOnt*(*A*), *OutOnt*(*A*), and *ExtOnt*(*A*) are used to express respectively internal, input, output and external state properties of the component *A*. For example, a state property expressed as a predicate *pain* may belong to *IntOnt*(*A*), whereas the atom *has_temperature*(*environment*,7) may belong to *ExtOnt*(*A*). Often in agent-based modelling input ontologies contain elements for describing perceptions of an agent from the external world (e.g, *observed*(*a*) means that a component has an observation of state property *a*), whereas output ontologies describe actions and communications of agents (e.g., *performing_action*(*b*) represents action *b* performed by a component in its environment).

To express dynamic properties, TTL includes special sorts: $TIME$ (a set of linearly ordered time points), $STATE$ (a set of all state names of a system), $TRACE$ (a set of all trace names; a trace or a trajectory can be thought of as a timeline with a state for each time point), $STATPROP$ (a set of all state property names), and $VALUE$ (an ordered set of numbers). Furthermore, for every sort S from the state language the following TTL sorts exist: the sort S^{VARS}, which contains all variable names of sort S, the sort S^{GTERMS}, which contains names of all ground terms, constructed using sort S; sorts S^{GTERMS} and S^{VARS} are subsorts of sort S^{TERMS}.

In TTL, formulae of the state language are used as objects. To provide names of object language formulae φ in TTL, the operator (*) is used (written as φ^*), which maps variable sets, term sets and formula sets of the state language to the elements of sorts S^{GTERMS}, S^{TERMS}, S^{VARS} and $STATPROP$ in the following way:

1. Each constant symbol c from the state sort S is mapped to the constant name c' of sort S^{GTERMS}.
2. Each variable $x : S$ from the state language is mapped to the constant name $x' \in S^{VARS}$.
3. Each function symbol $f : S_1 \times S_2 \times \ldots \times S_n \rightarrow S_{n+1}$ from the state language is mapped to the function name $f' : S_1^{TERMS} \times S_2^{TERMS} \times \ldots \times S_n^{TERMS} \rightarrow S_{n+1}^{TERMS}$.

4. Each predicate symbol $P : S_1 \times S_2 \times ... \times S_n$ is mapped to the function name $P' : S_1^{TERMS} \times S_2^{TERMS} \times ... \times S_n^{TERMS} \rightarrow STATPROP$.
5. The mappings for state formulae are defined as follows:

   a. $(\neg\varphi)^* = not(\varphi^*)$
   b. $(\varphi \,\&\, \psi)^* = \varphi^* \wedge \psi^*, (\varphi \,|\, \psi)^* = \psi^* \vee \psi^*$
   c. $(\varphi \Rightarrow \psi)^* = \varphi^* \rightarrow \psi^*, (\varphi \Leftrightarrow \psi)^* = \varphi^* \leftrightarrow \psi^*$
   d. $(\forall x\, \varphi(x))^* = \forall x'\, \varphi^*(x')$, where x is variable over sort S and x' is any constant of S^{VARS}; the same for $\exists$.

It is assumed that the state language and the TTL define disjoint sets of expressions. Therefore, further in TTL formulae we shall use the same notations for the elements of the object language and for their names in the TTL without introducing any ambiguity. Moreover we shall use t with subscripts and superscripts for variables of the sort $TIME$; and γ with subscripts and superscripts for variables of the sort $TRACE$.

A state is described by a function symbol $state : TRACE \times TIME \rightarrow STATE$. A trace is a temporally ordered sequence of states. A time frame is assumed to be fixed, linearly ordered, for example, the natural or real numbers. Such an interpretation of a trace contrasts to Mazurkiewicz traces [306] that are frequently used for analysing behaviour of Petri nets. Mazurkiewicz traces represent restricted partial orders over algebraic structures with a trace equivalence relation. Furthermore, as opposed to some interpretations of traces in the area of software engineering, a formal logical language is used here to specify properties of traces.

The set of function symbols of TTL includes $\vee, \wedge, \rightarrow, \leftrightarrow$: $STATPROP \times STATPROP \rightarrow STATPROP$; $not : STATPROP \rightarrow STATPROP$, and $\forall, \exists : S^{VARS} \times STATPROP \rightarrow STATPROP$, of which the counterparts in the state language are boolean propositional connectives and quantifiers. Further we shall use $\vee, \wedge, \rightarrow, \leftrightarrow$ in infix notation and $\forall, \exists$ in prefix notation for better readability. For example, using such function symbols the state property about external world expressing that there is no rain and no clouds can be specified as: $not(rain) \wedge not(clouds)$.

To formalise relations between sorts $VALUE$ and $TIME$, functional symbols $-, +, /, \bullet : TIME \times VALUE \rightarrow TIME$ are introduced. Furthermore, for arithmetical operations on the sort $VALUE$ the corresponding arithmetical functions are included.

States are related to state properties via the satisfaction relation denoted by the prefix predicate $holds$ (or by the infix predicate $\models$): $holds(state(\gamma, t), p)$ (or $state(\gamma, t) \models p$), which denotes that state property p holds in trace γ at time point t.

Both $state(\gamma, t)$ and p are terms of the TTL language. In general, TTL terms are constructed by induction in a standard way from variables, constants and function symbols typed with all before-mentioned TTL sorts. Transition relations between states are described by dynamic properties, which are expressed by TTL-formulae. The set of *atomic TTL-formulae* is defined as:

1. If v_1 is a term of sort $STATE$, and u_1 is a term of the sort $STATPROP$, then $holds(v_1, u_1)$ is an atomic TTL formula.
2. If τ_1, τ_2 are terms of any TTL sort, then $\tau_1 = \tau_2$ is a TTL-atom.
3. If t_1, t_2 are terms of sort $TIME$, then $t_1 < t_2$ is a TTL-atom.
4. If v_1, v_2 are terms of sort $VALUE$, then $v_1 < v_2$ is a TTL-atom.

The set of well-formed TTL-formulae is defined inductively in a standard way using Boolean connectives and quantifiers over variables of TTL sorts. An example of the TTL formula, which describes observational belief creation of an agent, is given below:

In any trace, if at any point in time $t1$ the agent A observes that it is raining, then there exists a point in time $t2$ after $t1$ such that at $t2$ in the trace the agent A believes that it is raining.

$$\forall \gamma \; \forall t1 \; [holds(state(\gamma, t1), observation\_result(itsraining)) \Rightarrow$$
$$\exists t2 > t1 \;\; holds(state(\gamma, t2), belief(itsraining))]$$

The possibility to specify arithmetical operations in TTL allows modelling of continuous systems, which behaviour is usually described by differential equations. Such systems can be expressed in TTL either using discrete or dense time frames. For the discrete case, methods of numerical analysis that approximate a continuous model by a discrete one are often used, e.g., Euler's and Runge-Kutta methods [334]. For example, by applying Euler's method for solving a differential equation $dy/dt = f(y)$ with the initial condition $y(t_0) = y_0$, a difference equation $y_{i+1} = y_i + h * f(y_i)$ (with i the step number and $h > 0$ the step size) is obtained. This equation can be modelled in TTL in the following way:

$$\forall \gamma \; \forall t \; \forall v : VALUE \; holds(state(\gamma, t), \; has\_value(y, v)) \Rightarrow$$
$$holds(state(\gamma, t + 1), \; has\_value(y, v + h \bullet f(v)))$$

The traces γ satisfying the above dynamic property are the solutions of the difference equation.

Furthermore, a dense time frame can be used to express differential equations with derivatives specified using the epsilon-delta definition of a limit, which is expressible in TTL. To this end, the following relation is introduced, expressing that $x = dy/dt$:

is_diff_of$(\gamma, \mathbf{x}, \mathbf{y})$:

$$\forall t, w \; \forall \epsilon > 0 \; \exists \delta > 0 \; \forall t', v, v'$$
$$0 < dist(t', t) < \delta \; \& \; holds(state(\gamma, t), has\_value(x, w))$$

$\&holds(state(\gamma,t),has\_value(y,v))$

$\&holds(state(\gamma,t'),has\_value(y,v'))$

$\Rightarrow\ dist((v'-v)/(t'-t),w) < \epsilon$

where $dist(u,v)$ is defined as the absolute value of the difference.

Furthermore, a study has been performed in which a number of properties of continuous systems and theorems of calculus were formalized in TTL and used in reasoning [83].

11.3 Semantics of TTL

An *interpretation* of a TTL formula is based on the standard interpretation of an order sorted predicate logic formula and is defined by a mapping I that associates each:

1. sort symbol S to a certain set (subdomain) D_S, such that if $S \subseteq S'$ then $D_S \subseteq D'_S$
2. constant c of sort S to some element of D_S
3. function symbol f of type $< X_1,...,X_i > \rightarrow X_{i+1}$ to a mapping: $I(X_1)\times...\times I(X_i) \rightarrow I(X_{i+1})$
4. predicate symbol P of type $< X_1,...,X_i >$ to a relation on $I(X_1)\times...\times I(X_i)$

A *model* M for the TTL is a pair $M =< I,V >$, where I is an interpretation function, and V is a variable assignment, mapping each variable x of any sort S to an element of D_S. We write $V[x/v]$ for the assignment that maps variables y other than x to $V(y)$ and maps x to v. Analogously, we write $M[x/v] =< I,V[x/v] >$.

If $M =< I,V >$ is a model of the TTL, then *the interpretation of a TTL term* τ, denoted by τ^M, is inductively defined by:

1. $(x)^M = V(x)$, where x is a variable over one of the TTL sorts.
2. $(c)^M = I(c)$, where c is a constant of one of the TTL sorts.
3. $f(\tau_1,...,\tau_k)^M = I(f)(\tau_1^M,...,\tau_k^M)$, where f is a TTL function of type $S_1\times...\times S_n \rightarrow S$ and $\tau_1,...,\tau_n$ are terms of TTL sorts $S_1,...,S_n$.

The truth definition of TTL for the model $M =< I,V >$ is inductively defined by:

1. $\models_M P_i(\tau_1,...,\tau_k)$ iff $I(Pi)(\tau_1^M,...,\tau_k^M) = true$
2. $\models_M \neg\varphi$ iff $\not\models_M \varphi$
3. $\models_M \varphi\wedge\psi$ iff $\models_M \varphi$ and iff $\models_M \psi$
4. $\models_M \forall x(\varphi(x))$ iff $\models_{M[x/v]} \varphi(x)$ for all $v \in D_S$, where x is a variable of sort S.

The semantics of connectives and quantifiers is defined in the standard way. A number of important properties of TTL are formulated in form of axioms:

1. Equality of traces:
   $\forall \gamma_1, \gamma_2 \, [\forall t [state(\gamma_1, t) = state(\gamma_2, t)] \Rightarrow \gamma_1 = \gamma_2]$
2. Equality of states:
   $\forall s_1, s_2 [\forall a : STATPROP [truth\_value(s_1, a) = truth\_value(s_2, a)] \Rightarrow s_1 = s_2]$
3. Truth value in a state:
   $holds(s, p) \Leftrightarrow truth\_value(s, p) = true$
4. State consistency axiom:
   $\forall \gamma, t, p \; (holds(state(\gamma, t), p) \Rightarrow \neg holds(state(\gamma, t), not(p)))$
5. State property semantics:
   a. $holds(s, (p1 \wedge p2)) \Leftrightarrow holds(s, p_1) \; \& \; holds(s, p_2)$
   b. $holds(s, (p1 \vee p2)) \Leftrightarrow holds(s, p_1) \mid holds(s, p_2)$
   c. $holds(s, not(p_1)) \Leftrightarrow \neg holds(s, p_1)$
6. For any constant variable name x from the sort S^{VARS}:
   $holds(s, (\exists (x, F))) \Leftrightarrow \exists x' : S^{GTERMS} \; holds(s, G)$, and $holds(s, (\forall (x, F))) \Leftrightarrow \forall x' : S^{GTERMS} \; holds(s, G)$ with G, F terms of sort $STATPROP$, where G is obtained from F by substituting all occurrences of x by x'.
7. Partial order axioms for the sort $TIME$:
   a. $\forall t \; t \leq t$ (Reflexivity)
   b. $\forall t_1, t_2 \; [t_1 \leq t_2 \wedge t_2 \leq t_1] \Rightarrow t_1 = t_2$ (Anti-Symmetry)
   c. $\forall t_1, t_2, t_3 \; [t_1 \leq t_2 \wedge t_2 \leq t_3] \Rightarrow t_1 \leq t_3$ (Transitivity)
8. Axioms for the sort $VALUE$: the same as for the sort $TIME$ and standard arithmetic axioms.
9. Axioms, which relate the sorts $TIME$ and $VALUE$:
   a. $(t + v_1) + v_2 = t + (v_1 + v_2)$
   b. $(t \bullet v_1) \bullet v_2 = t \bullet (v_1 \bullet v_2)$
10. (Optional) Finite variability property (for any trace γ).
    This property ensures that a trace is divided into intervals such that the overall system state is stable within each interval, i.e., each state property changes its truth value at most a finite number of times:
    $\forall t_0, t_1 \; t_0 < t_1 \Rightarrow \exists \delta > 0 [\forall t [t_0 \leq t \; \& \; t \leq t_1] \Rightarrow \exists t_2 [t_2 \leq t \; \& \; t < t_2 + \delta \; \& \; \forall t_3 [t_2 \leq t_3 \; \& \; t_3 \leq t_2 + \delta]] \Rightarrow state(\gamma, t_3) = state(\gamma, t)]$

11.4 Multi-level Modelling of Multi-Agent Systems in TTL

With increase of the number of elements within a multi-agent system, the complexity of the dynamics of the system grows considerably. To analyze the behaviour of a complex multi-agent system (e.g., for critical domains such as air traffic control and health care), appropriate approaches for handling the dynamics of the multi-agent system are important. Two of such approaches for TTL specifications of multi-agent systems are considered in this section: aggregation by agent clustering is considered in Section 11.4.1 and organisation structures are discussed in Section 11.4.2.

11.4.1 Aggregation by agent clustering

One of the approaches to manage complex dynamics is by distinguishing *different aggregation levels*, based on clustering of a multi-agent system into parts or components with further specification of their dynamics and relations between them; e.g., [264]. At the lowest aggregation level a component is an agent or an environmental object (e.g., a database), with which agents interact. Further, at higher aggregation levels a component has the form of either a group of agents or a multi-agent system as a whole. In the simplest case two levels can be distinguished: the lower level at which agents interact and the higher level, where the whole multi-agent system is considered as one component. In the general case the number of aggregation levels is not restricted. Components interact with each other and the environment via input and output interfaces described in terms of interaction (i.e., input and output) ontologies. A component receives information at its input interface in the form of observation results and communication from other components. A component generates at its output communication, observation requests and actions performed in the environment. Some elements from the agent's interaction ontology are provided in Table 11.1.

Table 11.1 Interaction ontology

Ontology element	Description
$observation\_request\_from\_for(C : COMPONENT, I : INFO\_ELEMENT)$	I is to be observed in the world for C (active observation)
$observation\_result\_to\_for(C : COMPONENT, I : INFO\_ELEMENT)$	Observation result I is provided to C (for active observation)
$observed(I : INFO\_ELEMENT)$	I is observed at the component's input (passive observation)
$communicated\_from\_to(C1 : COMPONENT, C2 : COMPONENT, s\_act : SPEECH\_ACT, I : INFO\_ELEMENT)$	Specifies speech act s_act (e.g., inform, request, ask) from $C1$ to $C2$ with the content I
$to\_be\_performed(A : ACTION)$	Action A is to be performed

For the explicit indication of an aspect of a state for a component, to which a state property is related, sorts $ASPECT\_COMPONENT$ (a set of the component aspects of a system; i.e., input, output, internal); $COMPONENT$ (a set of all component names of a system); $COMPONENT\_STATE\_ASPECT$ (a set of all names of aspects of all component states) and a function symbol

$$comp\_aspect : ASPECT\_COMPONENT \times COMPONENT \rightarrow COMPONENT\_STATE\_ASPECT$$

are used. In multi-agent system specifications, in which the indication of the component's aspects is needed, the definition of the function symbol state introduced earlier is extended as $state : TRACE \times TIME \times COMPONENT\_STATE\_ASPECT \rightarrow STATE$. For example,

$$holds(state(trace1, t1, input(A)), observation\_result(sunny\_weather))$$

Here $input(A)$ belongs to sort $COMPONENT\_STATE\_ASPECT$.
At every aggregation level the behaviour of a component is described by a set of dynamic properties. The dynamic properties of components of a higher aggregation level may have the form of a few temporal expressions of high complexity. At a lower aggregation level a system is described in terms of more basic steps. This usually takes the form of a specification consisting of a large number of temporal expressions in a simpler format. Furthermore, the dynamic properties of a component of a higher aggregation level can be logically related by an interlevel relation to dynamic properties of components of an adjacent lower aggregation level. This interlevel relation takes the form that a number of properties of the lower level logically entail the properties of the higher level component.

In the following a running example used throughout the chapter is introduced to illustrate aggregation by agent clustering in a multi-agent system for co-operative information gathering. For simplicity, this system is considered at two aggregation levels (see Figure 11.1). At the higher level the multi-agent system as a whole is considered. At the lower level four components and their interactions are specified: two information gathering agents A and B, agent C, and environment component E representing the external world. Each of the agents is able to acquire partial information from an external source (component E) by initiated observations. Each agent can be reactive or proactive with respect to the information acquisition process. An agent is proactive if it is able to start information acquisition independently of requests of any other agents, and an agent is reactive if it requires a request from some other agent to perform information acquisition.

Observations of any agent taken separately are insufficient to draw conclusions of a desired type; however, the combined information of both agents is sufficient. Therefore, the agents need to co-operate to be able to draw conclusions. Each agent can be proactive with respect to the conclusion generation, i.e., after receiving both observation results an agent is capable to generate and communicate a conclusion

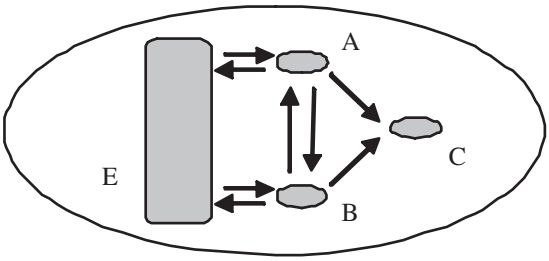

Fig. 11.1 The co-operative information gathering multi-agent system. *A* and *B* represent information gathering agents; *C* is an agent that obtains the conclusion information; *E* is an environmental component.

to agent C. Moreover, an agent can be request pro-active to ask information from another agent, and an agent can be pro-active or reactive in provision of (already acquired) information to the other agent.

For the lower-level components of the multi-agent system, a number of dynamic properties were identified and formalized as it is shown below. In the formalization the variables $A1$ and $A2$ are defined over the sort $AGENT^{TERMS}$, the constant E belongs to the sort $ENVIRONMENTAL\_COMPONENT^{GTERMS}$, the variable IC is defined over the sort $INFORMATION\_CHUNK^{TERMS}$, the constants $IC1$, $IC2$ and $IC3$ belong to the sort $INFORMATION\_CHUNK^{GTERMS}$ and the constant C belongs to the sort $AGENT^{TERMS}$.

DP1(A1, A2) (Effectiveness of information request transfer between agents)

If agent $A1$ communicates a request for an information chunk to agent $A2$ at any time point $t1$, then this request will be received by agent $A2$ at time point $t1+c$.

$\forall IC \forall t1$
$[\ holds(state(\gamma, t1, output(A1)), communicated\_from\_to(A1, A2, request, IC)))$
$\Rightarrow holds(state(\gamma, t1+c, input(A2)),$
$communicated\_from\_to(A1, A2, request, IC))]$

DP2(A1, A2) (Effectiveness of information transfer between agents)

If agent $A1$ communicates information chunk to agent $A2$ at any time point $t1$, then this information will be received by agent $A2$ at the time point $t1+c$.

$\forall IC \forall t1$
$[\ holds(state(\gamma, t1, output(A1)), communicated\_from\_to(A1, A2, inform, IC)))$
$\Rightarrow holds(state(\gamma, t1+c, input(A2)),$
$communicated\_from\_to(A1, A2, inform, IC)))]$

DP3(A1, E) (Effectiveness of information transfer between an agent and environment)

If agent $A1$ communicates an observation request to the environment at any time point $t1$, then this request will be received by the environment at the time point $t1+c$.

$$\forall IC \forall t1\ [holds(state(\gamma, t1, output(A1)), observation\_request\_from\_for(A1, IC)) \Rightarrow holds(state(\gamma, t1+c, input(E)), observation\_request\_from\_for(A1, IC))]$$

DP4(A1, E) (Information provision effectiveness)

If the environment receives an observation request from agent $A1$ at any time point $t1$, then the environment will generate a result for this request at the time point $t1+c$.

$$\forall IC \forall t1[holds(state(\gamma, t1, input(E)), observation\_request\_from\_for(A1, IC)) \Rightarrow holds(state(\gamma, t1+c, output(E)), observation\_result\_to\_for(A1, IC))]$$

DP5(E, A1) (Effectiveness of information transfer between environment and an agent)

If the environment generates a result for an agent's information request at any time point $t1$, then this result will be received by the agent at the time point $t1+c$.

$$\forall IC \forall t1[holds(state(\gamma, t1, output(E)), observation\_result\_to\_for(A1, IC)) \Rightarrow holds(state(\gamma, t1+c, input(A1)), observation\_result\_to\_for(A1, IC))]$$

DP6(A1, A2) (Information acquisition reactiveness)

If agent $A2$ receives a request for an information chunk from agent $A1$ at any time point $t1$, then agent $A2$ will generate a request for this information to the environment at the time point $t1+c$.

$$\forall IC \forall t1[holds(state(\gamma, t1, input(A2)), communicated\_from\_to(A1, A2, request, IC)) \Rightarrow holds(state(\gamma, t1+c, output(A2)), observation\_result\_to\_for(A2, IC))]$$

DP7(A1, A2) (Information provision reactiveness)

If exists a time point $t1$ when agent $A2$ received a request for a chunk of information from agent $A1$, then for all time points $t2$ when the requested information is provided to agent $A2$, this information will be further provided by agent $A2$ to agent $A1$ at the time point $t2+c$.

$$\forall IC\ [\exists t1\ [\ t1 < t\ \&\ holds(state(\gamma, t1, input(A2)), communicated\_from\_to(A1, A2, request, IC))]] \Rightarrow \forall t2[\ t < t2\ \&\ holds(state(\gamma, t2, input(A2)), observation\_result\_to\_for(A2, IC)) \Rightarrow holds(state(\gamma, t2+c, output(A2)), communicated\_from\_to(A2, A1, inform, IC))]]$$

DP8(A1, A2) (Conclusion proactiveness)

For any time points $t1$ and $t2$, if agent $A1$ receives a result for its observation request from the environment (information chunk $IC1$) at $t1$ and it receives information required for the conclusion generation from agent $A2$ (information chunk $IC2$) at $t2$, then agent $A1$ will generate a conclusion based on the received information (information chunk $IC3$) to agent C at a time point $t4$ later than $t1$ and $t2$.

$$\forall t1, t2\ t1 < t\ \&\ t2 < t\&$$
$$holds(state(\gamma, t1, input(A1)), observation\_result\_to\_for(A1, IC1))\&$$
$$holds(state(\gamma, t2, input(A1)), communicated\_from\_to(A2, A1, inform, IC2))$$
$$\Rightarrow \exists\, t4 > t\&$$
$$[holds(state(\gamma, t4, output(A1)), communicated\_from\_to(A1, C, inform, IC3))]$$

DP9(A1, E) (Information acquisition proactiveness)

At some time point an observation request for information chunk IC1 is generated by agent $A1$ to the environment.

$$holds(state(\gamma, c, output(A1)), observation\_request\_from\_for(A1, IC1))$$

DP10(A1, A2) (Information request proactiveness)

At some time point a request for information chunk IC2 is communicated by agent $A1$ to agent $A2$.

$$holds(state(\gamma, c, output(A1)), communicated\_from\_to(A1, A2, request, IC2))$$

11.4.2 Organisation structures

Organisations have proven to be a useful paradigm for analyzing and designing multi-agent systems [146, 172]. Representation of a multi-agent system as an organisation consisting of roles and groups can tackle major drawbacks concerned with traditional multi-agent models; e.g., high complexity and poor predictability of dynamics in a system [172]. We adopt a generic representation of organisations, abstracted from instances of real agents. As has been shown in [240], organisational structure can be used to limit the scope of interactions between agents, reduce or explicitly increase redundancy of a system, or formalize high-level system goals, of which a single agent may be not aware. Moreover, organisational research has recognized the advantages of agent-based models; e.g., for analysis of structure and dynamics of real organisations.

An *organisation structure* is described by relationships between roles at the same and at adjoining aggregation levels and between parts of the conceptualized environment and roles. The specification of an organisation structure uses the following elements:

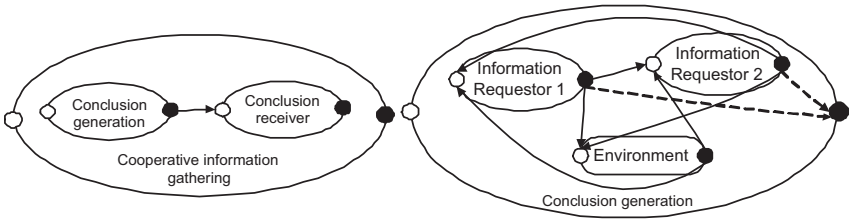

Fig. 11.2 An organisation structure for the co-operative information gathering multi-agent system represented at the aggregation level 2 (left) and at the aggregation level 3 (right).

1. A *role* represents a subset of functionalities, performed by an organisation, abstracted from specific agents who fulfil them.
   Each role can be composed by several other roles, until the necessary detailed level of aggregation is achieved, where a role that is composed of (interacting) subroles, is called a composite role. Each role has an input and an output interface, which facilitate in the interaction (communication) with other roles. The interfaces are described in terms of interaction (input and output) ontologies. At the highest aggregation level, the whole organisation can be represented as one role. Such representation is useful both for specifying general organisational properties and further utilizing an organisation as a component for more complex organisations. Graphically, a role is represented as an ellipse with white dots (the input interfaces) and black dots (the output interfaces). Roles and relations between them are specified using sorts and predicates from the structure ontology (see Table 11.2). For the example of co-operative information gathering system considered in Section 11.4.1, an organisation structure may be defined as shown in Figure 11.2. The structure is represented at three aggregation levels: at the first level the organization as a whole is considered, at the second level the Co-operative information gathering role with its subroles is considered; at the third aggregation level the Conclusion generation role with its subroles is represented.
2. An *interaction link* represents an information channel between two roles at the same aggregation level. Graphically, it is depicted as a solid arrow, which denotes the direction of possible information transfer.
3. *The conceptualized environment* represents a special component of an organisation model. Similarly to roles, the environment has input and output interfaces, which facilitate in the interaction with roles of an organisation. The interfaces are conceptualized by the environment interaction (input and output) ontologies.
4. An *interlevel link* connects a composite role with one of its subroles. It represents information transfer between two adjacent aggregation levels. It may describe an ontology mapping for representing mechanisms of information abstraction. Graphically, it is depicted as a dashed arrow, which shows the direction of the interlevel transition.

Table 11.2 Ontology for formalizing organizational structure

Predicate	Description
$is\_role : ROLE$	Specifies a role in an organization
$has\_subrole : ROLE \times ROLE$	For a subrole of a composite role
$source\_of\_interaction : ROLE \times INTERACTION\_LINK$	Specifies a source role of an interaction
$destination\_of\_interaction : ROLE \times INTERACTION\_LINK$	Specifies a destination role of interaction
$interlevel\_connection\_from : ROLE \times INTERLEVEL\_LINK$	Identifies a source role of an interlevel link
$interlevel\_connection\_to : ROLE \times INTERLEVEL\_LINK$	Identifies a destination role of an interlevel link
$part\_of\_env\_in\_interaction : ENVIRONMENT \times ENVIRONMENT\_INTERACTION\_LINK$	Identifies the conceptualized part of the environment involved in interaction with a role
$has\_input\_ontology : ROLE \times ONTOLOGY$	Specifies an input ontology for a role
$has\_output\_ontology : ROLE \times ONTOLOGY$	Specifies an output ontology for a role
$has\_input\_ontology : ENVIRONMENT \times ONTOLOGY$	Specifies an input ontology for the environment
$has\_output\_ontology : ENVIRONMENT \times ONTOLOGY$	Specifies an output ontology for the environment
$has\_interaction\_ontology : ROLE \times ONTOLOGY$	Specifies an interaction ontology for a role

At each aggregation level, it can be specified how the organization's behaviour is assumed to be. The dynamics of each structural element are defined by the specification of a set of dynamic properties. We define five types of dynamic properties:

1. A *role property* (RP) describes the relationship between input and output states of a role, over time. For example, a role property of Information requester 2 is:

   Information acquisition reactiveness

   $\forall IC \forall t1 [holds(state(\gamma, t1, input(InformationRequester2)),$
   $communicated\_from\_to(InformationRequester1, InformationRequester2,$
   $request, IC))$

   $\Rightarrow holds(state(\gamma, t1 + c, output(InformationRequester2)),$
   $observation\_result\_to\_for(InformationRequester2, IC))]$

2. A *transfer property* (TP) describes the relationship of the output state of the source role of an interaction link to the input state of the destination role. For example, a transfer property for the roles Information requester 1 and Information requester 2 is:

   Effectiveness of information transfer between roles

   $\forall IC \forall t1 [holds(state(\gamma, t1, output(InformationRequester1)),$
   $communicated\_from\_to(InformationRequester1, InformationRequester2,$
   $inform, IC)))$

$\Rightarrow holds(state(\gamma, t1+c, input(InformationRequester2)),$
$communicated\_from\_to(InformationRequester1, InformationRequester2,$
$inform, IC)))]$

3. An *interlevel link property* (ILP) describes the relationship between the input or output state of a composite role and the input or output state of its subrole. Note that an interlevel link is considered to be instantaneous: it does not represent a temporal process, but gives a different view (using a different ontology) on the same information state. An interlevel transition is specified by an ontology mapping, which can include information abstraction.
4. An *environment property* (EP) describes a temporal relationship between states or properties of objects of interest in the environment.
5. An *environment interaction property* (EIP) describes a relation either between the output state of the environment and the input state of a role (or an agent) or between the output state of a role (or an agent) and the input state of the environment. For example,

Effectiveness of information transfer between a role and environment

$\forall IC \forall t1\ [holds(state(\gamma, t1, output(InformationRequester1)),$
$observation\_request\_from\_for(InformationRequester1, IC))$
$\Rightarrow holds(state(\gamma, t1+c, input(E)),$
$observation\_request\_from\_for(InformationRequester1, IC))]$

The specifications of organisation structural relations and dynamics are imposed onto the agents, who will eventually enact the organisational roles. For more details on organisation-oriented modelling of multi-agent systems we refer to [263].

11.5 Relation to Other Languages

In this section TTL is compared to a number of existing languages for modelling dynamics of a system.

Executable languages can be defined as sublanguages of TTL. An example of such a language, which was designed for simulation of dynamics in terms of both qualitative and quantitative concepts, is the LEADSTO language, cf. [81]. The LEADSTO language models direct temporal or causal dependencies between two state properties in states at different points in time as follows. Let α and β be state properties of the form 'conjunction of atoms or negations of atoms', and e, f, g, h non-negative real numbers (constants of sort $VALUE$). In LEADSTO the notation $\alpha \longrightarrow_{e,f,g,h} \beta$, means:

If state property α holds for a certain time interval with duration g, then after some delay (between e and f) state property β will hold for a certain time interval of length h.

A specification in LEADSTO format has as advantages that it is executable and that it can often easily be depicted graphically, in a causal graph or system dynamics style. In terms of TTL, the fact that the above statement holds for a trace γ is expressed as follows:

$\forall t1[\forall t[t1 - g \leq t \;\&\; t < t1 \Rightarrow holds(state(\gamma, t), \alpha)] \Rightarrow$

$\exists d : VALUE[e \leq d \;\&\; d \leq f \;\&\; \forall t'[t1 + d \leq t' \;\&\; t' < t1 + d + h \Rightarrow$

$holds(state(\gamma, t'), \beta)]$

Furthermore, TTL has some similarities with the situation calculus [365] and the event calculus [272]. However, a number of important syntactic and semantic distinctions exist between TTL and both calculi. In particular, the central notion of the situation calculus - a situation - has different semantics than the notion of a state in TTL. That is, by a situation is understood a history or a finite sequence of actions, whereas a state in TTL is associated with the assignment of truth values to all state properties (a 'snapshot' of the world). Moreover, in contrast to situation calculus, where transitions between situations are described by execution of actions, in TTL action executions are used as properties of states.

Moreover, although a time line has been introduced to the situation calculus [339], still only a single path (a temporal line) in the tree of situations can be explicitly encoded in the formulae. In contrast, TTL provides more expressivity by allowing explicit references to different temporally ordered sequences of states (traces) in dynamic properties. For example, this can be useful for expressing the property of trust monotonicity:

> For any two traces $\gamma 1$ and $\gamma 2$, if at each time point t agent A's experience with public transportation in $\gamma 2$ at t is at least as good as A's experience with public transportation in $\gamma 1$ at t, then in trace $\gamma 2$ at each point in time t, A's trust is at least as high as A's trust at t in trace $\gamma 1$.

$\forall \gamma 1, \gamma 2[\forall t, \forall v1 : VALUE[holds(state(\gamma 1, t), has\_value(experience, v1))\&$

$[\forall v2 : VALUE\ holds(state(\gamma 2, t), has\_value(experience, v2) \rightarrow v1 \leq v2)]] \Rightarrow$

$[\forall t, \forall w1 : VALUE[holds(state(\gamma 1, t), has\_value(trust, w1))\&$

$[\forall w2 : VALUE\ holds(state(\gamma 2, t), has\_value(trust, w2) \rightarrow w1 \leq w2)]]]]$

In contrast to the event calculus, TTL does not employ the mechanism of events that initiate and terminate fluents. Event occurrences in TTL are considered to be state occurrences the external world. Furthermore, similarly to the situation calculus, also in the event calculus only one time line is considered.

Formulae of the loosely guarded fragment of the first-order predicate logic [16], which is decidable and has good computational properties (deterministic exponential time complexity), are also expressible in TTL:

$\exists y((\alpha_1 \wedge ... \wedge \alpha_m) \wedge \psi(x, y))$ or $\forall y((\alpha_1 \wedge ... \wedge \alpha_m) \rightarrow \psi(x, y))$,

where x and y are tuples of variables, $\alpha_1 ... \alpha_m$ are atoms that relativize a quantifier (the guard of the quantifier), and $\psi(x,y)$ is an inductively defined formula in the guarded fragment, such that each free variable of the formula is in the set of free variables of the guard.

Similarly the fluted fragment [348] and $\exists * \forall *$ [3] can be considered as sublanguages of TTL.

TTL can also be related to temporal languages that are often used for verification (e.g., LTL and CTL [39, 198]). For example, dynamic properties expressed as formulae in LTL can be translated to TTL by replacing the temporal operators of LTL by quantifiers over time. E.g., consider the LTL formula

$$\mathbf{G}(observation\_result(itsraining) \rightarrow \mathbf{F}(belief(itsraining)))$$

where the temporal operator **G** means 'for all later time points', and **F** 'for some later time point'. The first operator can be translated into a universal quantifier, whereas the second one can be translated into an existential quantifier.

Using TTL, this formula then can be expressed, for example, as follows:

$$\forall t1[holds(state(\gamma, t1), observation\_result(itsraining)) \Rightarrow$$
$$\exists t2 > t1 \; holds(state(\gamma, t2), belief(itsraining))]$$

Note that the translation is not bi-directional, i.e., it is not always possible to translate TTL expressions into LTL expressions due to the limited expressive power of LTL. For example, the property of trust monotonicity specified in TTL above cannot be expressed in LTL because of the explicit references to different traces. Similar observations apply for other well-known modal temporal logics such as CTL.

In contrast to the logic of McDermott [309], TTL does not assume structuring of traces in a tree. This enables reasoning about independent sequences of states (histories) in TTL (e.g., by comparing them), which is also not addressed by McDermott.

11.6 Normal Forms and Transformation Procedures

In this Section, normal forms for TTL formulae and the related transformation procedures are described. Normal forms create the basis for the automated analysis of TTL specifications, which is addressed later in this chapter. In Section 11.6.1 the past implies future normal form and a procedure for transformation of any TTL formula into this form are introduced. In Section 11.6.2 the executable normal form and a procedure for transformation of TTL formulae in the past implies future normal form into the executable normal form are described. A procedure for abstraction of executable specifications is described in Section 11.6.3.

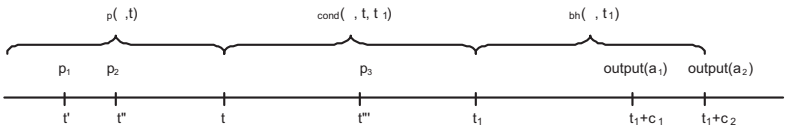

Fig. 11.3 Graphical illustration of the structure of the past implies future normal form

11.6.1 Past Implies Future Normal Form

First, the past implies future normal form is introduced.

Definition 11.1. (Past Implies Future Normal Form) The past implies future normal form for TTL formulae is specified by a logical implication from a temporal input pattern to a temporal output pattern:

$$[\varphi_p(\gamma,t) \Rightarrow \varphi_f(\gamma,t)],$$

where $\varphi_p(\gamma,t)$ is a past statement (i.e., for all time variables s in $\varphi_p(\gamma,t) : s \leq t$ or $s < t$) and $\varphi_f(\gamma,t)$ is a future statement (i.e., for all time variables s in $\varphi_f(\gamma,t) : s \geq t$ or $s > t$). The future statement is represented in the form of a conditional behaviour:

$$\varphi_f(\gamma,t) \Leftrightarrow \forall t_1 > t[\varphi_{cond}(\gamma,t,t_1) \Rightarrow \varphi_{bh}(\gamma,t_1)],$$

where $\varphi_{cond}(\gamma,t,t_1)$ is an interval statement over the interaction ontology, which describes a condition for some specified action(s) and/or communication(s), and $\varphi_{bh}(\gamma,t_1)$ is a (conjunction of) future statement(s) for t_1 over the output ontology of the form $holds(state(\gamma,t_1+c),output(a))$, for some integer constant c and action or communication a.

A graphical illustration of the structure of the past implies future normal form is given in Figure 11.3. When a past formula $\varphi_p(\gamma,t)$ is true for γ at time t, a potential to perform one or more action(s) and/or communication(s) exists. This potential is realized at time t_1 when the condition formula $\varphi_{cond}(\gamma,t,t_1)$ becomes true, which leads to the action(s) and/or communication(s) being performed at the time point(s) $t_1 + c$ indicated in $\varphi_{bh}(\gamma,t_1)$.

For example, the dynamic property $DP7(A1,A2)$ (Information provision reactiveness) from the specification of co-operative information gathering multi-agent system from Section 11.4.1 can be specified in the past implies future normal form $\varphi_p(\gamma,t) \Rightarrow \varphi_f(\gamma,t)$, with $\varphi_p(\gamma,t)$ is a formula

$\exists t2 \leq t$ & $holds(state(\gamma,t2,input(A2)),communicated\_from\_to(A1,A2,request,IC))$

and $\varphi_f(\gamma,t)$ is a formula

$\forall t1 > t\ [holds(state(\gamma,t1,input(A2)),observation\_result\_to\_for(A2,IC)) \Rightarrow$

$holds(state(\gamma,t1+c,output(A2)),communicated_from_to(A2,A1,inform,IC))]$

with $\varphi_{cond}(\gamma,t,t1)$ is a formula

$holds(state(\gamma,t1,input(A2)),observation\_result\_to\_for(A2,IC))$

and $\varphi_{bh}(\gamma,t1)$ is $holds(state(\gamma,t1+c,output(A2)),communicated\_from\_to(A2,A1,inform,IC))]$,

where t is the present time point with respect to which the formulae are evaluated and c is some natural number.

In general, any TTL formula can be automatically translated into the past implies future normal form. The transformation procedure is based on a number of steps. First, the variables in the formula are related to the given t (current time point) by differentiation. The resulting formula is rewritten in prenex conjunctive normal form. Each clause in this formula is reorganised in past implies future format. Finally, the quantifiers are distributed over and within these implications. Now consider the detailed description of these steps (for a more profound description of the procedure see [418]).

Differentiating Time Variables A formula is rewritten into an equivalent one such that time variables that occur in this formula always either are limited (relativized) to past or to future time points with respect to t. As an example, suppose $\psi(t_1,t_2)$ is a formula in which time variables t_1, t_2 occur. Then, different cases of ordering relation for each of the time variables with respect to t are considered: $t_1 < t$, $t_1 \geq t$ and $t_2 < t$, $t_2 \geq t$, i.e., in combination four cases: $t_1 < t$ and $t_2 < t$, $t_1 < t$ and $t_2 \geq t$, $t_1 \geq t$ and $t_2 < t$, $t_1 \geq t$ and $t_2 \geq t$. To eliminate ambiguity, for $t_i < t$ the variable t_i is replaced by (*past time variable*) u_i, for $t_i \geq t$ by (*future time variable*) v_i.

The following transformation step introduces for any occurring time variable t_i a differentiation into a pair of new time variables: u_i used over the past and v_i used over the future with respect to t.

For any occurrence of a universal quantifier over t_i:

$\forall t_i\ A \longmapsto [\forall u_i < t\ A[u_i/t_i] \wedge \forall v_i \geq t\ A[v_i/t_i]]$

For any occurrence of an existential quantifier over t_i:

$\exists t_i\ A \longmapsto [\exists u_i < t\ A[u_i/t_i] \vee \exists v_i \geq t\ A[v_i/t_i]]$

Assuming differentiation of time variables into past and future time variables, state-related atoms (in which only one time variable occurs) can be classified in a straightforward manner as a past atom or future atom. For example, atoms of the form $holds(state(\gamma,u_i),p)$ are past atoms and $holds(state(\gamma,v_j),p)$ are future atoms. For non-unary relations, in the special case of the time ordering relation ¡ the ordering axioms are given, e.g., transitivity. Atoms that are mixed (containing both a past and a future variable) are eliminated by the following transformation rules:

$u_i = v_j \rightarrow false \quad v_j < u_i \rightarrow false \quad u_i < v_j \rightarrow true$

Obtaining prenex conjunctive normal form This step is performed using a well-known transformation procedure [180].

From a Clause to a Past to Future Implication By partitioning the set of occurring atoms into past atoms and future atoms, it is not difficult to rewrite a clause into a past to future implication format: transform a clause C into an implication of the form $A \rightarrow B$ where A is the conjunction of the negations of all past literals in C and B is the disjunction of all future literals in C. Thus, a quantifier free formula in Conjunctive Normal Form can be transformed into a conjunction of implications from past to future by the transformation rule

$$\vee\ PL_i \vee \vee FL_j \ \longmapsto \wedge\ \sim PL_i\ \rightarrow \vee\ FL_j$$

where the past and future literals are indicated by PL_i and FL_j, respectively, and if a is an atom, $\sim a = \neg a$, and $\sim \neg a = a$.

Distribution of Quantifiers Over Implications The quantifiers can be rewritten to quantifiers with a single implication as their scope, and even one step further, to quantifiers with a single antecedent or a single consequent of an implication as their scope. Notice that quantifiers addressed here are both time quantifiers and non-time quantifiers.

Let φ be a formula in the form of a conjunction of past to future implications $\wedge_{i\in I}[A_i \rightarrow B_i]$ and let x be a (either past or future) variable occurring in φ. The following transformation rules handle existential quantifiers for variables in one or more of the B_i, respectively in one or more of the A_i. Here P denotes taking the power set.

1. if x occurs in the B_i but does not occur in the A_i :
   $\exists x\ \wedge_{i\in I}\ [A_i \rightarrow B_i] \longmapsto \wedge_{j\in P(I)}\ \exists x[\wedge_{i\in j} A_i \rightarrow \wedge_{i\in j} B_i]$
   $\exists x[\wedge_{i\in j} A_i \rightarrow \wedge_{i\in j} B_i] \longmapsto [\wedge_{i\in j} A_i \rightarrow \exists x \wedge_{i\in j} B_i]$
2. if x occurs in the A_i but does not occur in the B_i :
   $\exists x \wedge_{i\in I} [A_i \rightarrow B_i] \longmapsto \wedge_{j\in P(I)} \exists x[\vee_{i\in j} A_i \rightarrow \vee_{i\in j} B_i]$
   $\exists x[\vee_{i\in j} A_i \rightarrow \vee_{i\in j} B_i] \longmapsto [\forall x[\vee_{i\in j} A_i] \rightarrow \vee_{i\in j} B_i]$

The following transformation rules handle universal quantifiers for variables in one or more of the B_i, respectively in one or more of the A_i:

1. if x occurs in the A_i or in the B_i :
   $\forall x\ \wedge_{i\in I}\ [A_i \rightarrow B_i]\ \longmapsto\ \wedge_{i\in I} \forall x[A_i \rightarrow B_i]$
2. if x occurs in the B_i but does not occur in the A_i :
   $\forall x[A_i \rightarrow B_i]\ \longmapsto\ A_i \rightarrow \forall x B_i$
3. if x occurs in the A_i but does not occur in the B_i :
   $\forall x[A_i \rightarrow B_i]\ \longmapsto\ [\exists x A_i] \rightarrow B_i$

11.6.2 Executable Normal Form

Although the past implies future normal form imposes a standard structure on the representation of TTL formulae, it does not guarantee the executability of formulae, required for automated analysis methods (i.e., some formulae may still contain complex temporal relations that cannot be directly represented in analysis tools). Therefore, to enable automated analysis, normalized TTL formulae should be translated into an executable normal form.

Definition 11.2. Executable Normal Form A TTL formula is in executable normal form if it has one of the following forms, for certain state properties , X and Y with $X \neq Y$, and integer constant c.

1. $\forall t\ holds(state(\gamma,t),X) \Rightarrow holds(state(\gamma,t+c),Y)$ (states relation property)
2. $\forall t\ holds(state(\gamma,t),X) \Rightarrow holds(state(\gamma,t+1),X)$ (persistency property)
3. $\forall t\ holds(state(\gamma,t),X) \Rightarrow holds(state(\gamma,t),Y)$ (state relation property)

For the translation postulated internal states of a component(s) specified in the formula, are used. These auxiliary states include memory states that are based on (input) observations (sensory representations) or communications ($memory : LTIME^{TERMS} \times STATPROP \rightarrow STATROP$). For example, $memory(t, observed(a))$ expresses that the component has memory that it observed a state property a at time point t. Furthermore, before performing an action or communication it is postulated that a component creates an internal preparation state. For example, $preparation\_for(b)$ represents a preparation of a component to perform an action or a communication.

In the following a transformation procedure from the normal form $[\varphi_p(\gamma,t) \Rightarrow \varphi_f(\gamma,t)]$ for the property $\varphi_p(\gamma,t)$ to the executable normal form is described and illustrated for the property $DP7(A1,A2)$ (Information provision reactiveness) considered above. For a more profound description of the transformation procedure we refer to [398].

First, an intuitive explanation for the procedure is provided. The procedure transforms a non-executable dynamic property in a number of executable properties. These properties can be seen as an execution chain, which describes the dynamics of the non-executable property. In this chain each unit generates intermediate states, used to link the following unit. In particular, first a number of properties are created to generate and maintain memory states (step 1 below). These memory states are used to store information about the past dynamics of components, which is available afterwards at any point in time. Then, executable properties are created to generate preparation for output and output states of components (steps 2 and 3 below). In these properties temporal patterns based on memory states are identified required for generation of particular outputs of components. In the end all created properties are combined in one executable specification.

More specifically, the transformation procedure consists of the following steps:

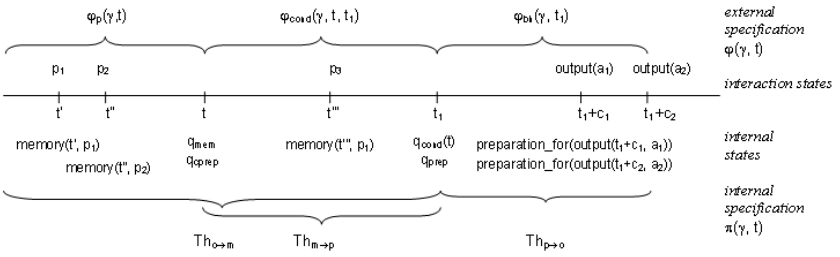

Fig. 11.4 A graphical representation of relations between interaction states described by a non-executable dynamic property and internal states described by executable rules.

1. Identify executable temporal properties, which describe transitions from the component states specified in $\varphi_p(\gamma,t)$ to memory states (for a graphical representation of relations between the states considered in this procedure see Figure 11.4).
   The general rules that form the basis for the executable properties are the following:
   $\forall t'\ holds(state(\gamma,t'),p) \Rightarrow holds(state(\gamma,t'),memory(t',p))$
   $\forall t''\ holds(state(\gamma,t''),memory(t',p)) \Rightarrow\ holds(state(\gamma,t''+1),memory(t',p))$

   Furthermore, at this step the memory formula $\varphi_{mem}(\gamma,t)$ is defined that is obtained by replacing all occurrences in $\varphi_p(\gamma,t)$ of subformulae of the form $holds(state(\gamma,t'),p)$ by $holds(state(\gamma,t),memory(t',p))$. According to Lemma 1 (given in [398]) $\varphi_{mem}(\gamma,t)$ is equivalent to some formula $\delta^*(\gamma,t)$ of the form $holds(state(\gamma,t),q_{mem}(t))$, where $q_{mem}(t)$ is called *the normalized memory state formula* for $\varphi_{mem}(\gamma,t)$, which uniquely describes the present state at the time point t by a certain history of events. Moreover, q_{mem} is the state formula $\forall u'\ [present\_time(u') \rightarrow q_{mem}(u')]$.

   For the property $DP7(A1,A2)$:

   $\forall t'\ holds(state(\gamma,t',input(A2)),communicated\_from\_to(A1,A2,request,IC)))$
   $\Rightarrow\ holds(state(\gamma,t',internal(B)),$
   $memory(t',communicated\_from\_to(A1,A2,request,IC))))$
   $\forall t''\ holds(state(\gamma,t'',internal(B)),$
   $memory(t',communicated\_from\_to(A1,A2,request,IC)))) \Rightarrow$
   $holds(state(\gamma,t''+1,internal(B)),$
   $memory(t',communicated\_from\_to(A1,A2,request,IC))))$

2. Identify executable temporal properties, which describe transitions from memory states to preparation states for output. At this step the following formulae are defined: *The condition memory formula* $\varphi_{cmem}(\gamma,t,t_1)$ is obtained by replacing all occurrences in $\varphi_{cond}(\gamma,t,t_1)$ of $holds(state(\gamma,t'),p)$ by $holds(state(\gamma,t_1), memory(t',p))$. $\varphi_{cmem}(\gamma,t,t_1)$ contains a history of events, between the time point t, when $\varphi_p(\gamma,t)$ is true and the time point t_1, when the formula $\varphi_{cond}(\gamma,t,t_1)$ becomes true. Again by Lemma 1 $\varphi_{cmem}(\gamma,t,t_1)$ is equivalent to the formula $holds(state(\gamma,t_1),q_{cond}(t,t_1))$, where $q_{cond}(t,t_1)$ is called *the normalized condition state formula for* $\varphi_{cmem}(\gamma,t,t_1)$, and $q_{cond}(t)$ is the state formula $\forall u'$ $[present\_time(u') \rightarrow q_{cond}(t,u')]$. The state formula constructed by Lemma 1 for the preparation formula $\varphi_{prep}(\gamma,t_1)$ is called *the (normalized) preparation state formula* and denoted by $q_{prep}(t_1)$. Moreover, q_{prep} is the state formula $\forall u'$ $[present\_time(u') \rightarrow q_{prep}(u')]$. The formula $\varphi_{cprep}(\gamma,t_1)$ of the form $holds(state(\gamma,t_1),\forall u1 > t[q_{cond}(t,u1) \rightarrow q_{prep}(u1)])$ is called *the conditional preparation formula* for $\varphi_f(\gamma,t)$. The state formula $\forall u1 > t[q_{cond}(t,u1) \rightarrow q_{prep}(u1)]$ is called *the normalized conditional preparation state formula* for $\varphi_{cprep}(\gamma,t)$ and denoted by $q_{cprep}(t)$. Moreover, q_{cprep} is the formula $\forall u'$ $[present\_time(u') \rightarrow q_{cprep}(u')]$.

   The general executable rules that form basis for executable properties are defined as follows:

   $\forall t'\ holds(state(\gamma,t'),p) \Rightarrow holds(state(\gamma,t'),memory(t',p) \wedge stimulus\_reaction(p))$

   $\forall t'',t'\ holds(state(\gamma,t''),memory(t',p)) \Rightarrow holds(state(\gamma,t''+1),memory(t',p))$

   $\forall t'\ holds(state(\gamma,t'),q_{mem}) \Rightarrow holds(state(\gamma,t'),q_{cprep})$

   $\forall t',t\ holds(state(\gamma,t'),q_{cprep} \wedge q_{cond}(t) \wedge \wedge_p stimulus\_reaction(p)) \Rightarrow holds(state(\gamma,t'),q_{prep})$

   $\forall t'\ holds(state(\gamma,t'),stimulus\_reaction(p) \wedge \neg preparation\_for(output(t'+c,a))) \Rightarrow holds(state(\gamma,t'+1),stimulus\_reaction(p))$

   $\forall t'\ holds(state(\gamma,t'),preparation\_for(output(t'+c,a)) \wedge \neg output(a)) \Rightarrow holds(state(\gamma,t'+1),preparation\_for(output(t'+c,a)))$

   $\forall t'\ holds(state(\gamma,t'),present\_time(t') \wedge \forall u'[present\_time(u') \rightarrow preparation\_for(output(u'+c),a)]) \rightarrow holds(state(\gamma,t'),preparation\_for(output(t'+c),a))$

The auxiliary functions $stimulus\_reaction(a)$ are used for reactivation of component preparation states for generating recurring actions or communications.

For the property $DP7(A1, A2)$:

$$\forall t'[holds(state(\gamma, t', input(A2)), observation\_result\_to\_for(A2, IC))$$
$$\Rightarrow holds(state(\gamma, t', internal(A2)),$$
$$memory(t', observation\_result\_to\_for(A2, IC)) \wedge$$
$$stimulus\_reaction(observation\_result\_to\_for(A2, IC))]])$$

$$\forall t'' \; holds(state(\gamma, t'', internal(A2)),$$
$$memory(t', observation\_result\_to\_for(A2, IC))) \Rightarrow$$
$$holds(state(\gamma, t'' + 1, internal(A2)),$$
$$memory(t', observation\_result\_to\_for(A2, IC)))$$

$$\forall t' \; holds(state(\gamma, t'), \forall u''[present\_time(u'') \rightarrow$$
$$\exists u2[memory(u2, communicated\_from\_to(A1, A2, request, IC))]]) \Rightarrow$$
$$holds(state(\gamma, t'), \forall u'''[present\_time(u''') \rightarrow [\forall u1 > u'''$$
$$[memory(u1, observation\_result\_to\_for(A2, IC)) \rightarrow$$
$$preparation\_for(output(u1 + c,$$
$$communicated\_from\_to(A2, A1, inform, IC)))]]])$$

$$\forall t', tholds(state(\gamma, t'), [\forall u'''[present\_time(u''') \rightarrow [\forall u1 > u'''$$
$$[memory(u1, observation\_result\_to\_for(A2, IC))) \rightarrow$$
$$preparation\_for(output(u1 + c,$$
$$communicated\_from\_to(A2, A1, inform, IC)))]]]$$
$$\wedge \forall u''[present\_time(u'') \rightarrow$$
$$memory(u'', observation\_result\_to\_for(A2, IC)))] \wedge$$
$$stimulus\_reaction(observation\_result\_to\_for(A2, IC)))]) \Rightarrow$$
$$holds(state(\gamma, t', internal(A2)), \forall u1[present\_time(u1) \rightarrow$$
$$preparation\_for(output(u1 + c,$$
$$communicated\_from\_to(A2, A1, inform, IC))))])$$

$$\forall t' \; holds(state(\gamma, t'), stimulus\_reaction(observation\_result\_to\_for(A2, IC)) \wedge$$
$$not(preparation\_for(output(t' + c,$$
$$communicated\_from\_to(A2, A1, inform, IC)))]) \Rightarrow$$
$$holds(state(\gamma, t' + 1), stimulus\_reaction(observation\_result\_to\_for(A2, IC)))$$

$$\forall t' \; holds(state(\gamma, t', internal(A2)),$$

$[preparation\_for(output(t' + c, observation\_result\_to\_for(A2, IC)))$
$\wedge\ not(output(observation\_result\_to\_for(A2, IC)))] \Rightarrow$
$holds(state(\gamma, t' + 1, internal(A2)),$
$preparation\_for(output(t' + c, observation\_result\_to\_for(A2, IC))))$

3. Specify executable properties, which describe the transition from preparation states to the corresponding output states.

   The preparation state $preparation\_for(output(t1 + c, a))$ is followed by the output state, created at the time point t_{1+c}. The general executable rule is the following:

   $\forall t'\ holds(state(\gamma, t'), preparation\_for(output(t' + c, a))) \Rightarrow holds(state(\gamma, t' + c), output(a))$

   For the property $DP7(A1, A2)$:

   $\forall t'\ holds(state(\gamma, t', internal(A2)),$
   $preparation\_for(output(t' + c,$
   $communicated\_from\_to(A2, A1, inform, IC)))) \Rightarrow$
   $holds(state(\gamma, t' + c, output(A2)),$
   $output(communicated\_from\_to(A2, A1, inform, IC)))$

To automate the proposed procedure the software tool was developed in $Java^{TM}$. The transformation algorithm searches in the input file for the standard predicate names and the predefined structures, then performs string transformations that correspond precisely to the described steps of the translation procedure, and adds executable rules to the output specification file.

11.6.3 Abstraction of executable specifications

Sometimes (executable) specifications of multi-agent systems may be very detailed, with opaque global dynamics. To establish higher level dynamic properties of such systems, abstraction of specifications can be performed. In particular, internal dynamics of agents described by executable cognitive specifications may be abstracted to behavioural (or interaction) specifications of agents as shown in [399]. To express properties of behavioural and cognitive specifications past and past-present statements are used.

Definition 11.3. (Past-Present Statement) A past-present statement (abbreviated as a pp-statement) is a statement φ of the form $B \Leftrightarrow H$, where the formula B, called the body and denoted by $body(\varphi)$, is a past statement for t, and H, called the head and denoted by $head(\varphi)$, is a statement of the form $holds(state(\gamma, t), p)$ for some state property p.

It is assumed that each output state of an agent A specified by an atom $holds(state(\gamma, t), \psi)$ is generated based on some input and internal agent's dynamics that can be specified by a set of formulae over $\varphi(\gamma, t) \Rightarrow holds(state(\gamma, t), \psi)$ with φ a past statement over $InOnt(A) \cup IntOnt(A)$. Furthermore, a completion can be made (similar to Clark's completion in logic programming) that combines all statements $[\varphi_1(\gamma, t) \Rightarrow holds(state(\gamma, t), \psi), \varphi_2(\gamma, t) \Rightarrow holds(state(\gamma, t), \psi), ..., \varphi_n(\gamma, t) \Rightarrow holds(state(\gamma, t), \psi)]$ with the same consequent in the specification, into one past-present-statement $\varphi_1(\gamma, t) \vee \varphi_2(\gamma, t) \vee ... \vee \varphi_n(\gamma, t) \Leftrightarrow holds(state(\gamma, t), \psi)$. Sometimes this statement is called *the definition* of $holds(state(\gamma, t), \psi)$.

Furthermore, the procedure is applicable only to cognitive specifications that can be stratified.

Definition 11.4. (Stratification of a Specification) An agent specification Π is stratified if there is a partition $\Pi = \Pi_1 \cup ... \cup \Pi_n$ into disjoint subsets such that the following condition holds: for $i > 1$: if a subformula $holds(state(\gamma, t), \varphi)$ occurs in a body of a statement in Π_i, then it has a definition within $\cup_{j \leq i} \Pi_j$.

The notation $\varphi[holds_1, ..., holds_n]$ is used to denote a formula φ with $holds_1$, ..., $holds_n$ as its atomic subformulae.

The rough idea behind the procedure is as follows. Suppose for a certain cognitive state property the pp-specification $B \Leftrightarrow holds(state(\gamma, t), p)$ is available; here the formula B is a past statement for t. Moreover, suppose that in B only two atoms of the form $holds(state(\gamma, t1), p1)$ and $holds(state(\gamma, t2), p2)$ occur, whereas as part of the cognitive specification also specifications $B1 \Leftrightarrow holds(state(\gamma, t1), p1)$ and $B2 \Leftrightarrow holds(state(\gamma, t2), p2)$ are available. Then, within B the atoms can be replaced (by substitution) by the formula $B1$ and $B2$. This results in a

$$B[B1/holds(state(\gamma, t1), p1), B2/holds(state(\gamma, t2), p2)] \Leftrightarrow holds(state(\gamma, t), p)$$

which again is a pp-specification. Here for any formula C the expression $C[x/y]$ denotes the formula C transformed by substituting x for y. Such a substitution corresponds to an abstraction step. For the general case the procedure includes a sequence of abstraction steps; the last step produces a behavioural specification that corresponds to a cognitive specification.

Let us describe and illustrate the procedure for a simple executable pp-specification that corresponds to the property $DP7(A1, A2)$ considered in Section 11.6.2:

CP1(A1, A2) (memory state generation and persistence)

$holds(state(\gamma, t1, internal(A2)),$
$memory(t2, communicated\_from\_to(A1, A2, request, IC))) \Leftrightarrow$

$\exists t2\ t2 < t1\ \&\ holds(state(\gamma, t2, input(A2)),$
$communicated\_from\_to(A1, A2, request, IC))$

CP2(A1, A2) (conclusion generation)

$holds(state(\gamma, t3, output(A2)), communicated\_from\_to(A2, A1, inform, IC)) \Leftrightarrow$

$\exists t4, t5\ t4 < t3\ \&\ t5 < t4\ \&\ holds(state(\gamma, t4, internal(A2)),$
$memory(t5, communicated\_from\_to(A1, A2, request, IC)))\&$
$holds(state(\gamma, t4, input(A2)), observation\_result\_to\_for(A2, IC))$

To obtain an abstracted specification for a specification X the following sequence of steps is followed:

1. Enforce temporal completion on X.
2. Stratify X:

   a. Define the set of formulae of the first stratum ($h = 1$) as:
   $\{\varphi_i : holds(state(\gamma, t), a_i) \leftrightarrow \psi i_p(holds_1, ..., holds_m) \in X | \forall k\ m \geq k \geq 1\ \ holds_k$ is expressed using $InOnt\}$;

   proceed with $h = 2$.

   In the considered example $CP1(A1, A2)$ belongs to the first stratum.

   b. The set of formulae for stratum h is identified as
   $\{\varphi_i : holds(state(\gamma, t), a_i) \leftrightarrow \psi i_p(holds_1, ..., holds_m) \in X | \forall k\ m \geq k \geq 1\ \exists l$

   $l < h\ \exists \psi \in STRATUM(X, l)\ AND\ head(\psi) = holds_k\ AND\ \exists j\ m \geq j \geq 1\ \exists \xi \in STRATUM(X, h-1)\ AND\ head(\xi) = holds_j\}$;

   proceed with $h = h + 1$.

   In the considered example $CP2(A1, A2)$ belongs to the stratum 2.

   c. Until a formula of X exists not allocated to a stratum, perform 2b.

3. Replace each formula of the highest stratum n $\varphi_i : holds(state(\gamma, t), a_i)$ $\leftrightarrow \psi i_p(holds_1, ..., holds_m)$ by $\varphi_I \delta$ with renaming of temporal variables if required, where $\delta = \{holds_k \backslash body(\varphi_k)$ such that $\varphi_k \in X$ and $head(\varphi_k) = holds_k\}$. Further, remove all formulae $\{\varphi \in STRATUM(X, n-1) | \exists \psi \in STRATUM(X, n)$ $AND\ head(\varphi)$ is a subformula of the $body(\varphi)\}$).

   In the considered example the atom $holds(state(\gamma, t4, internal(A2)), memory(t5, communicated\_from\_to($ $A1, A2, request, IC)))$ in $CP2$ is replaced by its definition given by $CP1$:

   $BP1 : holds(state(\gamma, t3, output(A2)),$
   $communicated\_from\_to(A2, A1, inform, IC))$

$\Leftrightarrow \exists t4, t5\ t4 < t3\ \&\ t5 < t4 \&\ holds(state(\gamma, t5, input(A2)),$
$communicated\_from\_to(A1, A2, request, IC)) \&\ holds(state(\gamma, t4, input(A2)),$
$observation\_result\_to\_for(A2, IC))$

Furthermore, both $CP1$ and $CP2$ are removed from the specification. Thus, the obtained property is a behavioural specification expressed using $InOnt$ and $OutOnt$ only that corresponds to the considered cognitive specification.

4. Append the formulae of the stratum n to the stratum $n-1$, which now becomes the highest stratum (i.e, $n = n-1$).

   For the example, BP1 becomes the only property that belongs to the stratum 1.

5. Until $n > 1$, perform steps 3 and 4.

The algorithm has been implemented in $Java^{TM}$. The worst case time complexity is $O(|X|^2)$. The representation of a higher level specification Φ is more compact than of the corresponding lower level specification Π. First, only $IntOnt$ is used to specify the formulae of Φ, whereas $InOnt \cup OutOnt \cup IntOnt$ is used to specify the formulae of Π. Furthermore, only a subset of the temporal variables from Π is used in Φ, more specifically, the set of temporal variables from

$\{body(\varphi_i) | \varphi_i \in \Pi\} \cup \{head(\varphi_i) | \varphi_i \in \Pi\ AND\ head(\varphi_i)$ is expressed over $InteractOnt\}$.

11.7 Verification of Specifications of Multi-Agent Systems in TTL

In this Section two verification techniques of specifications of multi-agent systems are considered. In Section 11.7.1 a verification approach of TTL specifications by model checking is discussed. Checking of TTL properties on a limited set of traces obtained empirically or by simulation is considered in Section 11.7.2.

11.7.1 Verification of interlevel relations in TTL specifications by model checking

The dynamic properties of a component of a higher aggregation level can be logically related by an interlevel relation to dynamic properties of components of an adjacent lower aggregation level. This interlevel relation takes the form that a number of properties of the lower level logically entail the properties of the higher level component.

Identifying interlevel relations is usually achieved by applying informal or semi-formal early requirements engineering techniques; e.g., i^* [120] and SADT [300]. To formally prove that the identified interlevel relations are indeed correct, model checking techniques [100, 310] may be of use. The idea is that the lower level properties in an interlevel relation are used as a system specification, whereas the higher level properties are checked for this system specification. However, model checking techniques are only suitable for systems specified as finite-state concurrent systems. To apply model checking techniques it is needed to transform an original behavioural specification of the lower aggregation level into a model based on a finite state transition system. To obtain this, as a first step a behavioural description for the lower aggregation level is replaced by one in executable temporal format using the procedure described in Section 11.6.2. After that, using an automated procedure an executable temporal specification is translated into a general finite state transition system format that consists of standard transition rules. Such a representation can be easily translated into an input format of one of the existing model checkers. To translate an executable specification into the finite state transition system format, for each rule from the executable specification the corresponding transition rule should be created. For translation the atom *present_time* is used, which is evaluated to true only in a state for the current time point. For example, consider the translation of the memory state creation and persistence rules given in Table 11.3. The translation of other rules is provided in [398].

Table 11.3 Translation of the memory state creation and persistence rules into the corresponding finite state transition rules

Rule from the executable specification	Corresponding transition rules
Memory state creation rule	
$\forall t'\ holds(state(\gamma,t'),p) \Rightarrow$ $holds(state(\gamma,t'),memory(t',p))$	$present\_time(t) \wedge p \longrightarrow memory(t,p)$
Memory persistence rule	
$\forall t''\, holds(state(\gamma,t''),memory(t',p)) \Rightarrow$ $holds(state(\gamma,t''+1),memory(t',p))$	$memory(t,p) \longrightarrow memory(t,p)$

The executable properties obtained in Section 11.6.2 for the property $DP7(A1,A2)$ from the running example were translated into the transition rules as follows:

$$present\_time(t) \wedge communicated\_from\_to(A,B,request,IC) \longrightarrow$$
$$memory(t,communicated\_from\_to(A,B,request,IC))$$

$$present\_time(t) \wedge observation\_result\_to\_for(B,IC) \longrightarrow$$
$$memory(t,observation\_result\_to\_for(B,IC)) \wedge$$
$$stimulus\_reaction(observation\_result\_to\_for(B,IC))$$

$$memory(t, communicated\_from\_to(A, B, request, IC)) \longrightarrow$$
$$memory(t, communicated\_from\_to(A, B, request, IC))$$

$$memory(t, observed(observation\_result\_to\_for(B, IC)) \longrightarrow$$
$$memory(t, observed(observation\_result\_to\_for(B, IC))$$

$$present\_time(t) \wedge$$
$$\exists u2 \leq t\ memory(u2, communicated(request\_from\_to\_for(A, B, IC))) \longrightarrow$$
$$conditional\_preparation\_for($$
$$output(communicated\_from\_to(B, A, inform, IC)))$$

$$present\_time(t) \wedge$$
$$conditional\_preparation\_for($$
$$output(communicated\_from\_to(B, A, inform, IC))) \wedge$$
$$memory(t, observed(observation\_result\_to\_for(B, IC))) \wedge$$
$$stimulus\_reaction(observed(observation\_result\_to\_for(B, IC))) \longrightarrow$$
$$preparation\_for(output(t + c, communicated\_from\_to(B, A, inform, IC)))$$

$$present\_time(t) \wedge$$
$$stimulus\_reaction(observed(observation\_result\_to\_for(B, IC))) \wedge$$
$$not(preparation\_for($$
$$output(t + c, communicated\_from\_to(B, A, inform, IC))))$$
$$\longrightarrow stimulus\_reaction(observed(observation\_result\_to\_for(B, IC)))$$

$$preparation\_for(output(t + c, communicated\_from\_to(B, A, inform, IC))) \wedge$$
$$not(output(communicated\_from\_to(B, A, inform, IC))) \longrightarrow$$
$$preparation\_for(output(t + c, communicated\_from\_to(B, A, inform, IC)))$$

$$preparation\_for(output(t + c, communicated\_from\_to(B, A, inform, IC))) \wedge$$
$$present\_time(t + c - 1) \longrightarrow$$
$$output(communicated\_from\_to(B, A, inform, IC))$$

The obtained general representation for a finite state transition system was used further as a model for the model checker SMV [310]. SMV was used to perform the automatic verification of relationships between dynamic properties of components of different aggregation levels. For this purpose a procedure was developed for translating the general description of a transition system into the input format of the SMV model checking tool. For the description of the translation procedure and the complete SMV specification for the considered example we refer to [398].

One of the possible dynamic properties of the higher aggregation level that can be verified against the generated SMV specification is formulated and formalized in CTL as follows:

GP (Concluding effectiveness): If at some point in time environmental component E generates all the correct relevant information, then later agent C will receive a correct conclusion.

AG ($E\_output\_observed\_provide\_result\_from\_to\_E\_A\_info$ &
$E\_output\_observed\_provide\_result\_from\_to\_E\_B\_info$

→ **AF** $input\_C\_communicated\_send\_from\_to\_A\_C\_info$),

where **A** is a path quantifier defined in CTL, meaning "for all computational paths", **G** and **F** are temporal quantifiers that correspond to "globally" and "eventually" respectively.

The automatic verification by the SMV model checker confirmed that this property holds with respect to the considered model of the multi-agent system as specified at the lower level.

11.7.2 Verification of Traces in TTL

This section introduces a technique for verification of TTL specifications. Using this technique TTL properties are checked upon a limited set of traces. On the one hand, this set can be obtained by performing simulation of particular scenarios based on the TTL specification. In this case only a relevant subset of all possible traces is considered for the analysis. On the other hand, a set of traces can be obtained by formalising empirical data. Then, both verification of TTL properties on these traces and validation of TTL specifications by empirical data can be performed. For this type of verification a dedicated algorithm and the software tool TTL Checker have been developed [80] (see Figure 11.5) [1] .

As an input for this analysis technique either a simulation or a formalized empirical trace(s) is/are provided. A trace is represented by a finite number of state atoms, changing their values over time a finite number of times, i.e., complies with the finite variability property defined in Section 11.3. The verification algorithm is a backtracking algorithm that systematically considers all possible instantiations of variables in the TTL formula under verification. However, not for all quantified variables in the formula the same backtracking procedure is used. Backtracking over variables occurring in *holds* predicates is replaced by backtracking over values occurring in the corresponding *holds* atoms in traces under consideration. Since there are a finite number of such state atoms in the traces, iterating over them often will be more efficient than iterating over the whole range of the variables occurring in the holds atoms.

[1] The TTL Checker tool can be downloaded at http://www.few.vu.nl/ wai/TTL/

As time plays an important role in TTL-formulae, special attention is given to continuous and discrete time range variables. Because of the finite variability property, it is possible to partition the time range into a minimum set of intervals within which all atoms occurring in the property are constant in all traces. Quantification over continuous or discrete time variables is replaced by quantification over this finite set of time intervals.

In order to increase the efficiency of verification, the TTL formula that needs to be checked is compiled into a Prolog clause. Compilation is obtained by mapping conjunctions, disjunctions and negations of TTL formulae to their Prolog equivalents, and by transforming universal quantification into existential quantification. Thereafter, if this Prolog clause succeeds, the corresponding TTL formula holds with respect to all traces under consideration.

The complexity of the algorithm has an upper bound in the order of the product of the sizes of the ranges of all quantified variables. However, if a variable occurs in a holds predicate, the contribution of that variable is no longer its range size, but the number of times that the holds atom pattern occurs (with different instantiations) in trace(s) under consideration. The contribution of an isolated time variable is the number of time intervals into which the traces under consideration are divided.

The specific optimisations discussed above make it possible to check realistic dynamic properties with reasonable performance. To illustrate this technique the specification of the co-operative information gathering multi-agent system from Section 11.4 was instantiated for the case, when agents A and B collect and combine information about orthogonal projections of a three-dimensional shape: A collects information about the side view and B collects information about the bottom view. For example, if A observes a triangle and B observes a circle, then the shape is a cone. Using the simulation software environment LeadsTo [81] a number of simulation traces were generated and loaded into the TTL Checker. Then, a number of TTL properties were checked automatically on the traces, among which:

P1 (Successfulness of the cone determination)

$\forall \gamma\ \exists t\ \exists V : COMPONENT\ holds(state(\gamma, t, input(C)),$
$communicated\_from\_to(V, C, inform, conclusion(cone)))$

P2 (Successfulness of the projection acquisition for a cone)

$\forall \gamma\ \forall t1, t2\ holds(state(\gamma, t1, input(A)),$
$observation\_result\_to\_for(A, side\_view(triangle)))\&$
$holds(state(\gamma, t2, input(B)), observation\_result\_to\_for(B, bottom\_view(circle)))$

Checking the property P2 took 0.46 sec. on a regular PC. With the increase of the number of traces with similar complexity as the first one, the verification time grows linearly: for 3 traces - 1.3 sec., for 5 traces - 2.25 sec. However, the verification time is polynomial in the number of isolated time range variables occurring in the formula under verification.

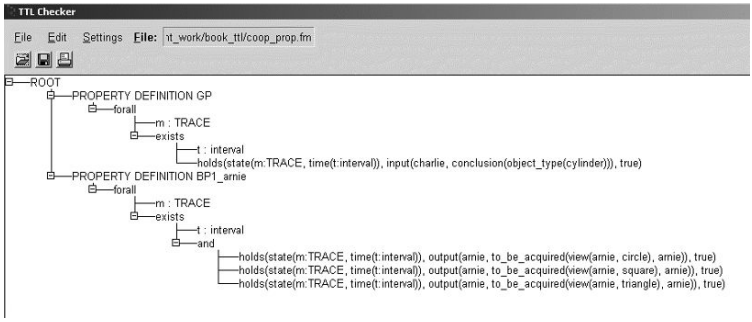

Fig. 11.5 Screenshot from the TTL Checker Tool

11.8 Conclusions

This chapter presents the predicate logical Temporal Trace Language (TTL) for formal specification and analysis of dynamic properties. TTL allows the possibility of explicit reference to time points and time durations, which enables modelling of the dynamics of continuous real-time phenomena. Although the language has a logical foundation, it supports the specification of both qualitative and quantitative aspects of a system, and subsumes specification languages based on differential equations.

Sometimes dynamical systems that combine both quantitative and qualitative aspects are called hybrid systems [133]. In contrast to many studies on hybrid systems in computer science, in which a state of a system is described by assignment of values to variables, in the proposed approach a state of a system is defined by (composite) objects using a rich ontological basis (i.e., typed constants, variables, functions and predicates). This provides better possibilities for conceptualizing and formalizing different kinds of systems (including those from natural domains). Furthermore, by applying numerical approximation methods for continuous behaviour of a system, variables in a generated model become discrete and are treated in the same manner as finite-state transition system variables. Therefore, so-called control points [297], at which values of continuous variables are checked and changes in a system's functioning mode are made, are not needed.

Furthermore, more specialised languages can be defined as a sublanguage of TTL. For simulation, the executable language LEADSTO has been developed [81]. For verification, decidable fragments of predicate logics and specialized languages with limited expressivity can be defined as sublanguages of TTL. TTL has similarities (as well as important conceptual distinctions) with (from) situation and event calculi. A proper subclass of TTL formulae can be directly translated into formulae of temporal logics (e.g., LTL and CTL).

In this chapter an automatically supported technique for verifying TTL properties on a limited set of simulation or empirical traces was described. Furthermore, it was shown how model checking techniques can be used for verification of TTL

specifications. To enable model checking, a model should be provided in the form of a finite state transition system. In this chapter it was shown how a TTL specification that comprises formulae in the executable normal form can be automatically translated into a finite state transition system. Using such an approach relations between dynamic properties of adjacent aggregation levels of a multi-agent system can be checked automatically, as also demonstrated in this chapter. The proposed approach has similarities with compositional reasoning and verification techniques [238] in the way how it handles complex dynamics of a system. Compositional reasoning approaches developed in the area of software engineering are based on one common idea that the analysis of global properties of a software system can be reduced to the analysis of local properties of system components. More specifically, the problem of satisfaction of global properties of a complex software system can be reduced to two (easier) problems: (i) identifying and justifying relations between global properties of the system and local properties of its components (parts); (ii) verifying local properties of system components with respect to components specifications.

In [338] formal methods for the analysis of hardware specifications expressed in the language PSL (an extension of the standard temporal logics LTL and CTL), are described. By means of the suggested property assurance technique supported by a tool, different global system properties (e.g., consistency) can be verified on specifications and in such a way the correctness of specifications can be established. The verification is based on bounded model checking techniques. Besides the specification language, an essential difference between this analysis method and the approach described in this chapter is that the latter provides means for the multi-level (or compositional) representation and verification of properties in specifications. This allows system modelling at a necessary level of abstraction and the reduction of the complexity of verification of system dynamics.

Similar differences can be identified in comparison with the approach proposed in [186]. This approach allows semi-automatic formalization of informal graphical specifications of multi-agent systems with the subsequent verification of dynamic properties using model checking techniques. Formalized specifications comprise descriptions of classes that describe components of a multi-agent system and relations between them, constraints over these components, assertions and possibilities. Although the first-order temporal logic that is used for formalizing these specifications is expressive enough to define complex temporal relations, it is does not provide the complete expressivity allowed by TTL (e.g., arithmetical operations, references to multiple traces in the same formula). Furthermore, although such specifications can be built and analyzed in parts, the idea of compositional verification, central in our approach, is not elaborated in this approach.

Compositional verification may be used for analysis of dynamics of large socio-technical systems (e.g., in the area of incident management). Such systems are characterized by a large complexity of internal dynamics of and interaction among diverse types of agents, including human and artificial intelligent agents (e.g., ambient devices). It is expected that in the future the complexity of such systems will increase considerably with a further development and implementation of ambient

intelligence technologies. Formal analysis of such systems presents a big conceptual and computational challenge for existing verification tools in the area of multi-agent systems. To enable effective and efficient analysis of systems of such type, new methods based on appropriate (dynamic) abstraction mechanisms need to be developed. For this the idea of compositional verification may serve as the starting point. Further, findings from the area of nonlinear system analysis, control theory and complex systems in general could be used.

Finally, TTL and the related analysis techniques proved their value in a number of research projects in such disciplines as artificial intelligence, cognitive science, biology, and social science. In particular, the analysis of continuous models (i.e., based on differential equations) is illustrated by the case study on trace conditioning considered in [83]. In [79] TTL is used for modelling and analysis of adaptive agent behaviour specified by complex temporal relations. The use of arithmetical operations in TTL to perform statistical analysis is illustrated by a case study from the criminology [78]. More examples of applications of TTL are described in [80].

Chapter 12

Assurance of Agent Systems: What Role Should Formal Verification Play?

M. Winikoff

Abstract In this chapter we consider the broader issue of gaining assurance that an agent system will behave appropriately when it is deployed. We ask to what extent this problem is addressed by existing research into formal verification. We identify a range of issues with existing work which leads us to conclude that, broadly speaking, verification approaches on their own are too narrowly focussed. We argue that a shift in direction is needed, and outline some possibilities for such a shift in direction.

M. Winikoff
University of Otago, New Zealand e-mail: michael.winikoff@otago.ac.nz

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2_12,

12.1 Introduction

A key issue in the deployment of multi-agent systems is being able to obtain assurance that the system will behave correctly. Before deploying a system, we need to convince those who will rely on the system (or those who will be liable/responsible if it fails) that the system will, in fact, work. Traditionally, this assurance is done through testing. However, it is generally accepted that adaptive systems exhibit a wider and more complex range of behaviours, making testing harder. For example, Munroe *et al.* [320, Section 3.7.2] say that:

> ... validation through extensive tests was mandatory ... However, the task proved challenging for several reasons. First, agent-based systems explore realms of behaviour outside people's expectations and often yield surprises ...

The lack of good ways of obtaining assurance that an agent system will behave correctly is seen to be a significant obstacle to industrial adoption. For instance, one of the four obstacles to widespread adoption of agent technology in manufacturing noted by Hall *et al.* [204] is

> Can the aggregate behavior of the agent-based system be guaranteed to meet all the system requirements?

Similarly, Pěchouček and Mařík [349, Page 413] note that[1]:

> Although the agent system performed very well in all the tests, to release the system for production would require testing all the steel recipes with all possible configurations of cooling boxes.

Before going further we need to briefly introduce some terminology. The term "**assurance**" refers to a process that aims to obtain confidence that (in this case) an agent system will behave appropriately. The term is used in a broad (and somewhat imprecise) sense. Where there is a clear specification (which is not always the case!) then we can use the two standard terms "verification" and "validation". Verification in this context refers to checking whether software meets its specification, and validation refers to checking whether the specification meets the user's requirements. There are a range of techniques for performing verification, including *formal verification* where mathematical reasoning techniques are used to formally establish a relationship between software and a formal mathematical specification. In some cases the formal specification is the whole software specification, in other cases it may be certain key properties, e.g. freedom from deadlock.

Unfortunately, the state of the art in assuring the correct behaviour of agent systems is still somewhat limited, and, perhaps surprisingly, there is not a large amount

[1] On the other hand, for another application they note that [349, Page 407]: "*Even though this negotiation process has not been theoretically proved for cycles' avoidance* [sic], *practical experiments have validated its operation*"

of existing and ongoing work. It is only relatively recently that testing began to receive significant attention from the agents community (e.g. [157,199,322,323,447]), and existing work on verification (typically using model checking; see section 12.2) is still rather limited in the size of systems that can be verified.

This chapter's key contribution is to look at the broad issue of obtaining assurance, and to ask to what extent this need is likely to be met by (the eventual outcomes of) current research on agent system verification. Put more concisely, if assuring correct behaviour is the problem, what role does formal verification have in the solution? We identify a number of issues which lead us to conclude that existing approaches are only *part* of the solution. We then suggest some alternative directions for investigation that aim to find out how to use existing work on testing and on verification as part of an assurance solution for agent systems.

This chapter doesn't question *whether* formal methods should be used, since it is clear that they have a role to play [436]. Rather, this chapter questions the traditional (narrow) view of verification typically assumed in agent verification research, and argues that, to be used as part of a broadly applicable approach for obtaining assurance, the scope of verification needs to be broadened, and certain assumptions need to be reconsidered. One possible broadly applicable approach for obtaining assurance is presented, and one key aspect (combining testing and proving) is discussed in more detail.

This chapter is structured as follows. We briefly review existing work on agent system verification (section 12.2), then introduce a simple case study (section 12.3) and prove that it is correct (section 12.4) before proceeding to discuss various issues (section 12.5) and considering how some of these issues manifest themselves in the case study (section 12.6). Section 12.7 proposes a new approach for assurance of agent systems, and section 12.8 elaborates on a key part of the approach: combining testing and proving techniques. We conclude in section 12.9.

12.2 Existing Work

In this section we briefly set the context by reviewing some existing work on verification of agent systems. Note that we focus in this section on verification, and do not describe in detail any of the work on testing agent systems which, roughly speaking, covers either support for running tests on agent systems (e.g. [157,199]), and/or ways of generating test cases (based on design models [447], ontologies [322], or using evolutionary techniques where soft-goals are rendered as quality functions that are used to guide the evolution of good test cases [323]). Note that this section is not intended to be a comprehensive survey, but merely to give a flavour of the recent work that has been done in the area. For a more comprehensive overview of the area we refer the reader to other chapters in this book.

All of the work discussed below considers, in some form, the problem of establishing beyond doubt (i.e. through proof or exhaustive analysis) that an agent pro-

gram[2] $\mathcal{P}$ meets a specification $\mathcal{S}$ (typically given in temporal logic, or an extension thereof). Much (but not all) of the existing work focues on model checking.

An early piece of work on verification was by Wooldridge *et al.* [441]. It introduced a simple imperative language (MABLE) and a specification language ($\mathcal{MORA}$, a simplified form of Wooldridge's $\mathcal{LORA}$). The $\mathcal{MORA}$ notation combines temporal logic and dynamic logic, and adds modalities for beliefs, desires and intentions. Each agent's program is translated into a Promela process, and claims about the system (in $\mathcal{MORA}$) are translated and checked using the SPIN model checker.

Bordini and colleagues [68, 73] extended this work by adopting as their agent programming language an agent-oriented programming language: AgentSpeak(F), a subset of Rao's AgentSpeak(L) limited to be finite. A subsequent paper [72] introduced a slicing algorithm which eliminates parts of the agent program that are not relevant to the property being checked, thus reducing the state space and the effort required to check the property (time required reduced by around 25% for the two examples they considered).

More recently, their work has shifted to support a wider range of agent-oriented programming languages, by translating to a common underlying abstract language, AIL, which is then translated to Java, and checked using a variant of the Java PathFinder tool[3] called AJPF [140]. Compared with using JPF, AJPF shows significant efficiency improvements [67]. However, the programs being verified are still "toy" programs, e.g. a six-line contract net example with three agents.

Another strand of work is that of Lomuscio and colleagues (e.g. [355]) which focuses on verifying so-called interpreted systems where, roughly speaking, an agent is modelled as a finite state machine. The specification logic is temporal logic augmented with a knowledge operator. Some recent work [168] is interesting in that it explicitly considers injecting faults, and looks at a range of specifications (in CTL) and what they mean in terms of the requirement on the system to be able to recover from faults. For example. a (CTL) specification of the form $AG(injected \rightarrow EF\phi)$ (where ϕ is the desired property) requires the system to have the possibility of eventually recovering from a fault, but doesn't require that it always do so, whereas $AG(injected \rightarrow AF\phi)$ requires that the system always (eventually) recover from an injected fault. While a promising direction in model checking research, this particular work is still somewhat preliminary: they model checked a single bit transmission protocol.

All of the work discussed so far in this section has used model checking. However, there is also some work that uses theorem proving. The work of Shapiro *et al.* [395] defined the Cognitive Agent Specification Language (CASL), and proved properties of ConGolog programs by translating to PVS, a typed higher-order logic with available tool support. They were able to verify properties of a simple meeting scheduler.

[2] In some work this is a single agent program, in other work this is a collection of agent programs, as well as a model of the system's environment

[3] http://javapathfinder.sourceforge.net/

More recent work includes that of Alechina *et al.* [6–8] which also uses tool-supported theorem proving. They use a simple language (SimpleAPL), and translate SimpleAPL programs (along with the starting state of the agent) into Propositional Dynamic Logic (PDL). Safety and liveness properties can then be verified using an existing PDL theorem prover. A key contribution of Alechina *et al.* is the ability to model the agent's execution strategy, and prove properties that may rely on a given execution strategy (e.g. interleaved vs. non-interleaved execution of plans). The approach appears to have only been applied to toy programs (single agent, a couple of plans, and propositional beliefs).

Finally, another example of theorem proving is the work of Mermet *et al.* [312] which proves correctness using proof schemas. The agents are specified using Goal Decomposition Trees which specify how each goal is achieved by its sub-goals, for example, by a sequence of sub-goals. Unlike model checking, showing that a robot on a grid works correctly does not depend on the size of the grid. However, the proofs appear to be done by hand (tool support is mentioned as future work).

12.3 Case Study: A Waste Disposal Robot

Our discussion of issues will be made concrete and illustrated using a case study. Our case study scenario is a simple model of a waste disposal robot, and is inspired by examples used in a range of previous papers (e.g. [68, 312, 323, 357, 379]). Note that certain aspects of the case study are done in a way that could be improved (for instance not having a separate sense action); this was done in order to better allow a range of issues to be illustrated using a single case study.

The world consists of a grid of locations $\mathbb{L} = \{0 \ldots \text{MAX}\} \times \{0 \ldots \text{MAX}\}$ (for some value of $MAX \in \mathbb{N}$), with typical element $\ell \in \mathbb{L}$. Each location contains an amount of rubbish which is specified by the model $m : \mathbb{L} \to \mathbb{N}$, that is, for a given location ℓ the amount of rubbish at that location is $m(\ell) \in \mathbb{N}$. A certain subset of the locations $b \subseteq \mathbb{L}$ has bins for the disposal of rubbish (and it is assumed that these locations are known to the robot).

A robot's state consists of its position $p \in \mathbb{L}$, the amount of rubbish it is holding $h \in \mathbb{N}$, and its view of the environment $v : \mathbb{L} \to \mathbb{N} \cup \{\bot\}$, where $v(\ell) = \bot$ denotes that the robot does not know anything about the location ℓ (we assume in what follows that any numerical condition on $v(\ell)$ such as $v(\ell) > 0$ has an implicit additional condition that $v(\ell) \neq \bot$). The robot also has a fixed capacity for carrying rubbish, $c > 0$.

The overall goal of the robot is to eliminate all rubbish: $\forall \ell : v(\ell) = 0$. In fact, we also want the robot to not be holding rubbish, and so the robot's goal $\mathcal{G}$ is $h = 0 \wedge \forall \ell . v(\ell) = 0$.

The actions available to the robot are specified in Figure 12.1 and comprise moving, picking up and dropping rubbish, and sensing how much rubbish is located

at its current location. Actions are specified using simultaneous assignment statements, rather than the more traditional post-conditions for two reasons: it is closer to an implementation, and it avoids needing to specify explicitly what things are left unchanged by an action.

Summary of model

$\mathbb{L} = \{0 \ldots MAX\} \times \{0 \ldots MAX\}$	locations	$p \in \mathbb{L}$	robot's position
$\ell \in \mathbb{L}$	location	$h \in \mathbb{N}$	rubbish held by robot
$m : \mathbb{L} \to \mathbb{N}$	environment model	$v : \mathbb{L} \to \mathbb{N} \cup \{\bot\}$	robot's (partial) view
$b \subseteq \mathbb{L}$	locations of bins	$c \in \mathbb{N}$	robot's carrying capacity

Actions

Action	Pre-condition	Effect (simultaneous assignment)
move(ℓ_{dest})	true	$p := \ell_{dest}$
pick()	$h < c$	$h := h + t$ $m(p) := m(p) - t$ $v(p) := v(p) - t$ where $t = \min(c - h, m(p))$
drop()	$p \in b$	$h := 0$
sense()	true	$v(p) := m(p)$

Robot Program

(Notation: *event* : *condition* ← *planbody*)

1a $+!clean : v(p) = \bot \leftarrow$ sense() ; $!clean$
1b $+!clean : h = 0 \wedge \forall \ell.v(\ell) = 0 \leftarrow$ stop (do nothing)
1c $+!clean : h = 0 \wedge v(p) = 0 \wedge \ell \in \mathbb{L} \wedge v(\ell) > 0 \leftarrow$ move(ℓ) ; $!clean$
1c' $+!clean : h = 0 \wedge v(p) = 0 \wedge \forall \ell'.v(\ell') \in \{0, \bot\} \wedge \ell \in \mathbb{L} \wedge v(\ell) = \bot \leftarrow$ move(ℓ) ; $!clean$
2 $+!clean : h = 0 \wedge v(p) > 0 \leftarrow$ pick() ; $!clean$
3 $+!clean : h > 0 \wedge p \notin b \wedge \ell \in b \leftarrow$ move(ℓ) ; $!clean$
4 $+!clean : h > 0 \wedge p \in b \leftarrow$ drop() ; $!clean$

Fig. 12.1 Summary of model, actions, and robot program

The robot's program could be specified in a wide range of notations, ranging from simple to quite complex. We use the AgentSpeak(L) notation [357] to specify the robot's program, although a simpler notation such as condition-action rules would have sufficed. The agent program captures the following cases (see Figure 12.1 for details).

1. If the robot is not carrying anything and there is no rubbish at its location, then explore[4]

[4] In fact, there are a few sub-cases here: if the robot does not know how much rubbish is at its current location it should perform a sense action, if the robot knows about the location of rubbish,

2. If the robot is not carrying any rubbish and there is rubbish at the current location, then pick it up
3. If the robot is carrying rubbish and it is not at a bin, then move to a bin
4. If the robot is carrying rubbish and is at a bin, then drop the rubbish

This basic robot can be extended and improved in various ways, such as improving its efficiency. One issue is that the robot picks up rubbish, and then proceeds to a bin immediately, even though it may be able to carry more rubbish. Our first optimisation ("opt1") is to have the robot continue to collect rubbish until it is full. This is done by modifying the condition $h = 0$ in plans 1c and 2 to $h < c$. A second efficiency-related optimisation is to modify the selection of locations in plans 1c and 3 to select a[5] *closest* location, rather than an arbitrary (random) location. We term this improvement "opt2".

This simple robot was implemented [6], including the two optimisations (opt1 and opt2) just discussed. In addition to implementing the actions and the robot's program, the implementation included:

- Error checking of a range of conditions (such as the robot going into an infinite loop, the pre-conditions of actions not being met, etc.) that should not occur.
- Tracking the (Manhattan[7]) distance that a robot travelled (in order to allow for the effects of the two optimisations to be measured).
- Random initialisation of the starting configuration.

Note that although the program in Figure 12.1 is given in an agent-oriented programming language, the implementation used for experimental purposes was actually done in a general purpose scripting language (namely Lua, `http://www.lua.org`).

Another elaboration that is possible is to add to the robot an amount f of fuel, and have the amount of fuel be decreased when the robot moves. The robot's behaviour would also need to include a way of refuelling when needed.

In our model the robot begins with knowledge of the locations of all the bins. This is not very realistic, and could be changed as follows. First, introduce a variable $b_r : \mathcal{P}(\ell)$ which is used to track the locations of bins known to the robot; initially $b_r = \emptyset$. Then plan 3 is split into two cases: if b_r is non-empty, then select $\ell \in b_r$; otherwise, if $b_r = \emptyset$, then instead select ℓ such that $v(\ell) = \bot$. The sense action also needs to be modified to update b_r (e.g. $b_r := b_r \cup (\{p\} \cap b)$).

Finally, the system only has a single agent and clearly it could be extended to have a collection of robots exploring the landscape. Perhaps the simplest way of

then it should go straight there, and if it knows that there is no rubbish anywhere then it should stop. Figure 12.1 includes these sub-cases.

[5] There may be more than one equally close location that satisfies the desired condition, e.g. being a bin, or having rubbish.

[6] Source code available on request.

[7] Where the distance between (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$.

doing this is just to have multiple robots roaming the landscapes, oblivious to each others' presence. However, effectiveness would be improved by having the robots share their knowledge of the environment by communicating in some way, either directly by messages, or indirectly through the environment.

12.4 Correctness Proof

We now briefly argue that the robot program given in Figure 12.1 meets the specification. Note that the following proof is informal. We do not give a formal proof for a number of reasons. Firstly, it can be argued that an informal proof of this sort is more representative of common practice than formal proofs. Secondly, for an agent program written in a "real" language, there typically do not exist tools that can assist with creating or checking formal proofs. Finally, we feel that expanding this section to be formal would not add much to this chapter, but would distract from the key points in the following sections.

The proof proceeds by defining a numerical measure of progress, and then arguing that the robot's program works to decrease it, ultimately reaching 0. Our metric, $\mathcal{M}$, is defined as follows. Suppose that a given starting configuration has at most N units of rubbish in a given location, that is, $\forall \ell : v(\ell) \leq N$. For each location ℓ we let $w(\ell) = v(\ell)$ if $v(\ell) \in \mathbb{N}$, else, if $v(\ell) = \bot$, we let $w(\ell) = N + 1$. Then $\mathcal{M}$ is just the sum over all locations of $w(\ell)$, with an additional term accounting for the rubbish that the robot is holding:

$$\mathcal{M} = \frac{h}{2} + \sum_{\ell \in \mathbb{L}} w(\ell)$$

Observe that $\mathcal{M} = 0$ exactly when the robot's goal, $\mathcal{G} \equiv h = 0 \wedge \forall \ell . v(\ell) = 0$, is satisfied.

We now prove that the robot's program works. Firstly, we argue that the condition on each action implies the action's pre-conditions. This is trivial: move and sense both have a true pre-condition. For pick we require that $h < c$, which follows from $h = 0$ (the condition of plan 2) and $c > 0$, and for drop we require that $p \in b$ which follows trivially from the condition of plan 4.

Secondly, we argue that the effects of the robot's actions are to monotonically decrease the metric $\mathcal{M}$. Certain actions have the effect of directly reducing $\mathcal{M}$. For pick we observe that $v(p)$ is reduced and h increased by the same amount, since $\mathcal{M}$ counts $h/2$ this yields a reduction. For drop we observe that the only change is a reduction in h. For sense we note that the action is only performed where $v(p) = \bot$, and this thus reduces the value of $w(p)$ from $N + 1$ to N or less.

The remaining action, move, does not affect $\mathcal{M}$, but whenever the robot moves, it creates a situation where the following action is not a move. Specifically, we have the following three cases:

1. If the robot moves as a result of plan 1c then it moves to a location such that $v(\ell) > 0$ and the resulting state has $h = 0$ (unchanged from the condition of plan 1c) and $v(p) > 0$ and hence the next plan to apply is plan 2 and a pick action will be done.
2. If the robot moves as a result of plan $1c'$ then it moves to a location such that $v(\ell) = \bot$ and the resulting state has $v(p) = \bot$, and so a sense action will be done (plan 1a).
3. Finally, if the robot moves as a result of plan 3, then the resulting state has $h > 0$ (unchanged from the condition of plan 3) and $p \in b$, and thus the next action to be performed is a drop (plan 4), or, if the robot doesn't know how much rubbish is at the bin's location, a sense followed by a drop.

Since a move does not increase $\mathcal{M}$, the robot cannot perform a sequence of movements, and all other actions do reduce $\mathcal{M}$, we have that the robot's actions progressively reduce $\mathcal{M}$. This allows us to conclude that the robot will eventually reach $\mathcal{M} = 0$ at which point its goal is achieved.

12.5 Issues

Suppose that we use model checking or a formal proof to show that the robot program ("$\mathcal{P}$") presented in the previous section meets the system's specification ("$\mathcal{S}$"). In this section we consider a range of issues associated with doing so.

Firstly, we consider issues to do with capturing the right specification $\mathcal{S}$. It turns out that in practice the notion of correctness isn't always that easy to capture formally, even for such a simple case study. We discuss this issue below in section 12.5.1.

However, even if we do manage to capture the right specification, an issue in verifying that program $\mathcal{P}$ meets specification $\mathcal{S}$ is that it only considers the program and a specification: it doesn't consider the broader context and such factors as human errors and hardware errors. Additionally, proofs tend to be abstract, and it turns out to be easy to have hidden implicit assumptions in models or proofs, which can be dangerous. We discuss this issue in section 12.5.2.

It is worth noting that whilst some of the second group of issues are generic, i.e. they apply to any software system, not just agent systems, the nature of the "broader context" is different for agent systems. Furthermore, the first group of issues is agent specific. For instance, since agent systems are often situated in failure-prone environments where success cannot be guaranteed, the form of the specification needs to change from requiring success, to only requiring success if success is actually possible.

12.5.1 Problems with Specifications

Typically specifications are expressed in temporal logic (or an extension thereof), and there are a range of standard properties that are specified and checked against (e.g. liveness, lack of deadlock). Indeed, there are libraries of commonly used specification patterns [153].

However, capturing the right notion of "correctness" is not always straightforward with agent systems, even for the very simple case study that we consider. For instance, one common pattern, which corresponds to a goal of achieving a desired property ϕ, is to require that ϕ eventually holds ($\Diamond\phi$). However, for agent systems which may be deployed in a failure-prone environment, there may be situations where failure cannot be avoided, and so $\Diamond\phi$ is too strong, and will not be provable. For example, part of the environment may be unreachable, due to blocked paths, or due to the robot not being able to hold enough fuel. A more appropriate specification is that the robot succeeds "where possible". However, specifying the "where possible" condition is not easy. Furthermore, it depends on the agent program: an agent that is able to plan ahead and realise that it needs to refuel before heading out to retrieve some rubbish will have different failure conditions to a robot that just heads out and refuels when it is close to running out [152].

In order to capture a suitable correctness condition we turn to *dynamic logic* [347], in which *action expressions* may be primitive actions a, sequences of actions $a_1;a_2$, zero or more iterations of an action expression a^*, or a choice of action expressions $a_1 + a_2$. Then $\langle A\rangle\phi$ is read as "after performing action expression A the property ϕ *may* hold"; and $[A]\phi$ is read as "after performing action expression A the property ϕ *must* hold". Now suppose that the desired goal is $\mathcal{G}$, that the actions available to the agent are $A = a_1, \ldots, a_n$, and that the set of all action sequences is A^*. We denote the robot's program (or, more precisely, its translation into dynamic logic) by $\mathcal{P}$, and any assumptions that are being made about the initial state are denoted by $\mathcal{I}$.

In order to capture the requirement that the robot must succeed we can write $\mathcal{I} \Rightarrow [\mathcal{P}]\mathcal{G}$, but, as discussed earlier, this is too strong. What we want to capture is that the robot is only required to succeed if, given the initial assumptions, success is actually possible. The notion that "success is possible" is formalised as the existence of a sequence of actions that realises the goal, thus the overall requirement is written as:

$$(\mathcal{I} \wedge \exists A \in A^*.[A]\mathcal{G}) \Rightarrow [\mathcal{P}]\mathcal{G}$$

This formalisation only requires the agent to succeed if success is possible, which is what we want. However, the definition of "success is possible" ($\exists A \in A^*.[A]\mathcal{G}$) is not quite right. In certain situations success may be theoretically possible but not practically possible. These situations occur when there exists a sequence of actions that succeeds (which meets the definition of "success is possible"), but where the information available does not allow one to select which actions to perform. For example, suppose there are two doors, exactly one of which hides a large prize, and

we have one chance to open a door and claim whatever lies behind it. Then there exists an action that claims the large prize, and hence the definition of "success is possible" is met. However, given the information available, we are not able to reliably select the correct action. We believe that this issue can be fixed by specifying a correct formalisation of "success is possible", but this involves introducing a representation for the information available, and is a diversion from the main point of this chapter.

Another issue with a specification of the form $\Diamond\phi$ is that "eventually" can be soon, or in a very long time, and in a real deployed system, knowing that the system will "eventually" achieve some desired state isn't enough: we need to know that "eventually" will arrive "reasonably soon". This issue is (somewhat) specific to agent systems. Consider a classical concurrent system, such as a printer spooler. In such systems efficiency is not typically an issue: if there are no deadlocks, then the system will complete spooling print jobs in a timely manner. However, an agent system such as a manufacturing scheduling system, or indeed, a robotic cleaner, may take too long to run, even if there are no deadlocks.

Unfortunately, "reasonably soon" can be relative to the problem instance: how long a cleaning robot could reasonable be expected to take depends on the amount and distribution of rubbish in the environment. Thus, defining the desired property of "reasonably soon ϕ" requires specifying how long an ideal robot would take to clean a given environment[8].

Although efficiency and performance issues are often dealt with through means other than formal verification, in some systems performance is critical. In these systems we often do need to provide strong guarantees about performance, and its variability, in a way that cannot be met by testing with sample cases. We return to this issue, in the context of the case study, in section 12.8.2.

In summary, getting the specification right for an agent system, so that it captures both what is actually needed (e.g. "reasonably soon" rather than "eventually") and also isn't too demanding ("succeeds where possible" rather than "always succeeds") is not easy, and specifications that are short and simple (e.g. $\Diamond\forall\ell.m(\ell) = 0$) become more complex when these issues are taken into account. However, if these issues

[8] To specify "reasonably soon" we first define the following notion of cost: given a sequence of actions A, its cost is denoted by $cost(A)$ (where the function $cost$ maps an action sequence to a natural number). This notion can be generalised to an action expression A_E (or program $\mathcal{P}$) by defining the cost of an action expression A_E executed in starting state S_0 as being the cost of the sequence of actions that is performed, that is, $cost(\mathcal{P})$ when the program is executed in starting state S_0 is defined as $cost(A)$ where A is the sequence of actions that the program performs when executed in S_0. We then define the most efficient action sequence $S_0 \in A^*$ as being a solution for the goal $\mathcal{G}$, with the additional condition that any other solution, A, has a higher (or equal) cost. We formalise this by firstly defining a solution $S \in A^*$ of a goal $\mathcal{G}$: $solution(S,\mathcal{G}) \iff (\mathcal{I} \wedge \exists A \in A^*.[A]\mathcal{G}) \Rightarrow [S]\mathcal{G}$ and then specifying the best solution S_0 as: $best(S_0,\mathcal{G}) \iff solution(S_0,\mathcal{G}) \wedge (\forall S \in A^*.solution(S,\mathcal{G}) \Rightarrow cost(S) \geq cost(S_0))$ We can now define a "reasonably good" solution as being one that is within some desired factor N of the best possible solution (clearly this is just one possible notion of "reasonably good"): $good(\mathcal{P},\mathcal{G},N) \iff \forall S_0 \in A^*.best(S_0,\mathcal{G}) \Rightarrow cost(S_0) * N \geq cost(\mathcal{P})$

are not taken into account, then a proof may not be possible (because the robot in fact cannot deal with all configurations of the environment) or may establish a result that looks nice ("it always eventually ...") but in fact is too weak to be practically useful.

Note that this discussion has focussed on the *form* or *structure* of the specification, e.g. using $\Diamond\phi$ as opposed to $(\mathcal{I} \wedge \exists A \in A^*.[A]\mathcal{G}) \Rightarrow [\mathcal{P}]\mathcal{G}$. A related, and well-known, issue is getting the *contents* of the specification right (i.e. the choice of *which* ϕ) [261].

It is also worth noting that the point of this subsection is not to argue that the correctness specification should be exactly as described, but to highlight that attempting to address the two key issues of using simple and natural specifications such as $\Diamond\phi$ results in a significantly more complex specification. Furthermore, and perhaps more importantly, the resulting more complex specification appears to be rather more challenging to verify (but we have not verified this yet).

12.5.2 Problems with Proofs

> Beware of bugs in the above code; I have only proved it correct, not tried it — D. Knuth[9]

As noted earlier, the enterprise of verification is concerned with showing that a program $\mathcal{P}$ meets its specification $\mathcal{S}$. The previous subsection discussed a number of issues with getting the specification $\mathcal{S}$ right. However, even if we can get the specification right, there are still two significant issues with proving that a program meets its specification:

1. Showing that a program meets its specification does not consider the wider context: despite a correct proof being given, issues may still arise due to human interaction, or physical interference. This motivates the argument (in section 12.7) that we need to *broaden* the scope of verification.
2. A proof may be abstract and may contain implicit (typically "obvious") assumptions that turn out to fail, making the proof wrong. Knuth's comment highlights that, in some cases, these "obvious" assumptions can be detected very easily by testing, which motivates the argument (in section 12.8) that we should look at ways to combine testing and proving techniques.

The first issue is that traditional approaches to verification consider only the program, ignoring other issues, such as user interfaces, human errors, and hardware faults. The assumption that seems to justify the narrow focus on programs and (formal) specifications seems to be that software errors — that is, differences between the (formal) specification and the implementation — are the key issue.

[9] http://www-cs-faculty.stanford.edu/~knuth/faq.html

However, although software errors clearly are significant, analysis of computer-related failures suggests that software errors are only a part, and possibly a rather small part, of the problem. For instance, an analysis[10] of computer-related deaths[11] [294, Chapter 9] found that, of roughly 1,100 deaths[12] that were caused by computer failures up to 1992[13] [295, Chapter 9, p300]:

> ... Over 90 percent of these deaths were caused by faulty human-computer interaction (often the result of poorly designed interfaces or of organizational failings as much as of mistakes by individuals). Physical faults such as electromagnetic interference were implicated in a further 4 percent of deaths, while the focus of Hoare's and Licklidder's warnings, software "bugs", caused no more than 3 percent, or thirty deaths: two from a radiation-therapy machine whose software control system contained design faults, and twenty-eight from faulty software in the Patriot antimissile system that caused a failed interception in the 1991 Gulf War.

Writing about these results, Jackson [250, Pages 86-87] notes that (emphasis added):

> ... coding errors were cited as causes only 3% of the time. **Problems with requirements and usability dwarf the problems of bugs in code, suggesting that the emphasis on coding** practices and tools, both in academia and industry, **may be mistaken**.

That is, the vast majority of the time (97%), computer-related deaths were not due to problems in coding, and would not have been caught by formal verification of implementation against specification. The key point here is that in order to cover more than 3% of MacKenzie's cases, the scope of the specification needs to be broadened to include social, organisational, and human aspects of the system.

We have already seen (in the previous section) that getting the specification to reflect the real need is particularly challenging for agent systems. When verifying agent systems, which are situated in an environment, it is important to capture this environment. Furthermore, for agent systems that support human activity, such as disaster response coordination [389], or space exploration [69], it is important to capture the human and organisational context of the system as part of a specification. In the next section we will discuss some examples of verification that considers human aspects. However, the bulk of the work on agent verification does not consider humans to be within the scope of the specification.

The second issue with proof is assumptions: all too often assumptions made are hidden, rather than made explicit. An excellent example, in a very simple setting, is binary search [250, Page 87] (footnotes and emphasis added):

[10] The original paper was published in *Science and Public Policy* in 1994, and was re-printed as Chapter 9 of [294]. A less detailed discussion of the results of the analysis appeared in Chapter 9 of [295].

[11] More precisely, it focussed on "computer-related accidental death", where "computer-related" indicates a careful analysis of whether the presence of computers was a causal factor in the death(s), and where "accidental" excludes deaths caused by military computer systems that are designed to kill (the analysis also, reluctantly, excludes any associated "collateral" civilian deaths).

[12] MacKenzie's analysis focussed on deaths, because deaths are clearly defined (unlike injury, which would include RSI), and because deaths are more likely to be reported.

[13] The analysis only considered data up to 1992; however, there does not appear to be any more recent analysis.

> Proof is not foolproof, however. When a bug was reported in his own code (part of the Sun Java library), Joshua Bloch found[14] that the binary search algorithm—proved correct many years before (by, amongst others Jon Bentley in his *Communications* column) and upon which a generation of programmers had relied—harbored a subtle flaw. The problem arose when the sum of the low and high bounds exceeded the largest representable integer[15]. Of course, the proof wasn't wrong in a technical sense; there was an **assumption** that no integer overflow would occur (which was reasonable when Bentley wrote his column, given that computer memories back then were not large enough to hold such a large array). In practice, however, **such assumptions will always pose a risk, as they are often hidden in the very tools we use to reason about systems** and we may not be aware of them until they are exposed.

In the next section we consider this issue in the context of our waste disposal robot case study and explore where implicit assumptions have been made.

12.6 Assumptions in the Waste Disposal Robot Case Study Revisited

Considering the earlier proof that the robot program meets its specification (section 12.4), it turns out, in fact, that the proof isn't quite right. It makes a number of assumptions without stating them. These assumptions may seem reasonable, but they may be false. Worse, because the proof is abstract it is easy to miss these assumptions (did *you* spot all of them?).

The first (implicit) assumption is that there are bins (i.e. that $b \neq \emptyset$). If this assumption is violated, then, unless there is no rubbish in the environment, the agent will fail: once it has located and picked up rubbish, it will attempt to use plan 3, and there is no way to select an ℓ such that $\ell \in b$ if $b = \emptyset$.

A related issue is the other places where the robot selects a location ℓ that satisfies some condition (in plans 1c and 1c'). Do we have a similar issue there? The precondition for plan 1c includes the condition $\neg\forall \ell.v(\ell) = 0$ which can be rewritten as $\exists \ell.v(\ell) \neq 0$. Thus, the precondition guarantees that there will exist at least one location ℓ such that $\ell > 0$ or $\ell = \bot$, so the "select ℓ such that ..." cannot fail. But actually this reasoning also relies on an assumption: that the value of $v(\ell)$ will be either $\bot$ or a natural number. If this assumption doesn't hold, then it is in fact possible for there to not be any possible selection for ℓ that satisfies the conditions of plan 1c (and this in fact did occur in testing — see below).

The second key assumption is that the robot's view, v, "mirrors" the real environment, m. More precisely, that $\forall \ell.v(\ell) = m(\ell) \vee v(\ell) = \bot$, that is, for each location, the robot's view is either unknown ($\bot$), or matches the environment. In our experimentation we considered a range of initial situations where this assumption was

[14] http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

[15] That is, code of the form `middle = (high + low) / 2`.

violated. However, an inconsistency between v and m can also arise *during* execution because of errors in sensing or acting (thinking you've picked up rubbish when in fact you haven't), and also because of exogenous activity (such as other robots cleaning rubbish, or pesky humans littering their environment).

We now explore a range of issues that can arise when, for some location ℓ, $m(\ell) \neq v(\ell)$ (and $v(\ell)$ is not $\perp$). It is worth emphasising that **all of these issues have been observed in the implementation.**

Case 1: One case is where $m(\ell) > 0$ but $v(\ell) = 0$. In this case the robot incorrectly believes that location ℓ is clean, and the robot will, in fact, never visit the location (since there is no need), and, if there are no other issues, will complete execution believing that it has successfully cleaned the world, whereas there is still rubbish at ℓ.

Case 2: Another case is where $m(\ell) = 0$ but $v(\ell) > 0$, i.e. the robot believes there is rubbish at ℓ, but in fact there isn't. In this case the robot will eventually (if no other errors intervene) arrive at ℓ and proceed to pick up the rubbish. The pick action changes $m(\ell)$ and $v(\ell)$ by t, which in this case is zero, i.e. the pick action has no effect. This means that the situation is not changed, and the next plan that is applicable remains plan 3, and the robot will again attempt to pick up the non-existent rubbish, leading to an infinite loop.

Note that, in fact, any situation where $v(\ell) > m(\ell)$ will end up with a loop, since (if no other errors intervene), the robot will pick rubbish, eventually reducing $m(\ell)$ to zero.

Case 3: We now turn to the situation where both $m(\ell)$ and $v(\ell)$ are greater than zero, but where $m(\ell) > v(\ell)$. A range of behaviours can result from this situation. We begin by observing that if $m(\ell)$ and $v(\ell)$ are both greater than 0, then pick action(s) will reduce both of them until eventually $v(\ell)$ is not greater than zero, resulting in one of the following sub-cases:

- If $v(\ell) = 0$ then we are left with the first scenario discussed above: the robot will incorrectly believe the location to be cleaned, and (if no other errors intervene), will eventually believe its goal to be completed, even though rubbish remains in the environment.
- If $v(\ell) < 0$ then we have an invalid value, i.e. the assumption that $v(\ell)$ is either a natural number or $\perp$ does not hold. The implementation actually uses -1 to represent $\perp$, which results in the following two sub-cases:
  - If $v(\ell) = -1$ then in fact the robot recovers: it interprets the -1 as being $\perp$ and performs a sense action which makes $v(\ell) = m(\ell)$, resolving the difference.
  - If $v(\ell) < -1$ (e.g. $v(\ell) = -2$) then plans 1a, 1b, 1c and 2 are not applicable. This leaves plans 3 and 4. If the location happens to be a bin and the robot is holding rubbish[16] then it will drop the rubbish, at which point *none* of

[16] Which will be the case, because a pick action is what reduced $v(\ell)$ to -2.

the plans are applicable and the robot halts with an error. If the location is not a bin (and the robot is holding rubbish) then it will move to a bin and drop the rubbish there. Once the rest of the grid is cleaned, the robot is left in a situation where the only applicable plan is 1c (since it is not done), but where the "select an ℓ such that ..." cannot be satisfied: there are no remaining locations for which $v(\ell) > 0$ or $v(\ell) = \perp$: the location that causes $\forall \ell.v(\ell) = 0$ to be false has a value of -2, which is neither of these cases.

To briefly summarise these cases we have the following possible behaviours:

1. Finishing, believing the task to be completed, even though there is rubbish in the world (case 1, where $v(\ell) = 0$ and $m(\ell) > 0$)
2. An infinite loop (case 2, where $v(\ell) > 0$ and $m(\ell) = 0$)
3. Recovering because -1 is used in the implementation to represent $\perp$ (case 3 for $v(\ell) = -1$)
4. Aborting because either no plan is applicable, or plan 1c is applicable but no suitable ℓ can be selected (case 3 for $v(\ell) < -1$)
5. Being unable to execute plan 3 if no bins exist.

In our case study there are a number of further assumptions that are implicitly made. These include: that paths between locations are never blocked, that robots never break down, that rubbish is measured in discrete units that can be picked up by a single robot, that the environment doesn't change, and that the robot's navigation systems are perfect. Additionally, we assume that the environment is a grid, which clearly isn't true for a real world. If the robot operates in a real environment, then this last assumption amounts to assuming that a symbolic representation of the environment is used, and that this representation is able to be accurately determined from sensors.

The point here is not to argue whether any one of these further assumptions is reasonable or not, or whether a given assumption would have been made. The key point is that assumptions are being made, and that they are often made implicitly and not documented questioned or justified.

To summarise, we have argued that correctly capturing the specification is not trivial, and that it is easy to either require too much (by requiring the agent to always succeed), or too little (for instance requiring something to happen "eventually" without considering reasonable time bounds). Furthermore, even if the specification is correct, there are a range of factors that are not typically considered in verification (such as human interaction), and it is possible for proofs to contain implicit assumptions that render them incorrect. In the next two sections we consider how to address these issues.

12.7 A New Approach to Assurance of Agent Systems

So, what *should* we be doing to obtain assurance of agent systems? Before we discuss our approach for obtaining assurance we need to emphasize that verification has a key role to play, since, in general testing on its own is not sufficient. Although in some applications testing may be enough — for instance, recall that for one application "*Even though this negotiation process has not been theoretically proved for cycles' avoidance* [sic], *practical experiments have validated its operation*" [349, Page 413] — we do not believe that in general it is sufficient.

The reason for not believing that testing is sufficient is twofold. Firstly, it is known, and widely accepted, that concurrent systems, which are able to exhibit time-dependent errors, are challenging to test. Multi-agent systems are concurrent systems. Worse, they are concurrent systems where the individual components (the agents) are able to behave flexibly and adaptively. This makes agent systems *harder* to test than other concurrent systems. Indeed, an analysis of the state space size for BDI systems [436] found that the space of possible behaviours was extremely large (e.g. in excess of 10^{100} for a uniform depth 3 tree, and over 10^{11} for a sample tree from an industrial workflow application). Tsai *et al.* report on similar analyses for knowledge-based systems, and conclude that [419, p205–206]:

> ... for systems that use either a selection method (such as MYCIN and INTERNIST) or the construction method (such as XCON, XSEL, and XFL), the potential sizes of the input and the output spaces for black-box testing are enormous.

Secondly, testing is not compositional, whereas proof can be. If we have a goal G which is decomposed into two sub-goals G_1 (achieved first) and G_2 (achieved second), then if we have tested G_1 and G_2 separately, it is not clear what we can conclude about G: the situations that result from achieving G_1 may not relate to the situations in which G_2 was tested. On the other hand, if we prove that G_1 always succeeds, and the assumptions that are needed to prove that G_2 succeeds are a consequence of the success of G_1, then separate proofs of the correctness of G_1 and G_2 *can* be combined to provide a proof of the correctness of G.

The remainder of this section briefly describes a proposed solution to obtaining assurance regarding the behaviour of a multi-agent system. As might be expected, the proposed solution is sketched briefly, and is somewhat tentative: more work is required.

The solution has two aspects: adopting a more pragmatic approach that assesses risks, and uses appropriate levels of mitigation and evidence of correctness (section 12.7.1); and combining testing and proving [17] (section 12.8). We also briefly discuss the issue of programming languages and approaches (section 12.7.2).

[17] In the remainder of this section we use the term "proving" broadly, to encompass any approach that (unlike testing) considers *all* possibilities; specifically, we use "proving" to encompass both mathematical proofs and model checking.

12.7.1 An Engineering Approach to Risk Management

Based on the issues highlighted, we feel that we should look at a more broad engineering solution: we need to adopt an engineering approach that quantifies the risk and then uses appropriate levels of evidence[18], as is argued by Jackson [250, Page 81]:

> . . . as in all engineering enterprises, dependability is a trade-off between benefits and risks, with the level of assurance (and the quality and cost of the evidence) being chosen to match the risk at hand.

Jackson argues for the use of *direct* evidence that a system meets its requirements: an end-to-end argument that provides evidence that the system exhibits desired properties. Jackson also argues that properties should be expressed in real world terms rather than in software terms. For example, specifying a safety property in terms of the radiation dose received by a patient, rather than in terms of the software computing a correct dose. This tends to encourage consideration of the wider context, for instance, how a radiation dose computed by the software is translated into a radiation dose that is delivered to a patient.

Rushby [378] also argues for a safety case that formally establishes that a claim follows from the system and (explicit) assumptions. He gives an example of an adaptive system, and proposes that verification before deployment be augmented with run-time monitoring of assumptions, which may be formally derived. It is worth noting that although he discusses "adaptive systems", these are not agent systems, and it is not clear whether agent systems are as complex, simpler, or more complex than the systems he discusses.

It is noteworthy that increasingly, the use of safety cases is becoming accepted, as reflected in government standards. For example, Bishop *et al.* [46, section 4.2] discuss a range of UK standards that have adopted safety cases, including the use of goal-based safety cases. The goal-based approach proposed by Bishop *et al.* [46] derives desired safety goals using a range of techniques, such as hazard analysis, and then provides *evidence* that supports the desired *claims* via *arguments*. A range of forms of evidence are used, including "*deterministic or analytical application of predetermined rules*" which covers formal proofs, as well as probabilistic analyses, and process-based arguments. Goal-based assurance is supported by a range of tools and notations, including the Goal Structuring Notation (GSN) [270]. Note that this work is not specific to software, let alone to agent software. It is clear (from earlier discussion in this paper) that assuring agent software presents particular challenges that affect the choice of claims, evidence and arguments; however, safety cases and goal-based assurance can still form a useful general framework for the assurance of agent systems.

[18] As a aside, it is interesting to observe that much of verification research is "European", whereas a more pragmatic approach based on risk management is more "American". One might speculate on whether this difference is cultural, or a result of the funding landscape, or of other factors, such as the relationships between academia and industry [147].

More broadly, there are a range of techniques for assessing possible failure modes and their risks (e.g. fault trees, event trees), but these have barely been used with MAS (an exception is [136]). These methodologies can allow assumptions to be identified, and can allow us to develop an *error model* which captures the errors that need to be dealt with.

In the context of our simple case study, applying one of these processes, perhaps a goal-based assurance case approach [46], would lead us to think about the wider context, and to realise that the desired top-level goal is not that the robot sees the world as being clean ($v(\ell) = 0$) but rather, that the world itself is empty of trash, i.e. $\forall \ell.m(\ell) = 0$.

One process for developing (formal) requirements that considers the larger context is that of Jones *et al.* [262]. They propose a design methodology for deriving the specification of the software-to-be from a specification of its environment. They present a design methodology that starts by capturing formally the "problem world", that is, an abstraction of the world in which the software is situated. The interface between the software-to-be and the problem world is specified abstractly in terms of interfaces, and "rely conditions": what conditions the software can rely upon; for example, that sensors and affectors work correctly. Doing this captures assumptions explicitly. A feature of the approach is that fault-free operation and faulty operation are dealt with separately.

In the context of the case study, an attempt to document the interface between the robot and its environment and explore the "rely conditions" would lead us to ask what grounds we have for believing that the robot's view matches reality, thus uncovering an (implicit) assumption.

We have argued that there is a need to broaden the scope of verification to consider humans. We now consider some examples of how verification techniques can be applied to verify systems including both software and humans.

The work of Bordini *et al.* [69] considers systems that involve collaboration between humans and software agents, specifically space missions. Agent teamwork is specified in *Brahms*, a language that has been developed over a number of years for modelling human activity. The key issues in verifying human-agent teamwork specified in Brahms are that the Brahms notation is quite complex, and that it lacks formal semantics. Three possible approaches are briefly discussed. One is to simply use Java Pathfinder (JPF) and run the Brahms implementation on top of this, but there are efficiency issues with doing this. Another approach is to reimplement Brahms within the AIL framework [140].

A third approach, which has some resemblance to the approach that is proposed in this chapter, is to use a stepping stone approach: translate Brahms models into Jason (using abstraction, so the Jason model isn't a precise re-implementation of the Brahms model). Then the Jason implementation can be formally verified, and runs of the Brahms model can be compared with runs of the Jason implementation.

The work of Rushby *et al.* [118, 376, 377] uses formal methods techniques to look for mode errors in cockpit interfaces, i.e. situations where a human pilot (who

may be operating under stress) is likely to mistakenly believe the system to be in a particular state when it is actually in a different state. This was done by modeling the system (e.g. as a finite state machine), modeling the human's mental model of the system (e.g. elicited from human pilots), and then using model checking to find divergences.

More broadly, there is work (e.g. [119,151]) that uses formal methods techniques to model human behaviour, and then reason about it. As Duce *et al.* [151, Section 3.3] note:

> ... safety critical systems typically involve human agents as well as computer agents, and once again we see that to be able to reason about the overall properties of the system we need to be able to reason at some level about the human agents in them.

This approach has begun to be explored in non-agent safety critical systems, but does not seem to have been considered in the context of agent systems.

12.7.2 "Send considered harmful?"

Finally, perhaps a more minor, but nonetheless important point, concerns the level of programming. We should aim to work at a level that avoid certain error classes. In the same way that avoiding manual memory allocation in favour of garbage collection avoids a whole class of errors, we should seek to work at a level that allow us to avoid error classes. For agents one particularly place to consider this issue concerns concurrency: agent systems are concurrent, but some concurrency-related errors could be avoided by working at a higher level than message sending and receiving. Building multi-agent interactions in terms of sending and receiving individual messages is error-prone, and could be argued to be analogous to programming based on "goto" statements. A similar argument has been independently put forward by Chopra and Singh [96].

A number of alternative ways of specifying and implementing interactions have been proposed in the agent literature (e.g. [95, 97, 273, 434, 435, 443]). What these approaches all have in common is that although they ultimately do realise communication by sending messages, the interaction is specified and implemented in terms of higher level constructs, and certain errors simply cannot occur as a result of this. For example, interactions that are designed (and implemented) in terms of social commitments are able to ensure "alignment" in the face of concurrency. Although different agents may perceive a different ordering of messages, under certain conditions, they will reach the same conclusions about the commitments that hold [97, 434, 435].

12.8 Combining Testing and Proving

As we have argued earlier, neither testing nor proving are sufficient on their own. However, each have strengths, and they can be used to complement each other. This is evidenced by the results from our case study: testing uncovered assumptions that the formal proof missed, but the formal proof covers all cases, which testing cannot. A similar argument about the complementability of testing (in the form of random search [329]) and model checking was presented by Gao *et al.* [187] who experimented with a specification that had been randomly injected with faults, and found that their Lurch tool found some bugs that the SPIN model checker didn't find (they also found that most of the randomly injected faults were easy to find using random testing).

So, we should look at combining testing and proving in a way that allows them to work effectively together. But how can we use testing and proving together in a meaningful way?

In this section we outline an approach for combining testing and proving. The proposed approach is generic, in that it applies to a range of software, not just agent systems. We believe that generality is an advantage, and that what is important is not whether the approach is *specific* to agent systems, but whether it is *applicable* to agent systems. It is possible for a generic approach to fail to be applicable, or to fail to be useful, in a more specific context.

The proposed approach is based on the recognition that testing and proving are merely two possible ways of trying to provide evidence of correctness, and that there are other, intermediate, approaches. We then use these intermediate approaches to build a "bridge" between testing and proving.

Testing and proving are just two particular techniques amongst many, which we classify along two dimensions (see Figure 12.2, ignore the arrows for now). The first dimension is *abstraction*: does the technique deal with the actual code that is running ("concrete"), or with an abstract model ("abstract")[19]? Specifically, an "abstract" model is one where the actual running code is not derived in an automated way from the model. We distinguish between "concrete" and "abstract" because an analysis of a concrete model tells us something about the actual running code, whereas analysis of an abstract model is one step removed from the actual executing code. The second dimension is the *coverage* of the state space: does the checking cover only certain points (testing)? does it cover all points within a sub-space (incomplete systematic exploration)? or does it cover all of the state space (complete systematic exploration)? We do note that some of these lines are somewhat imprecise: for instance, Java PathFinder does systematic exploration of Java code ("concrete"), but some approximations need to be made, so it's not completely concrete.

Testing and proving are familiar, but some of the other approaches in Figure 12.2 need to be briefly explained:

[19] We also have a "very abstract" classification for dynamical systems techniques.

Shallow Scope: The idea is that all possibilities are explored systematically (i.e. like model checking, rather than like testing) but only within a limited scope for variable values. For instance, we might consider $\mathbb{L} = \{0 \ldots 2\} \times \{0 \ldots 2\}$ and a maximum value for $m(\ell)$ or $v(\ell)$ of 1. The promise of this approach is the "shallow scope hypothesis" which states that many errors in models can be found with a fairly small scope. Experimentation with a range of models has provided evidence for the shallow scope hypothesis (e.g. [138, 249])

Systematic Enumeration: Systematically generating all possibilities, within a given scope. The difference between this and "Shallow Scope" is that it is done with an implementation, which may make it harder to work symbolically, or to map to other representations such as Ordered Binary Decision Diagrams. For example, in our case study, we could generate all possible starting configurations for a limited-size grid; and then execute them with the implementation.

Animation: Roughly speaking, executing specifications [269, 363]. Unlike executing programs, this may be possible only for some specifications (because not all specifications are executable), and may involve analysis to try and execute ("animate") specifications that are not in a convenient form to be executed directly.

Dynamical Systems: This approach is very abstract and considers the overall behaviour of the space of possible executions, viewed as a dynamical system. A typical question is whether there are attractors, and what is their basin of attraction. However, although this approach has promise, and is worthy of further work, there seems to have been little work on using such approaches for verification in the computing community (an exception is the work of Beer [29]). One challenge is that software systems are typically discrete, whereas dynamical systems are normally continuous.

So, how do we use these techniques to build a bridge between testing and proving? The key idea is to use small steps that only change one aspect of the taxonomy of Figure 12.2. For example (the numbers correspond to the numbered arrows in Figure 12.2):

❶ Comparing the outcome of testing the implementation (with selected test cases) with the outcome of systematic enumeration (still with the implementation) allows us to assess to what extent the selected test cases are sufficient.

If both approaches find the same errors, then this gives us confidence that the test cases are in fact sufficient. If we find errors with systematic generation that we don't find with the test cases, then the test cases are inadequate. If we find errors in the selected testing that are not found by systematic generation, then this tells us that the scope of generation is too limited (this is also assessed by comparing proving and systematic generation with the abstract model, discussed below).

❷ Comparing the outcome of systematic generation with the implementation and with an abstract model allows us to assess the difference that is made by changing to an abstract model.

Coverage: / **Abstraction:**	*Individual Test Cases*	*Systematic Exploration (incomplete)*	*Systematic Exploration (complete)*
Very abstract	-	-	Dynamical Systems
Abstract Model	Animation	Shallow Scope ❷ ↑	❸ ⟶ Model Checking, Proof
Concrete	Testing ❶ ⟶	Systematic Enumeration	-

Fig. 12.2 Approaches to Assurance: A two-dimensional taxonomy

If both approaches find the same errors, this gives us confidence that the abstract model corresponds to the implementation. If errors are found in the implementation but not in the abstract model (or vice versa), then this tells us that the abstract model is too abstract (or that it, or the implementation, is wrong).

❸ Comparing systematic generation within a limited scope ("Shallow Scope") with proving correctness (for the same model) gives us information on whether the scope is too limited, and hence is missing issues. If both approaches find the same errors, this gives us confidence that the scope limitation is not missing anything.

Taken together, these steps build a bridge that links proving and testing. At each step along the way we change only one thing which allows for conclusions to be drawn from differences, or lack thereof. Section 12.8.1 illustrates how these steps are applied to the waste disposal robot case study.

One particular issue is ensuring that multiple models capture the same thing. For instance, we might have an abstract formal model in one notation, and an implementation in another notation. The key idea that allows comparison is that we perform the *same* assurance approach on both models (typically systematic, but incomplete, exploration). This ensures that any differences found are due to differences between the models, not due to a difference in the assurance procedure. For example, we might be doing shallow scope exploration with an Alloy model and model checking with a Promela model. In this case, we would need to take multiple steps to move from one to the other. For instance an intermediate step might be model checking an Alloy model. Of course, our ability to do this may be limited by the available tools. Whilst systematic incomplete exploration can not provide perfect assurance

that two models correspond, it can provide stronger confidence than non-systematic exploration.

Another case where more steps may be useful is to consider a small scope when following arrow ❷, and then also doing shallow scope exploration (with the same abstract model) but with a larger scope. This can help gain further confidence that the smaller scope is not too small.

12.8.1 Applying the Proposed Approach to the Case Study

We now, for illustrative purposes, discuss the application of these steps to the case study. We begin with comparing systemic generation (within a limited scope) and selected test cases, both with the implementation (arrow ❶). The implementation was extended to systematically generate all initial states and to run the robot in each of these. We considered a grid of 2×2 (and also a smaller 2×1 grid). Initial states were generated with the following ranges of values:

- Each $v(\ell)$ was either $\perp$ (represented as -1), or was in the range $(0 \ldots 3)$
- Each $m(\ell)$ was in the range $0 \ldots 3$
- The robot's carrying capacity c ranged over $1 \ldots 4$
- The initial rubbish carried h ranged over $0 \ldots c$
- The number of bins ranged from 1 to the number of locations (i.e. $1 \ldots 4$ for the larger scope). We also separately experimented with allowing for no bins, which introduced a new error when plan 3 was unable to find a bin to move to.
- The robot could begin in any of the locations.

These generated 33,600 initial configurations for the 2-location scope, and 35,840,000 initial configurations for the 4-location scope. The robot program was run in each of the starting configurations (taking just under 44 minutes[20] for the larger scope, and less than 3 seconds for the smaller scope). The results were collected and analysed, and the following errors (described in section 12.5) were found to have occurred:

1. Completing execution without having cleaned the environment.
2. Going into an infinite loop[21] because pick had no effect.
3. Recovering (incorrectly) because -1 was interpreted as $\perp$, leading to a sense action.
4. No action being applicable (because $v(\ell) = -2$).

[20] On an idle 2.4 GHz Intel Core 2 Duo machine with 1 Gig of 667 MHz DDR2 SDRAM running Mac OS X Version 10.5.6.

[21] In fact the condition that led to this was detected and lead to an abort, rather than a loop.

5. Failure of plan 3 due to non-existence of bins in the environment.

We compared these errors with those found by random testing, which was done by randomly generating initial configurations. Each location ℓ had a 50% chance of having rubbish and each $v(\ell)$ was initialised to $\perp$ (90% chance), or to a random number (10% chance), which meant that there was less than 10% chance of an error for a given $v(\ell)$ because the randomly generated number might be the same as the corresponding $m(\ell)$. Running 100 random tests found the first two errors[22]. Considering the fourth error, it only occurs when $v(\ell) = -2$, which requires an initial setup of $m(\ell) = 3$ and $v(\ell) = 1$ and a capacity of at least 3. This combination of circumstances was not very likely to occur randomly. However, running 1,000 randomly generated test cases (taking less than half a second) did find all four error types. This comparison, between systematic generation and random testing, told us that our initial number of tests *was* too limited, and gave us confidence that the new number of tests was sufficient.

We then did a comparison between systematic generation with the implementation and with an abstract model (arrow ❷ in Figure 12.2). In our case study, since the proof is informal, we wanted to do systematic generation with a model that, like the proof, directly realised the model in Figure 12.1. We thus implemented a model in Prolog (available upon request) that directly followed that in Figure 12.1, and did systematic generation[23] with both a 2×2 and a 2×4 grid. In both grid sizes the same error cases were detected[24]:

1. Completing execution correctly, but with rubbish remaining in the environment (implementation error case 1)
2. Going into an infinite loop, due to pick not having any effect (implementation error case 2)
3. Out of domain error: attempting to store a value less than 0 into $v(\ell)$.
4. Being unable to select a bin if no bins exist (implementation error case 5)

Comparing these cases with the ones uncovered by the implementation, we can see that we find the same errors, with one exception. Because the abstract model checks whether values in $v(\ell)$ and $m(\ell)$ are in the appropriate range, it catches cases where a value less than 0 would be stored in $v(\ell)$ (by pick) and gives an error, which prevents error cases 3 and 4 (in the implementation) from occurring. This comparison tells us that the implementation is not quite faithful to the model, but in terms of the situations that cause errors, and the assumptions that are required to rule out such situations, the abstract model and the implementation do seem to be in agreement.

[22] We did not consider situations with no bins, so error 5 could not occur.

[23] For the larger scope this took just under 40 minutes (on a Dell PC with a 2.66 GHz Pentium 4, 1280 MB of RAM, running Ubuntu Linux 2.6.24-19) to generate 3,346,110 cases (with the assumptions not allowed to be violated, i.e. $b \neq \emptyset$ and initially $\forall \ell.v(\ell) = \perp$); whereas if these assumptions could be violated it took just under 1 hour and 46 minutes to generate 13,436,928 cases.

[24] If the assumptions that initially $\forall \ell.v(\ell) = \perp$ and that $b \neq \emptyset$ held then there were no errors.

Finally, considering the third link (arrow ❸ in Figure 12.2), we observe that the errors found by systematic generation in the abstract model catch the assumptions that render the proof faulty, and indeed, if we require these assumptions to hold, then systematic generation does not find any errors, which gives us additional confidence that the proof is now correct.

To summarise, we have applied the proposed "bridge building" mechanism to the case study and found that:

- 100 test cases (with the given parameters) were insufficient, but 1,000 test cases found the same errors as systematic generation within a limit scope. This gives us some confidence in the coverage of the tests.
- Systematic generation of tests cases with the implementation and the abstract model found the same error classes (with differences that highlighted the difference in error checking, and the use of -1 to represent $\perp$ in the implementation). This gives us some confidence that the abstract model and the implementation are capturing the same thing, and gives us some confidence in applying results about the abstract model to the implementation.
- Systematic generation of tests cases with the abstract model for both large and small scopes found the same error cases, which suggests that the small scope is sufficient.
- Comparing systematic generation with the abstract model and the proof of correctness given in section 12.4 we find that the systematic generation identifies the assumptions that the proof makes. Furthermore, the proof provides additional confidence that the correct behaviour observed in systematic generation with the assumptions will generalise and hold in all settings, not just within the limited scope of generation.

Taken together, these results give us confidence in the correctness of the robot program, by providing end-to-end evidence of correctness. More importantly, this process identifies the assumptions required for the proof.

12.8.2 Addressing Efficiency

As per the discussion in section 12.5.1 we do need to take care with specifications, in order to avoid having specifications that are too strong (requiring success, even though it may not always be possible), or too weak (only requiring success "eventually", rather than "reasonably soon"). For our case study the former is not relevant, because in this case study it is always possible to succeed. However, finishing within a reasonable amount of time *is* a possible issue and so we now consider efficiency.

There are a number of ways of looking at efficiency, and perhaps the simplest is to use the implementation, rather than the model. Of course, running the implementation with selected test cases is not very convincing: it may be that particular

situations result in much poorer performance, and certain applications do require a stronger guarantee on performance bounds than can be obtained with testing alone. For instance, we may need to be confident that our waste disposal robot will be able to clean a grid of a certain size within a certain amount of time. So, we use systematic generation to consider all possible initial configurations within a narrow scope. We use a 2×2 grid, where each $m(\ell)$ was in the range $0\ldots3$, capacity c ranged over $1\ldots4$, initial rubbish held h over $0\ldots c$, and the robot could begin in any of the locations. Since the question is, "if the robot completes, how long does it take?" we avoid errors (which would lead to non-completion) by enforcing the two assumptions discussed earlier. Specifically, we ensure that there are bins ($b \neq \emptyset$) and we initialise each $v(\ell)$ to $\perp$. We tracked the (Manhattan) distance travelled by the robot. This generated 57,344 initial configurations with the following efficiency:

- With no optimisations enabled ("noopt") the average distance travelled by the robot was 6.18. The minimum[25] distance was 3 and the maximum[26] was 26.
- With only the first optimisation enabled ("opt1") the average distance travelled by the robot was 6.23. The minimum was 3 and the maximum was 26.
- With only the second optimisation enabled ("opt2") the average distance was 5.35; the minimum and maximum were respectively 3 and 26.
- With both optimisations enabled ("optboth") the average distance travelled was 5.25; the minimum and maximum were 3 and 26.

Figure 12.3 shows the efficiency in terms of distance travelled in graphical form. The horizontal axis shows the distance travelled, and the vertical axis counts how many of the 57,344 cases required that given distance to be travelled. As can be seen, most of the cases don't involve a large distance, but there are a few that do. For instance, with no optimisations, around 93% of the possible initial configurations require traversing a distance of 10 or less; and around 99% of configurations require traversing a distance of 17 or less.

We also considered a 3×2 grid, for which the average distance was 14.6 without optimisations ("noopt") and 11.75 with the second optimisation ("opt2"), and the minimum distance was 5, and the maximum was 56 for both. As is shown in Figure 12.4, the distribution is similar to the 2×2 case. With no optimisations around 90% of configurations involve covering a distance of 21 or less, and 99% of configurations involve a distance of 36 or less (with the second optimisation these numbers are respectively 19 and 34). As in the 2×2 case, the maximum distance of 56 is very unlikely (4 cases out of 917,504 or 0.000436%).

Overall, from this exploration of efficiency we can conclude that, for the case study, there *are* cases where the effort required (in terms of distance) is significantly

[25] Since the robot did not begin with knowledge of the environment, even if there was no rubbish, it still had to cover the area to find this out.

[26] A distance of 26 may seem high for a 2×2 grid: it arose in situations where there was much rubbish (i.e. $m(\ell) = 3$ for at least locations 2 to 4), the robot had a low carrying capacity ($c = 1$), and only location 1 was a bin. This was a relatively rare situation (4 cases out of 57,344, or less than 0.01%).

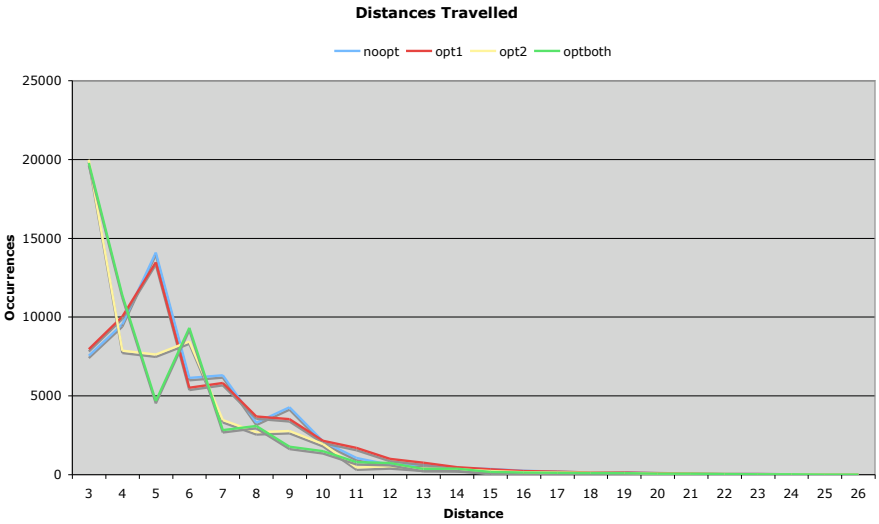

Fig. 12.3 Efficiency Profiles for 2×2 grid.

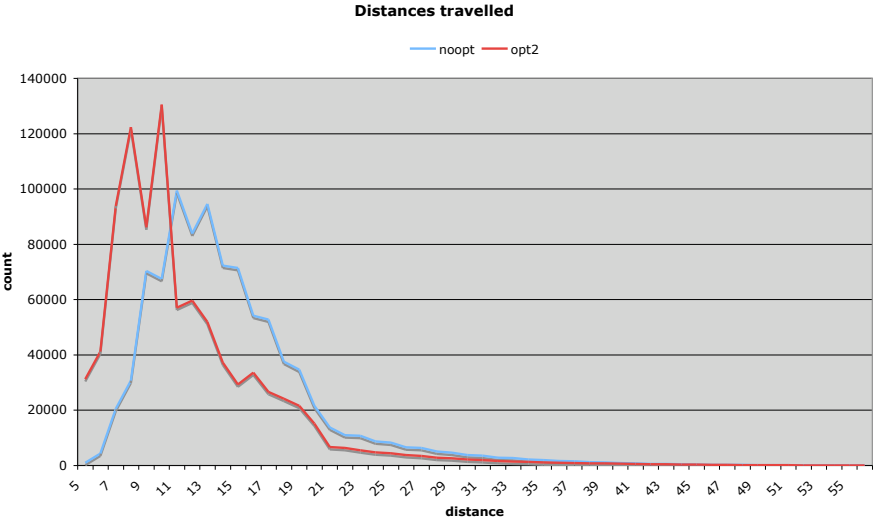

Fig. 12.4 Efficiency Profiles for 3×2 grid.

worse than the average or commonly encountered cases, and that consequently these cases are unlikely to be found through random testing. On the other hand, we have a bound on how much worse these cases are: the maximum distance travelled was around 3.8 to 4.95 times the average distance.

12.9 Conclusions

In this chapter we considered the issue of obtaining assurance that an agent system will behave appropriately when it is deployed. We asked what role formal verification are able to play in this process. We identified a number of issues relating to the difference between assurance and verification. Although this difference is not new, and applies to any type of software, there are some new issues that are specific to agent systems. For instance, the nature of the specification is different for agents: because they are situated in a dynamic and (often) failure-prone environment the specification cannot require that the agent always succeed, since success may not be possible in all situations.

Based on the range of issues identified, we concluded that formal verification techniques need to be used as part of a toolkit, in combination with other techniques. In order for this to happen, however, the focus of verification research needs to broaden to include human and organisational factors, and researchers need to be mindful of the context in which verification will be used.

We then outlined a possible solution for assuring the behaviour of agent systems that hinged on:

1. Adopting an engineering stance that develops an error model (which is also used for testing and model checking), and that seeks to quantify risks and develop appropriate levels of mitigation. It is important that in developing the error model we consider broadly the errors that can occur, including human errors.
2. Using the error model to guide various activities such as testing and model checking, which are linked by building a "bridge" between them, using intermediate techniques. In doing this is it important to take care with specifications, and to capture assumptions.

We also briefly discussed the issue of programming level, and how certain errors could perhaps be avoided by, for example, designing and programming agent interactions at a higher level then message sending/receiving.

We are not the first to propose the combination of proving and testing, although the combination that we proposed does appear to be novel in its detail and working. Lowry *et al.* [292] proposed integrating testing and proving, but focused on a cost-benefit trade-off evaluation method, rather than on proposing a specific mechanism for combining testing and proving (other than doing them separately). Richardson & Clarke [369] use a formal specification to help partition the space into equivalence

classes that are used in generating tests cases, they claim that "*Although this does not give total assurance in program reliability, we believe, and our evaluation supports this, that it provides strong evidence about the reliability of a program*". Dill [148] proposed a one-dimensional taxonomy which is similar in some ways to the one we use. Young & Taylor [444] also propose a taxonomy, and discuss using this to guide combining different techniques. They focus on approaches that would be regarded as testing rather than proving (e.g. testing, symbolic execution, and other analyses), but do briefly discuss proving (in section 7). Owen [330] is more recent, and considers the combination of tools from a pragmatic and empirical perspective. He uses a specification that has been randomly seeded with faults, and finds that many of the faults are not found by all of the tools, and that whilst running time varies, the vast majority of the faults are easy for at least one of the tools. He then proposes a detailed process for using the tools in combination to augment each other.

My hope is that this paper will perhaps encourage researchers working on agent verification techniques to reconsider their assumptions and scope, and in particular, to consider where the specification (often just assumed to be given as an input in some logic) comes from, what form it might have, what it might fail to capture, and how the specification form and notation might need to change to expand its scope (for instance to capture a social context).

As discussed in the previous sections, I see the key challenge as being how to integrate a range of techniques for obtaining assurance in such a way that allows them to strengthen and complement each other. For instance, finding a way of using testing and proving together so that we benefit from the ability of proving to consider all possibilities, and from the ability of testing to (sometimes) detect implicit assumptions. Whilst section 12.8 presented an approach for doing this, the approach is still not well developed, and more work is required, both on this specific approach, and on other approaches.

The key area for future work is thus to explore methods for obtaining assurance, such as the one presented in sections 12.7 and 12.8. Key considerations include how to capture the environment; how to develop an error model using techniques such as hazard analysis, event and fault trees; and how to include human and societal aspects in these models (by building on the work discussed in section 12.7.1).

A secondary area for work concerns approaches for specifying and implementing agent interactions. As discussed in section 12.7.2, approaches that work at the level of message sending are low-level and error-prone. A number of alternative approaches have been proposed in the literature, but these need to be assessed from an assurance perspective. Are these approaches really less error-prone? And are they easier to verify?

Some specific, more immediate, directions for future work include the following:

- Exploring the ideas of the previous section in the context of a larger, more realistic, case study.
- Determining whether (as discussed at the end of section 12.5.1) the corrected specification is harder to verify, and if so, how much harder?

- Looking at adapting ideas from the testing literature for agents, for example how might metamorphic testing [448] be used to test agent systems?
- In the previous section we argued that the various activities allowed us to have "some confidence" about a range of results. How much confidence? Can this be assessed? How?
- Looking at adapting ideas from the dynamical systems literature (e.g. [29]).

Acknowledgements I would like to thank Stephen Cranefield for discussions relating to this work, and for comments on a draft of this chapter. I would also like to thank the anonymous reviewers for their comments which were helpful in improving this paper.

References

1. Abrahamson, K.R.: Decidability and expressiveness of logics of processes. Ph.D. thesis, Department of Computer Science, University of Washington (1980)
2. Aştefănoaei, L., de Boer, F.: The refinement of multi-agent systems. In: this volume, Chapter 2
3. Ackermann, W.: Solvable Cases of the Decision Problem. North-Holland Publishing Company, Amsterdam (1962)
4. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based tool. In: Proc. of the European Meeting on Cybernetics and Systems Research, Vol. II, pp. 570–575 (2004)
5. Alchourrón, C.E., Bulygin, E.: Normative Systems. Springer Verlag (1971)
6. Alechina, N., Dastani, M., Logan, B., Meyer, J.-J. Ch.: A logic of agent programs. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007), pp. 795–800. AAAI Press (2007)
7. Alechina, N., Dastani, M., Logan, B., Meyer, J.-J. Ch.: Reasoning about agent deliberation. In: G. Brewka, J. Lang (eds.) Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08), pp. 16–26. AAAI, Sydney, Australia (2008)
8. Alechina, N., Logan, B., Dastani, M., Meyer, J.-J. Ch.: Reasoning about agent execution strategies. In: AAMAS (3), pp. 1455–1458 (2008)
9. Althoff, C.S., Thomas, W., Wallmeier, N.: Observations on determinization of büchi automata. Theor. Comput. Sci. **363**(2), 224–233 (2006). DOI http://dx.doi.org/10.1016/j.tcs.2006.07.026
10. Alur, R.: Timed automata. In: N. Halbwachs, D. Peled (eds.) CAV, *Lecture Notes in Computer Science*, vol. 1633, pp. 8–22. Springer (1999)
11. Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., Tasiran, S.: MOCHA user manual. In: Proceedings of CAV'98, *Lecture Notes in Computer Science*, vol. 1427, pp. 521–525 (1998)
12. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design **15**(1), 7–48 (1999)
13. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS), pp. 100–109. IEEE Computer Society Press (1997)
14. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM **49**, 672–713 (2002)
15. Anderson, A.: A reduction of deontic logic to alethic modal logic. Mind **22**, 100–103 (1958)
16. Andreka, H., van Benthem, J., Nemeti, I.: Modal languages and bounded fragments of predicate logic. Journal of Philosophical Logic **27**, 217–274 (1998)

M. Dastani et al. (eds.), *Specification and Verification of Multi-agent Systems*,
DOI 10.1007/978-1-4419-6984-2, © Springer Science+Business Media, LLC 2010

17. Apt, K.R., Bol, R.: Logic programming and negation: A survey. Journal of Logic Programming **19**, 9–71 (1994)
18. Arbab, F., Astefanoaei, L., de Boer, F.S., Dastani, M., Meyer, J.-J. Ch., Tinnemeier, N.: Reo connectors as coordination artifacts in 2APL systems. In: PRIMA, pp. 42–53 (2008)
19. Aştefănoaei, L., de Boer, F.S.: Model-checking agent refinement. In: Proceedings of the seventh international joint conference on autonomous agents and multiagent systems (AAMAS'08), pp. 705–712 (2008)
20. Aştefănoaei, L., de Boer, F.S., van Riemsdijk, M.B.: Using rewriting strategies for testing BUpL agents. In: D.D. Schreye (ed.) Preproceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2009) (2009)
21. Astefanoaei, L., Dastani, M., de Boer, F., Meyer, J.-J. Ch.: A verification framework for normative multi-agent systems. In: In the proceedings of The 11th Pacific Rim International Conference on Multi-Agents (PRIMA 2008), *LNAI*, vol. 5357. Springer (2008)
22. Bacmair, L., Ganzinger, H.: Resolution theorem proving. In: J.A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, chap. 2, pp. 19–99. Elsevier and MIT Press (2001)
23. de Bakker, J.: Mathematical Theory of Program Correctness. Prentice Hall, Inc. (1980)
24. Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Schifanella, C.: Verifying protocol conformance for logic-based communicating agents. In: Computational Logic in Multi-Agent Systems, pp. 196–212 (2004)
25. Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Schifanella, C.: Service selection by choreography-driven matching. In: C. Pautasso, T. Gschwind (eds.) WEWST, *CEUR Workshop Proceedings*, vol. 313. CEUR-WS.org (2007)
26. Ball, T., Rajamani, S.K.: The SLAM Toolkit. In: Proc. 13th International Conference on Computer Aided Verification (CAV), *Lecture Notes in Computer Science*, vol. 2102, pp. 260–264. Springer (2001)
27. Barringer, H., Fisher, M., Gabbay, D., Owens, R., Reynolds, M.: The Imperative Future: Principles of Executable Temporal Logic. John Wiley & Sons (1996)
28. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. Journal of Logic and Computation **In print** (2009)
29. Beer, R.D.: A dynamical systems perspective on agent-environment interaction. Artificial Intelligence **72**, 173–215 (1995)
30. Beeri, C.: On the menbership problem for functional and multivalued dependencies in relational databases. ACM Trans. Database Syst. **5**(3), 241–259 (1980)
31. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST, pp. 125–126. IEEE Computer Society (2006)
32. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE - a java agent development framework. In: Multi-Agent Programming: Languages, Platforms and Applications. Kluwer (2005)
33. Benerecetti, M., Cimatti, A.: Symbolic model checking for multi-agent systems. In: Proc. of the International Workshop on Model Checking and AI, pp. 1–8 (2002)
34. Benerecetti, M., Giunchiglia, F., Serafini, L.: Model checking multiagent systems. Journal of Logic and Computation **8**(3), 401–423 (1998)
35. Bentahar, J.: A pragmatic and semantic unified framework for agent communication. Ph.D. thesis, Laval University, Canada (2005)
36. Bentahar, J., Maamar, Z., Benslimane, D., Thiran, P.: An argumentation framework for communities of web services. IEEE Intelligent Systems **22**(6), 75–83 (2007)
37. Bentahar, J., Moulin, B., Chaib-draa, B.: A persuasion dialogue game based on commitments and arguments. In: Proc. of the International Workshop on Argumentation in Multi-Agent Systems, pp. 148–164 (2004)
38. Bentahar, J., Moulin, B., Meyer, J.-J. Ch., Chaib-draa, B.: A logical model for commitment and argument network for agent communication. In: Proc. of the International Joint Conference on AAMAS, pp. 792–799 (2004)
39. van Benthem, J.: The Logic of Time: A Model-theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse. Reidel, Dordrecht (1983)

40. van Benthem, J.: Extensive games as process models. Journal of Logic, Language and Information **11**, 289–313 (2002)
41. van Benthem, J.: Logic in games. Lecture Notes of the ILLC graduate course on Logic, Language and Information, Universiteit van Amsterdam, Amsterdam, The Netherlands (2005)
42. van Benthem, J., van Eijck, J., Stebletsova, V.: Modal logic, transition systems and processes. Journal of Logic and Computation **4**(5), 811–855 (1994)
43. Berezin, S., Clarke, E.M., Biere, A., Zhu, Y.: Verification of Out-Of-Order Processor Designs Using Model Checking and a Light-Weight Completion Function. Formal Methods in System Design **20**(2), 159–186 (2002)
44. Bhat, G., Cleaveland, R., Groce, A.: Efficient model checking via büchi tableau automata. In: Computer-Aided Verification, pp. 38–52 (2001)
45. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for ctl*. In: The IEEE Symposium on Logics in Computer Science, pp. 388–397 (1995)
46. Bishop, P., Bloomfield, R., Guerra, S.: The future of goal-based assurance cases. In: Proceedings of Workshop on Assurance Cases. Supplemental Volume of the 2004 International Conference on Dependable Systems and Networks, pp. 390–395 (2004)
47. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic, *Cambridge Tracts in Theoretical Computer Science*, vol. 53. Cambridge University Press (2001)
48. Boella, ., van der Torre, L.: Δ: The social delegation cycle. In: Deontic Logic: 7th International Workshop on Deontic Logic in Computer Science (ΔEON'04), *LNCS*, vol. 3065, pp. 29–42. Springer (2004)
49. Boella, G., Broersen, J., van der Torre, L.: Reasoning about constitutive norms, counts-as conditionals, institutions, deadlines and violations. In: PRIMA, pp. 86–97 (2008)
50. Boella, G., Pigozzi, G., van der Torre, L.: Five guidelines for normative multi-agent systems. In: Proceedings of JURIX 2009 (2009)
51. Boella, G., Pigozzi, G., van der Torre, L.: Norms in computer science: Ten guidelines for normative multi-agent systems. In: Normative Multi-Agent Systems (NorMAS'09), no. 09121 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI) (2009)
52. Boella, G., van der Torre, L.: Enforceable social laws. In: Procs. of 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05), pp. 682–689. ACM Press (2005)
53. Boella, G., van der Torre, L.: A game-theoretic approach to normative multi-agent systems. In: G. Boella, L. van der Torre, H. Verhagen (eds.) Normative Multi-agent Systems, no. 07122 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
54. Boella, G., van der Torre, L.: Substantive and procedural norms in normative multiagent systems. Journal of Applied Logic (In press)
55. Boella, G., van der Torre, L., Verhagen, H. (eds.): Normative Multi-Agent Systems, no. 07122 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
56. Boella, G., van der Torre, L.: The ontological properties of social roles in multi-agent systems: Definitional dependence, powers and roles playing roles. Artificial Intelligence and Law **15**(3), 201–221 (2007)
57. Boella, G., van der Torre, L., Verhagen, H.: Introduction to normative multi-agent systems. Computational and Mathematical Organization Theory **12**(2-3), 71–79 (2006)
58. Boella, G., van der Torre, L., Villata, S.: Conditional dependence networks in requirements engineering. In: Proceedings of COIN'09, LNCS (2009)
59. Boella, G., Verhagen, H., van der Torre, L.: Introduction to the special issue on normative multi-agent systems. Journal of Autonomous Agents and Multi Agent Systems **17**(1), 1–10 (2008)
60. de Boer, F.S., Hindriks, K.V., van der Hoek, W., Meyer, J.-J. Ch.: A Verification Framework for Agent Programming with Declarative Goals. Journal of Applied Logic **5**(2), 277–302 (2007)

61. Bordini, R., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking AgentSpeak. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'03), pp. 409–416 (2003)
62. Bordini, R., Visser, W., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking multi-agent programs with CASP. In: Computer-Aided Verification, pp. 110–113 (2003)
63. Bordini, R., Wooldridge, M., Hübner, J.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
64. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. Springer-Verlag (2005)
65. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Tools and Applications. Springer-Verlag (2009)
66. Bordini, R.H., Dennis, L.A., Farwer, B., Fisher, M.: Directions for agent model checking. In: this volume, Chapter 4
67. Bordini, R.H., Dennis, L.A., Farwer, B., Fisher, M.: Automated Verification of Multi-Agent Programs. In: Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 69–78 (2008)
68. Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking AgentSpeak. In: Autonomous Agents and Multiagent Systems (AAMAS), pp. 409–416 (2003)
69. Bordini, R.H., Fisher, M., Sierhuis, M.: Analysing human-agent teamwork. In: 10th ESA Workshop on Advanced Space Technologies for Robotics and Automation (ASTRA 2008). Noordwijk, The Netherlands. (2008)
70. Bordini, R.H., Fisher, M., Sierhuis, M.: Formal Verification of Human-robot Teamwork. In: Proceedings of the 4th ACM/IEEE International Conference on Human Robot Interaction (HRI), pp. 267–268. ACM (2009)
71. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Model Checking Rational Agents. IEEE Intelligent Systems **19**(5), 46–52 (2004)
72. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: State-Space Reduction Techniques in Agent Verification. In: Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 896–903. IEEE Computer Society (2004)
73. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying Multi-Agent Programs by Model Checking. Journal of Autonomous Agents and Multi-Agent Systems **12**(2), 239–256 (2006)
74. Bordini, R.H., Fisher, M., Wooldridge, M., Visser, W.: Property-Based Slicing for Agent Verification. Journal of Logic and Computation **to appear** (2009)
75. Bordini, R.H., Hübner, J.F., Vieira, R.: ***Jason*** and the Golden Fleece of agent-oriented programming. In: R.H. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, chap. 1. Springer-Verlag (2005)
76. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using *Jason*. Wiley Series in Agent Technology. John Wiley & Sons (2007)
77. Bordini, R.H., Moreira, A.F.: Proving the asymmetry thesis principles for a BDI agent-oriented programming language. Electronic Notes in Theoretical Computer Science **70(5)** (2002)
78. Bosse, T., Gerritsen, C., Treur, J.: Cognitive and social simulation of criminal behaviour: the intermittent explosive disorder case. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS'07, pp. 367–374. ACM Press (2007)
79. Bosse, T., Jonker, C., Los, S., van der Torre, L., Treur, J.: Formal analysis of trace conditioning. Cognitive Systems Research Journal **8**, 36–47 (2007)
80. Bosse, T., Jonker, C., van der Meij, L., Sharpanskykh, A., Treur, J.: Specification and verification of dynamics in agent models. International Journal of Cooperative Information Systems **18**(1), 167–193 (2009)
81. Bosse, T., Jonker, C., van der Meij, L., Treur, J.: A language and environment for analysis of dynamics by simulation. International Journal of Artificial Intelligence Tools **16**(3), 435–464 (2007)

82. Bosse, T., Jonker, C.M., van der Meij, L., Sharpanskykh, A., Treur, J.: Specification and verification of dynamics in cognitive agent models. In: IAT, pp. 247–254 (2006)
83. Bosse, T., Sharpanskykh, A., Treur, J.: Modelling complex systems by integration of agent-based and dynamical systems models. In: A. Minai, D. Braha, Y. Bar-Yam (eds.) Unifying Themes in Complex Systems VI, Proceedings of the Sixth International Conference on Complex Systems. Springer (2008)
84. Botía, J.A., Hernansaez, J.M., Gómez-Skarmeta, A.F.: On the application of clustering techniques to support debugging large-scale multi-agent systems. In: PROMAS, pp. 217–227 (2006)
85. Bouyer, P., Brinksma, E., Larsen, K.G.: Optimal infinite scheduling for multi-priced timed automata. Formal Methods in System Design **32**(1), 3–23 (2008)
86. Bratman, M.E.: Intentions, Plans, and Practical Reasoning. Harvard University Press: Cambridge, MA (1987)
87. Broersen, J., Dastani, M., Hulstijn, J., van der Torre, L.: Goal generation in the BOID architecture. Cognitive Science Quarterly **2**(3–4), 428–447 (2002)
88. Broersen, J., Dastani, M., van der Torre, L.: BDIOCTL: Obligations and the Specification of Agent Behavior. In: Proceedings of IJCAI'03, pp. 1389–1390 (2003)
89. Broersen, J., Herzig, A., Troquard, N.: A STIT-extension of ATL. In: JELIA, pp. 69–81 (2006)
90. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964)
91. Bulling, N.: Model checking coalition logic on implicit models is δ_3-complete. In: IfI Technical Reports (2010)
92. Bulling, N., Jamroga, W.: Verifying agents with memory is harder than it seemed. In: Proceedings of AAMAS 2010. ACM Press, Toronto, Canada (2010)
93. Bulygin, E.: Permissive norms and normative systems. In: A. Martino, F.S. Natali (eds.) Automated Analysis of Legal Texts, pp. 211–218. Publishing Company, Amsterdam (1986)
94. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison Wesley Publishing Company, Inc., Reading, Massachusetts (1988)
95. Cheong, C., Winikoff, M.: Hermes: Designing flexible and robust agent interactions. In: V. Dignum (ed.) Multi-Agent Systems – Semantics and Dynamics of Organizational Models, chap. 5, pp. 105–139. IGI (2009)
96. Chopra, A.K., Singh, M.P.: An architecture for multiagent systems: An approach based on commitments. In: Workshop on Programming Multiagent Systems (ProMAS) (2009)
97. Chopra, A.K., Singh, M.P.: Multiagent commitment alignment. In: Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 937–944 (2009)
98. Cimatti, A., Clarke, E., Giunchuglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Proc. 14th International Conference on Computer Aided Verification (CAV), *LNCS*, vol. 2404, pp. 359–364. Springer-Verlag (2002)
99. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proceedings of Logics of Programs Workshop, *LNCS*, vol. 131, pp. 52–71 (1981)
100. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
101. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems **8**(2), 244–263 (1986)
102. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
103. Clavel, M.: Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. The University of Chicago Press, Chicago (2000)
104. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude manual (version 2.4) (2009)
105. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, *Lecture Notes in Computer Science*, vol. 4350. Springer (2007)

106. Cleaveland, R.: Tableau-based model checking in the propositional mu-calculus. Acta Informatica **27**(8), 725–747 (1990)
107. Cleaveland, R., Sims, S.: Generic tools for verifying concurrent systems. Science of Computer Programming **41**(1), 39–47 (2002)
108. Cohen, M., Dam, M., Lomuscio, A., Russo, F.: Abstraction in model checking multi-agent systems. In: Proc. of AAMAS, pp. 945–952 (2009)
109. Cohen, P., Levesque, H.: Persistence, intentions and commitment. In: Intentions in Communication, pp. 33–69. MIT Press (1990)
110. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artificial Intelligence **42**(2-3), 213–261 (1990)
111. Cohen, P.R., Levesque, H.J.: Rational interaction as the basis for communication. In: P.R. Cohen, J. Morgan, M.E. Pollack (eds.) Intentions in Communication, pp. 221–255. MIT Press, Cambridge, MA (1990)
112. Cohen, P.R., Levesque, H.J.: Confirmations and Joint Action. In: Proc. International Joint Conference on Artificial Intelligence (IJCAI), pp. 951–959 (1991)
113. Cohen, P.R., Levesque, H.J.: Teamwork. Tech. Rep. 504, SRI International, Menlo Park, CA (1991)
114. Collier, R.: Debugging agents in agent factory. ProMAS 2006 pp. 229–248 (2007)
115. Colombetti, M.: A commitment-based approach to agent speech acts and conversations. In: Proc. of the International Autonomous Agent Workshop on Conversational Policies, pp. 21–29 (2000)
116. Conte, R., Castelfranchi, C., Dignum, F.: Autonomous norm acceptance. In: J. Müller, M.P. Singh, A.S. Rao (eds.) Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98), vol. 1555, pp. 99–112. Springer-Verlag: Heidelberg, Germany (1999)
117. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for verification of temporal properties. Formal Methods in System Design **1**, 275–288 (1992)
118. Crow, J., Javaux, D., Rushby, J.: Models and mechanized methods that integrate human factors into automation design. In: International Conference on Human-Computer Interaction in Aeronautics: HCI-Aero (2000)
119. Curzon, P., Rukšėnas, R., Blandford, A.: An approach to formal verification of human–computer interaction. Formal Aspects of Computing **19**(4), 513–550 (2007). DOI 10.1007/s00165-007-0035-6
120. Cysneiros, L., Yu., E.: Requirements engineering for large-scale multi-agent systems. In: Proceedings of the 1st International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'02), pp. 39–56. Springer Verlag (2002)
121. D. N. Lam, K.S.B.: Debugging agent behavior in an implemented agent system. ProMAS 2004 pp. 104–125 (2005)
122. Dastani, M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3), 214–248 (2008)
123. Dastani, M., Arbab, F., de Boer, F.S.: Coordination and composition in multi-agent systems. In: Procs. of AAMAS'05, pp. 439–446 (2005)
124. Dastani, M., Grossi, D., Meyer, J.-J. Ch., Tinnemeier, N.: Normative multi-agent programs and their logics. In: KRAMAS'08: Proceedings of the Workshop on Knowledge Representation for Agents and Multi-Agent Systems (2008)
125. Dastani, M., Grossi, D., Tinnemeier, N., Meyer, J.-J. Ch.: Normative multi-agent programs and their logics. In: G. Boella, G. Pigozzi, P. Noriega, H. Verhagen (eds.) Normative Multi-agent Systems, Dagstuhl Seminar Proceedings, vol. 09121 (2008)
126. Dastani, M., Meyer, J.-J. Ch.: Correctness of multi-agent programs: A hybrid approach. In: this volume, Chapter 6
127. Dastani, M., Meyer, J.-J. Ch.: A practical agent programming language. In: M. Dastani, A.E. Fallah-Seghrouchni, A. Ricci, M. Winikoff (eds.) Proceedings of the Fifth International Workshop on Programming Multi-agent Systems (ProMAS'07), *LNCS*, vol. 4908, pp. 107–123. Springer (2008)

128. Dastani, M., van Riemsdijk, M.B., Dignum, F., Meyer, J.-J. Ch.: A programming language for cognitive agents: Goal directed 3APL. In: Proc. ProMAS 2003, *LNCS*, vol. 3067, pp. 111–130. Springer (2004)
129. Dastani, M., van Riemsdijk, M.B., Meyer, J.-J. Ch.: Goal types in agent programming. In: Proceedings of the 17th European Conference on Artifical Intelligence 2006 (ECAI'06), *Frontiers in Artificial Intelligence and Applications*, vol. 141, pp. 220–224. IOS Press (2006)
130. Dastani, M., van Riemsdijk, M.B., Meyer, J.-J. Ch.: A grounded specification language for agent programs. In: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07), pp. 578–585. IFAAMAS, Honolulu, Hawaii (2007)
131. Dastani, M., Tinnemeier, N.A.M., Meyer, J.-J. Ch.: A programming language for normative multi-agent systems. In: V. Dignum (ed.) Multi-Agent Systems: Semantics and Dynamics of Organizational Models, chap. 16. IGI Global (2008)
132. Dastani, M., van Riemsdijk, M.B., Meyer, J.-J. Ch.: Programming Multi-Agent Systems in 3APL. chap. 2, pp. 39–67
133. Davoren, J., Nerode, A.: Logics for hybrid systems. Proceedings of the IEEE **88**(7), 985–1010 (2000)
134. De Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121**(1–2), 109–169 (2000)
135. De Giacomo, G., Levesque, H.J.: Progression using regression and sensors. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), pp. 160–165 (1999)
136. Dehlinger, J., Dugan, J.B.: Dynamic Event/Fault Tree Analysis of Multi-Agent Systems using Galileo. In: Integration of Software Engineering and Agent Technology (ISEAT), published as part of the Eighth International Conference on Quality Software (QSIC), pp. 429–434. IEEE Computer Society (2008). DOI 10.1109/QSIC.2008.14
137. Dembiński, P., Janowska, A., Janowski, P., Penczek, W., Półrola, A., Szreter, M., Woźna, B., Zbrzezny, A.: Verics: A tool for verifying timed automata and estelle specifications. In: Proceedings of the of the 9th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03), *LNCS*, vol. 2619, pp. 278–283. Springer (2003)
138. Dennis, G., Yessenov, K., Jackson, D.: Bounded verification of voting software. In: Second International Conference on Verified Software: Theories, Tools, Experiments, *LNCS*, vol. 5295, pp. 130–145. Springer (2008)
139. Dennis, L.A., Farwer, B.: Gwendolen: A BDI Language for Verifiable Agents. In: B. Löwe (ed.) Logic and the Simulation of Interaction and Reasoning. AISB, Aberdeen (2008)
140. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M.: A flexible framework for verifying agent programs. In: Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 1303–1306. IFAAMAS (2008)
141. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A Common Semantic Basis for BDI Languages. In: Proc. 7th International Workshop on Programming Multiagent Systems (ProMAS), *Lecture Notes in Artificial Intelligence*, vol. 4908, pp. 124–139. Springer Verlag (2008)
142. Dennis, L.A., Fisher, M.: Programming Verifiable Heterogeneous Agent Systems. In: Proc. 6th International Workshop on Programming in Multi-Agent Systems (ProMAS), *Lecture Notes in Computer Science*, vol. 5442, pp. 40–55. Springer Verlag (2008)
143. Dennis, L.A., Hepple, A., Fisher, M.: Language Constructs for Multi-Agent Programming. In: Proc. 8th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA), *Lecture Notes in Artificial Intelligence*, vol. 5056, pp. 137–156. Springer (2008)
144. Dennis, L.A., Tinnemeier, N.A.M., Meyer, J.-J. Ch.: Model checking normative agent organisations. In: Computational Logic in Multi-Agent Systems (CLIMA-X) (2009). To Appear
145. Dignum, V.: A model for organizational interaction. Ph.D. thesis, Utrecht University (2003)
146. Dignum, V. (ed.): Multi-Agent Systems - Semantics and Dynamics of Organizational Models. IGI Global (2009)

147. Dijkstra, E.W.: EWD611: On the fact that the Atlantic Ocean has two sides. http://www.cs.utexas.edu/users/EWD/index06xx.html, originally published as pages 268–276 of Selected Writings on Computing: A Personal Perspective, Springer-Verlag. ISBN 0–387–90652–5. (1982)
148. Dill, D.L.: What's between simulation and formal verification? (extended abstract). In: DAC '98: Proceedings of the 35th annual conference on Design automation, pp. 328–329. ACM, New York, NY, USA (1998). DOI 10.1145/277044.277138
149. van Ditmarsch, H.P., Herzig, A., De Lima, T.: Optimal regression for reasoning about knowledge and actions. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007), pp. 1070–1075. AAAI Press (2007)
150. van Drimmelen, G.: Satisfiability in alternating-time temporal logic. In: Proceedings of LICS'2003, pp. 208–217. IEEE Computer Society Press (2003)
151. Duce, D., Duke, D.: Syndetic modelling:: Computer science meets cognitive psychology. Electronic Notes in Theoretical Computer Science **43**, 50 – 74 (2001). DOI 10.1016/S1571-0661(04)80894-6. Formal Methods Elsewhere (a Satellite Workshop of FORTE-PSTV-2000 devoted to applications of formal methods to areas other than communication protocols and software engineering)
152. Duff, S., Harland, J., Thangarajah, J.: On Proactivity and Maintenance Goals. In: Proceedings of the fifth international joint conference on autonomous agents and multiagent systems (AAMAS'06), pp. 1033–1040. Hakodate (2006)
153. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering (ICSE), pp. 411–420 (1999)
154. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. Electronic Notes in Theoretical Computer Science **174**(11), 3–25 (2007)
155. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: F. Gaducci, U. Montanari (eds.) Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA 2002), *Electronic Notes in Theoretical Computer Science*, vol. 71. Elsevier (2002)
156. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker and its implementation. In: Model Checking Software: Proc. 10 th Intl. SPIN Workshop, *LNCS*, vol. 2648, pp. 230–234. Springer (2003). URL citeseer.ist.psu.edu/eker03maude.html
157. Ekinci, E.E., Tiryaki, A.M., Çetin, Ö.: Goal-oriented agent testing revisited. In: J.J. Gomez-Sanz, M. Luck (eds.) Ninth International Workshop on Agent-Oriented Software Engineering (AOSE), pp. 85–96 (2008)
158. Emerson, E., Halpern, J.: "sometimes" and "not never" revisited: On branching versus linear time temporal logic. Journal of the ACM **33**(1), 151–178 (1986)
159. Emerson, E.A.: Temporal and modal logic. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. Elsevier Science Publishers (1990)
160. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. In: SFCS '88: Proceedings of the 29th Annual Symposium on Foundations of Computer Science, pp. 328–337. IEEE Computer Society, Washington, DC, USA (1988)
161. Emerson, E.A., Lei, C.L.: Modalities for model checking: Branching time logic strikes back. Science of Computer Programming **8**(3), 275–306 (1987)
162. Emerson, E.A., Sistla, A.P.: Deciding branching time logic. In: STOC '84: Proceedings of EL87 sixteenth annual ACM symposium on Theory of computing, pp. 14–24. ACM, New York, NY, USA (1984)
163. Endriss, U., Maudet, N., Sadri, F., Toni, F.: Protocol conformance for logic-based agents. In: Proc. of the International Joint Conference on AI, pp. 679–684 (2003)
164. Engelfriet, J., Jonker, C.M., Treur, J.: Compositional verification of multi-agent systems in temporal multi-epistemic logic. In: J.P. Müller, M.P. Singh, A.S. Rao (eds.) Intelligent Agents V: Proceedings of the Fifth International Workshop on Agent Theories, Architectures and languages (ATAL'98), *LNAI*, vol. 1555, pp. 177–194. Springer-Verlag (1999)

165. Engelhardt, K., van der Meyden, R., Moses, Y.: Knowledge and the logic of local propositions. In: Proc. of the International Conference on Theoretical Aspects of Reasoning about Knowledge, pp. 29–41 (1998)
166. Esteva, M., Rodríguez-Aguilar, J., Rosell, B., Arcos, J.: Ameli: An agent-based middleware for electronic institutions. In: Proc. of AAMAS'04. New York, US (2004)
167. Esteva, M., Rosell, B., Rodríguez-Aguilar, J.A., Arcos, J.L.: AMELI: An Agent-Based Middleware for Electronic Institutions. In: AAMAS, pp. 236–243. IEEE Computer Society (2004)
168. Ezekiel, J., Lomuscio, A.: Combining fault injection and model checking to verify fault tolerance in multi-agent systems. In: Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 113–120 (2009)
169. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press: Cambridge, MA (1995)
170. Farwer, B., Dennis, L.: Translating into an intermediate agent layer: A prototype in Maude. In: Proceedings of Concurrency, Specification, and Programming (CS&P'07), pp. 168–179 (2007)
171. Ferber, J., Gutknecht, O.: A Meta-model for the Analysis and Design of Organizations in Multi-agent Systems. In: Proc. Third International Conference on Multi-Agent Systems (ICMAS), pp. 128–135 (1998)
172. Ferber, J., Gutknecht, O., Michel, F.: From Agents to Organizations: An Organizational View of Multi-agent Systems, vol. 2935, pp. 214–230. Springer (2004)
173. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. **18**(2), 194–211 (1979)
174. Fisher, M.: MetateM: The Story so Far. In: Proc. 3rd International Workshop on Programming Multiagent Systems (ProMAS), *LNAI*, vol. 3862, pp. 3–22. Springer Verlag (2006)
175. Fisher, M., Bordini, R., Hirsch, B., Torroni, P.: Computational Logics and Agents: A Roadmap of Current Technologies and Future Trends. Computational Intelligence **23**(1), 61–91 (2007)
176. Fisher, M., Bordini, R.H., Sierhuis, M.: Analysing Human-Agent Teamwork. In: Proc. 10th ESA Workshop on Advanced Space Technologies for Robotics and Automation (ASTRA). Katwijk, Netherlands. (2008)
177. Fisher, M., Ghidini, C., Hirsch, B.: Programming Groups of Rational Agents. In: Computational Logic in Multi-Agent Systems (CLIMA-IV), *Lecture Notes in Computer Science*, vol. 3259, pp. 849–856. Springer-Verlag (2004)
178. Fisher, M., Ghidini, C., Kakoudakis, T.: Dynamic Team Formation in Executable Agent-Based Systems. In: Rouff et al. [374]
179. Fisher, M., Kakoudakis, T.: Flexible Agent Grouping in Executable Temporal Logic. In: Gergatsoulis, Rondogiannis (eds.) Intensional Programming II. World Scientific Publishing Co. (2000)
180. Fitting, M.: First-order Logic and Automated Theorem Proving. MIT Press, Springer Verlag (1996)
181. Floyd, R.W.: Assigning Meaning to Programs. In: J.T. Schwartz (ed.) Mathematical Aspects of Computer Science: Proc. American Mathematics Soc. symposia, vol. 19, pp. 19–31. American Mathematical Society, Providence RI (1967)
182. Fornara, N., Colombetti, M.: Operational specification of a commitment-based agent communication language. In: Proc. of the International Joint Conference on AAMAS, pp. 535–542 (2002)
183. Fox, M.S.: An Organizational View of Distributed Systems. In: Distributed Artificial Intelligence, pp. 140–150. Morgan Kaufmann Publishers Inc., San Francisco, USA (1988)
184. Francez, N.: Fairness. Springer-Verlag, New York (1986)
185. Furst, M., Saxe, J.B., Sipser, M.: Parity, circuits, and the polynomial-time hierarchy. Math. Systems Theory **17**, 13–27 (1984)
186. Fuxman, A., Liu, L., Pistore, M., Roveri, M., Mylopoulos, J.: Specifying and analyzing early requirements in tropos. Requirements Engineering Journal **9**(2), 132–150 (2004)

187. Gao, J., Heimdahl, M., Owen, D., Menzies, T.: On the distribution of property violations in formal models: An initial study. In: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06). IEEE Computer Society (2006)
188. Garcia-Camino, A., Noriega, P., Rodriguez-Aguilar, J.A.: Implementing norms in electronic institutions. In: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (AAMAS'05), pp. 667–673. ACM Press (2005)
189. Gardelli, L., Viroli, M., Omicini, A.: Design Patterns for Self-Organizing Multiagent Systems. In: Proc. 2nd International Workshop on Engineering Emergence in Decentralised Autonomic Systems (EEDAS), pp. 61–70. CMS Press, London, UK (2007)
190. Gelernter, D., Zuck, L.D.: On what linda is: Formal description of linda as a reactive system. In: D. Garlan, D.L. Métayer (eds.) COORDINATION, *Lecture Notes in Computer Science*, vol. 1282, pp. 187–204. Springer (1997)
191. Georgeff, M.P., Lansky, A.L.: Reactive Reasoning and Planning. In: Proc. American National Conference on Artificial Intelligence (AAAI), pp. 677–682 (1987)
192. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)
193. Giacomo, G.d., Lespérance, Y., Levesque, H.: *ConGolog*, a Concurrent Programming Language Based on the Situation Calculus. Artificial Intelligence **121**(1-2), 109–169 (2000)
194. Giddens, A.: Social Theory and Modern Sociology. Polity Press (1984)
195. Giordano, L., Martelli, A., Schwind, C.: Verifying communicating agents by model checking in a temporal action logic. In: Logics in Artificial Intelligence, pp. 57–69 (2004)
196. van Glabbeek, R.J.: The linear time-branching time spectrum (extended abstract). In: J.C.M. Baeten, J.W. Klop (eds.) CONCUR, *Lecture Notes in Computer Science*, vol. 458, pp. 278–297. Springer (1990)
197. Goffman, E.: Strategic interaction. Basil Blackwell, Oxford (1970)
198. Goldblatt, R.: Logics of Time and Computation, *CSLI Lecture Notes*, vol. 7, 2nd edn. Springer (1992)
199. Gomez-Sanz, J.J., Botía, J., Serrano, E., Pavón, J.: Testing and debugging of MAS interactions with INGENIAS. In: J.J. Gomez-Sanz, M. Luck (eds.) Ninth International Workshop on Agent-Oriented Software Engineering (AOSE), pp. 133–144 (2008)
200. Goranko, V., van Drimmelen, G.: Complete axiomatization and decidability of the Alternating-time Temporal Logic (2003)
201. Goranko, V., Jamroga, W.: Comparing semantics of logics for multi-agent systems. Synthese **139**(2), 241–280 (2004)
202. Grossi, D.: Designing invisible handcuffs. formal investigations in institutions and organizations for multi-agent systems. Ph.D. thesis, Utrecht University, SIKS (2007)
203. Grossi, D., Aldewereld, H., Dignum, F.: Ubi lex ibi poena. designing norm enforcement in electronic institutions. In: V. Dignum, N. Fornara, P. Noriega, G. Boella, O. Boissier, E. Matson, J. Vázquez-Salceda J.zquez-Salceda (eds.) Proceedings of COIN@AAMAS'06, *LNCS*, vol. 4386, pp. 101–114. Springer (2006)
204. Hall, K.H., Staron, R.J., Vrba, P.: Experience with holonic and agent-based control systems and their adoption by industry. In: V. Mařík, R. Brennan, M. Pěchouček (eds.) HoloMAS 2005, *Lecture Notes in Artificial Intelligence (LNAI)*, vol. 3593, pp. 1–10 (2005)
205. Halpern, R.F.J., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT Press, Cambridge (1995)
206. Hansen, J.: Imperatives and Deontic Logic: On the Semantic Foundations of Deontic Logic. University of Leipzig (2008)
207. Hansen, J., Pigozzi, G., van der Torre, L.: Ten philosophical problems in deontic logic. In: G. Boella, L. van der Torre, H. Verhagen (eds.) Normative Multi-agent Systems, no. 07122 in DROPS Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
208. Harding, A., Ryan, M., Schobbens, P.Y.: Approximating ATL* in ATL. In: VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 289–301. Springer-Verlag, London, UK (2002)

209. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: D. Gabbay, F. Guenthner (eds.) Handbook of Philosophical Logic: Volume II: Extensions of Classical Logic, pp. 497–604. Reidel, Dordrecht, The Netherlands (1984)
210. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
211. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer (STTT) (1999)
212. Hennessy, M.: The Semantics of Programming Languages. John Wiley & Sons (1990)
213. Hepple, A., Dennis, L.A., Fisher, M.: A Common Basis for Agent Organisations in BDI Languages. In: Proc. International Workshop on LAnguages, methodologies and Development tools for multi-agent systemS (LADS), *Lecture Notes in Artificial Intelligence*, vol. 5118, pp. 171–88. Springer-Verlag (2008)
214. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: R.M. Hierons, J.P. Bowen, M. Harman (eds.) Formal Methods and Testing, *Lecture Notes in Computer Science*, vol. 4949, pp. 77–117. Springer (2008)
215. Hindriks, K.: Modules as Policy-Based Intentions: Modular Agent Programming in GOAL. In: Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'07), vol. 4908, pp. 156–171 (2008)
216. Hindriks, K., de Boer, F., van der Hoek, W., Meyer, J.-J. Ch.: A Programming Logic for part of the Agent Language 3APL. In: J. Rash (ed.) Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems, pp. 78–89. Springer-Verlag (2001)
217. Hindriks, K., van der Hoek, W.: GOAL agents instantiate intention logic. In: Proceedings of the 11th European Conference on Logics in Artificial Intelligence (JELIA'08), pp. 232–244 (2008)
218. Hindriks, K., Meyer, J.-J. Ch.: Agent logics as program logics: Grounding KARO. In: Proc. 29th German Conference on AI (KI 2006), *LNAI*, vol. 4314. Springer (2007)
219. Hindriks, K., Meyer, J.-J. Ch.: Toward a programming theory for rational agents. Journal of Autonomous Agents and Multi-Agent Systems (special issue FAMAS 2006) **19**, 4–29 (2009)
220. Hindriks, K., van Riemsdijk, B.: A computational semantics for communicating rational agents based on mental models. In: Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'09) (to appear)
221. Hindriks, K.V.: Programming Rational Agents in GOAL. In: Bordini et al. [65], chap. 4, pp. 119–157
222. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J. Ch.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems **2**(4), 357–401 (1999)
223. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J. Ch.: Agent Programming in 3APL. Autonomous Agents and Multi-Agent Systems **2**(4), 357–401 (1999)
224. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J. Ch.: Agent programming with declarative goals. In: C. Castelfranchi, Y. Lespérance (eds.) Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop, ATAL 2000, Boston, MA, USA, July 7-9, 2000, Proceedings, *Lecture Notes in Computer Science*, vol. 1986, pp. 228–243. Springer (2001)
225. Hindriks, K.V., Lespérance, Y., Levesque, H.: An Embedding of ConGolog in 3APL. In: Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'00), pp. 558–562 (2000)
226. Hindriks, K.V., Meyer, J.-J. Ch.: Towards a programming theory for rational agents. Autonomous Agents and Multi-Agent Systems **19**(1), 4–29 (2009)
227. Hindriks, K.V., van Riemsdijk, M.B., van der Hoek, W.: Agent programming with temporally extended goals. In: Proceedings of the 8th International Conference on Autonomous Agents and Multi-Agent Systems, p. to appear (2009)
228. Hindriks, K.V., Roberti, T.: Goal as a planning formalism. In: Proceedings of MATES 2009 (2009)
229. Hintikka, J.: Logic, Language Games and Information. Clarendon Press : Oxford (1973)
230. Hirsch, B.: Programming Rational Agents. Ph.D. thesis, Department of Computer Science, University of Liverpool (2005)

231. van der Hoek, W., Lomuscio, A., Wooldridge, M.: On the complexity of practical ATL model checking. In: P. Stone, G. Weiss (eds.) Proceedings of AAMAS'06, pp. 201–208 (2006)
232. van der Hoek, W., Wooldridge, M.: Model checking knowledge and time. In: SPIN 2002 - Proceedings of the Ninth International SPIN Workshop on Model Checking of Software, pp. 95–111 (2002)
233. van der Hoek, W., Wooldridge, M.: Cooperation, knowledge and time: Alternating-time Temporal Epistemic Logic and its applications. Studia Logica **75**(1), 125–157 (2003)
234. van der Hoek, W., Wooldridge, M.: Towards a logic of rational agency. Logic Journal of the IGPL **11**(2), 133–157 (2003)
235. Hollis, M.: Trust within reason. Cambridge University Press, Cambridge (1998)
236. Holzmann, G.: The model checker SPIN. IEEE Trans. Software Engineering **23**(5), 279–295 (1997)
237. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
238. Hooman, J.: Compositional verification of a distributed real-time arbitration protocol. Real-Time Systems **6**, 173–206 (1994)
239. Hopcroft, J.E., Motwani, R., Rotwani, Ullman, J.D.: Introduction to Automata Theory, Languages and Computability. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
240. Horling, B., Lesser, V.: A survey of multi-agent organizational paradigms. The Knowledge Engineering Review **19**(4), 281–316 (2004)
241. Hübner, J.F., Sichman, J.S., Boissier, O.: A Model for the Structural, Functional, and Deontic Specification of Organizations in Multiagent Systems. In: Proc. 16th Brazilian Symposium on Artificial Intelligence (SBIA), pp. 118–128. Springer-Verlag, London, UK (2002)
242. Hübner, J.F., Sichman, J.S., Boissier, O.: Moise+: towards a structural, functional, and deontic model for mas organization. In: AAMAS, pp. 501–502. ACM (2002)
243. Hubner, J.F., Sichman, J.S., Boissier, O.: Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. Int. J. Agent-Oriented Softw. Eng. **1**(3/4), 370–395 (2007)
244. Huget, M., Wooldridge, M.: Model checking for ACL compliance verification. In: Advances in Agent Communication, pp. 75–90 (2004)
245. Hurwicz, L.: Optimality and informational efficiency in resource allocation processes. In: K. Arrow, S. Karlin, P. Suppes (eds.) Mathematical Methods in the Social Sciences. Stanford University Press (1960)
246. Hustadt, U., Schmidt, R.A.: MSPASS: Modal reasoning by translation and first-order resolution. In: Proc. TABLEAUX 2000, *LNCS*, vol. 1847, pp. 67–71. Springer (2000)
247. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press (2004). (2nd Edition)
248. Immerman, N.: Number of quantifiers is better than number of tape cells. Journal of Computer and System Sciences **22**(3), 384 – 406 (1981)
249. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
250. Jackson, D.: A direct path to dependable software. CACM **52**(4), 78–88 (2009). DOI 10.1145/1498765.1498787
251. Jackson, M.O.: A crash course in implementation theory. Social Choice and Welfare **18**, 655–708 (2001)
252. Jackson, M.O.: Mechanism theory. In: U. Derigs (ed.) Encyclopedia of Life Support Systems. EOLSS Publishers (2003)
253. Jamroga, W.: A temporal logic for stochastic multi-agent systems. In: Proceedings of PRIMA'08, *LNCS*, vol. 5357, pp. 239–250 (2008)
254. Jamroga, W., Ågotnes, T.: Modular interpreted systems: A preliminary report. Tech. Rep. IfI-06-15, Clausthal University of Technology (2006)
255. Jamroga, W., Ågotnes, T.: Constructive knowledge: What agents can achieve under incomplete information. Journal of Applied Non-Classical Logics **17**(4), 423–475 (2007)
256. Jamroga, W., Ågotnes, T.: Modular interpreted systems. In: Proceedings of AAMAS'07, pp. 892–899 (2007)

257. Jamroga, W., Dix, J.: Do agents make model checking explode (computationally)? In: M. Pěchouček, P. Petta, L. Varga (eds.) Proceedings of CEEMAS 2005, *Lecture Notes in Computer Science*, vol. 3690, pp. 398–407. Springer Verlag (2005)
258. Jamroga, W., Dix, J.: Model checking ATL_{ir} is indeed Δ_2^P-complete. In: Proceedings of EUMAS'06 (2006)
259. Jamroga, W., Dix, J.: Model checking abilities of agents: A closer look. Theory of Computing Systems **42**(3), 366–410 (2008)
260. Jones, A.J.I., Sergot, M.: On the characterization of law and computer systems. Deontic Logic in Computer Science pp. 275–307 (1993)
261. Jones, C.: What can we do (technically) to get 'the right specification'? (2005). IFIP TC2 Working Conference, VSTTE, ETH Zurich
262. Jones, C.B., Hayes, I.J., Jackson, M.A.: Deriving specifications for systems that are connected to the physical world. In: C.B. Jones, Z. Liu, J. Woodcock (eds.) Formal Methods and Hybrid Real-Time Systems: Essays in Honour of Dines Bjørner and Zhou Chaochen on the Occassion of Their 70th Birthdays, *Lecture Notes in Computer Science*, vol. 4700, pp. 364–390. Springer Verlag (2007). DOI 10.1007/978-3-540-75221-9_16
263. Jonker, C., Sharpanskykh, A., Treur, J., Yolum, P.: A framework for formal modeling and analysis of organizations. Applied Intelligence **27**(1), 49–66 (2007)
264. Jonker, C., Treur, J.: Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. International Journal of Cooperative Information Systems **11**, 51–92 (2002)
265. Java PathFinder. `http://javapathfinder.sourceforge.net`
266. Kacprzak, M., Lomuscio, A., Penczek, W.: Verification of multiagent systems via unbounded model checking. In: Proc. of the International Joint Conference on AAMAS, pp. 638–645 (2004)
267. Kacprzak, M., Penczek, W.: Unbounded model checking for alternating-time temporal logic. In: The International Joint Conference on AAMAS, pp. 646–653 (2004)
268. Kanger, S.: New fondations for ethical theory. In: R. Hilpinen (ed.) Deontic Logic: Introductory and Systematic Readings, pp. 36–58. Reidel Publishing Company (1971)
269. Kazmierczak, E., Dart, P., Sterling, L., Winikoff, M.: Verifying requirements through mathematical modelling and animation. International Journal of Software Engineering and Knowledge Engineering **10**(2), 251–273 (2000)
270. Kelly, T., Weaver, R.: The goal structuring notation–a safety argument notation. In: Proc. DSN Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities (2004)
271. Kinny, D., Ljungberg, M., Rao, A.S., Sonenberg, E., Tidhar, G., Werner, E.: Planned Team Activity. In: Artificial Social Systems — Selected Papers from the Fourth European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW), *Lecture Notes in Artificial Intelligence*, vol. 830, pp. 226–256. Springer-Verlag (1992)
272. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing **4**, 67–95 (2002)
273. Kremer, R., Flores, R.: Using a performative subsumption lattice to support commitment-based conversations. In: F. Dignum, V. Dignum, S. Koenig, S. Kraus, M.P. Singh, M. Wooldridge (eds.) Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 114–121. ACM Press (2005)
274. Kumar, S., Cohen, P.R.: STAPLE: An Agent Programming Language Based on the Joint Intention Theory. In: Proc. 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1390–1391. IEEE Computer Society (2004)
275. Kupferman, O., Vardi, M., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM **47**(2), 312–360 (2000)
276. Lakemeyer, G., Levesque, H.J.: AOL: a logic of acting, sensing, knowing, and only knowing. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR-98), pp. 316–327 (1998)
277. Laroussinie, F.: About the expressive power of CTL combinators. Information Processing Letters **54**(6), 343–345 (1995)

278. Laroussinie, F., Markey, N., Oreiby, G.: Expressiveness and complexity of ATL. Tech. Rep. LSV-06-03, CNRS & ENS Cachan, France (2006)
279. Laroussinie, F., Markey, N., Oreiby, G.: On the expressiveness and complexity of atl. LNCS **4**, 7 (2008)
280. Laroussinie, F., Markey, N., Schnoebelen, P.: Model checking CTL+ and FCTL is hard. In: Proceedings of FoSSaCS'01, *Lecture Notes in Computer Science*, vol. 2030, pp. 318–331. Springer (2001)
281. Lespérance, Y., Levesque, H.J., Reiter, R.: A situation calculus approach to modeling and programming agents. In: A. Rao, M. Wooldridge (eds.) Foundations and Theories of Rational Agency, pp. 275–299. Kluwer (1999)
282. Levesque, H.J., Cohen, P.R., Nunes, J.H.T.: On Acting Together. In: Proc. 8th American National Conference on Artificial Intelligence (AAAI), pp. 94–99 (1990)
283. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31**, 59–84 (1997)
284. Lewis, D.: A problem about permission. In: E. Saarinen (ed.) Essays in Honour of Jaakko Hintikka, pp. 163–175. D. Reidel, Dordrecht (1979)
285. Lewis, D., Dobson, S.: Autonomic, Pervasive and Context-Aware Systems. Jouranl of Network System Management **15**(1), 1–3 (2007)
286. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 97–107. ACM, New York, NY, USA (1985)
287. Lindahl, L., Odelstad, J.: Open and closed intermediaries in normative systems. In: T. van Engers (ed.) Proceedings of the Nineteenth JURIX Conference on Legal Knowledge and Information Systems (JURIX 2006), pp. 91–100 (2006)
288. Liu, Y.: A Hoare-style proof system for robot programs. In: Proceedings of the Eighteenth National Conference on Artificial intelligence (AAAI/IAAI'02), pp. 74 – 79 (2002)
289. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1993)
290. Lomuscio, A., Raimondi, F.: Mcmas: A model checker for multi-agent systems. In: Proc. TACAS 2006, pp. 450–454 (2006)
291. Lopez, F., Luck, M., d'Inverno, M.: A normative framework for agent-based systems. Computational and Mathematical Organization Theory **12**, 227–250 (2006)
292. Lowry, M., Boyd, M., Kulkami, D.: Towards a theory for integration of mathematical verification and empirical testing. In: 13th IEEE International Conference on Automated Software Engineering, pp. 322–331 (1998). DOI 10.1109/ASE.1998.732690
293. Luck, M., McBurney, P., Preist, C.: Agent Technology: Enabling Next Generation Computing (A Roadmap for Agent Based Computing). AgentLink (2003)
294. MacKenzie, D.: Knowing Machines: Essays on Technical Change. MIT Press (1996). ISBN 0-262-13315-6
295. MacKenzie, D.: Mechanizing Proof: Computing, Risk, and Trust. MIT Press (2001). ISBN 0-262-13393-8
296. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc., New York, NY, USA (1992)
297. Manna, Z., Pnueli, A.: Verifying Hybrid Systems, vol. 736, pp. 4–35. Springer (1993)
298. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems - Safety. Springer-Verlag (1995)
299. Manzano, M.: Extensions of First Order Logic. Cambridge University Press (1996)
300. Marca, D.: SADT: Structured Analysis and Design Techniques. McGraw-Hill, Cambridge MA (1988)
301. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In: J. Meseguer (ed.) Electronic Notes in Theoretical Computer Science, vol. 4. Elsevier Science Publishers (2000)
302. Maruichi, T., Ichikawa, M., Tokoro, M.: Modelling Autonomous Agents and their Groups. In: Y. Demazeau, J.P. Müller (eds.) Decentralized AI 2 — Proc. 2nd European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW). Elsevier/North Holland (1991)

303. Maskin, E.: Nash equilibrium and welfare optimality. Review of Economic Studies **66**, 23–38 (1999)
304. Maudet, N., Chaib-draa, B.: Commitment-based and dialogue-game based protocols, new trends in agent communication languages. Knowledge Engineering Review **17**(2), 157–179 (2002)
305. Mayer, M.C., Limongelli, C., Orlandini, A., Poggioni, V.: Linear temporal logic as an executable semantics for planning languages. Journal of Logic, Lang and Information **16**, 63–89 (2007)
306. Mazurkiewicz, A.: Trace Theory, in Advances in Petri nets II: applications and relationships to other models of concurrency, *LNCS*, vol. 255. Springer (1987)
307. McBurney, P., Parsons, S.: Games that agents play: A formal framework for dialogues between autonomous agents. Journal of Logic, Language, and Information **11**(3), 315–334 (2002)
308. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: B. Meltzer, D. Michie (eds.) Machine Intelligence 4. Edinburgh University Press (1969)
309. McDermott, D.: A temporal logic for reasoning about processes and plans. Cognitive Science **6**, 101–155 (1982)
310. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
311. Meng, S., Arbab, F.: Web services choreography and orchestration in Reo and constraint automata. In: Y. Cho, R.L. Wainwright, H. Haddad, S.Y. Shin, Y.W. Koo (eds.) SAC, pp. 346–353. ACM (2007)
312. Mermet, B., Simon, G., Zanuttini, B., Saval, A.: Specifying and Verifying a MAS: The Robots on Mars Case Study. In: Post Proceedings ProMAS'07, *Lecture Notes in Artificial Intelligence (LNAI)*, vol. 4908, pp. 172–189 (2008). Detailed model and proofs available at http://users.info.unicaen.fr/~zanutti/data/articles/mssz07companion.pdf
313. Meyer, J.-J. Ch.: A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. Notre Dame Journal of Formal Logic **29**(1), 109–136 (1988)
314. Meyer, J.-J. Ch., van der Hoek, W.: Epistemic Logic for AI and Computer Science. Cambridge: Cambridge University Press (1995)
315. Meyer, J.-J Ch., van der Hoek, W., van Linder, B.: A Logical Approach to the Dynamics of Commitments. Artificial Intelligence **113**(1-2), 1–40 (1999)
316. Milner, R.: Operational and algebraic semantics of concurrent processes. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, pp. 1201–1242. Elsevier, Amsterdam (1990)
317. Misra, J.: A programming model for the orchestration of web services. In: SEFM, pp. 2–11. IEEE Computer Society (2004)
318. Moulin, B.: The social dimension of interactions in multi-agent systems. In: Proceedings of the Workshops on Commonsense Reasoning, Intelligent Agents, and Distributed Artificial Intelligence: Agents and Multi-Agent Systems Formalisms, Methodologies, and Applications, pp. 109 – 123 (1998)
319. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of rabin, mcnaughton and safra. Theor. Comput. Sci. **141**(1-2), 69–107 (1995). DOI http://dx.doi.org/10.1016/0304-3975(94)00214-4
320. Munroe, S., Miller, T., Belecheanu, R., Pechoucek, M., McBurney, P., Luck, M.: Crossing the agent technology chasm: Experiences and challenges in commercial applications of agents. Knowledge Engineering Review **21**(4), 345–392 (2006)
321. Newell, A.: The Knowledge Level. Artificial Intelligence **18**(1), 87–127 (1982)
322. Nguyen, C.D., Perini, A., Tonella, P.: Experimental evaluation of ontology-based test generation for multi-agent systems. In: J.J. Gomez-Sanz, M. Luck (eds.) Ninth International Workshop on Agent-Oriented Software Engineering (AOSE), pp. 165–176 (2008)

323. Nguyen, C.D., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M.: Evolutionary testing of autonomous software agents. In: Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 521–528 (2009)
324. North, D.C.: Institutions, Institutional Change and Economic Performance. Cambridge University Press, Cambridge (1990)
325. Nwana, H.S., Ndumu, D.T., Lee, L.C., Collis, J.C.: ZEUS. a toolkit for building distributed multi-agent systems. Applied Artificial Intelligence Journal **13**(1), 129–185 (1999)
326. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In: C.R. Ramakrishnan, J. Rehof (eds.) TACAS, *Lecture Notes in Computer Science*, vol. 4963, pp. 332–336. Springer (2008)
327. Osborne, M.J., Rubinstein, A.: A Course in Game Theory. MIT Press (1994)
328. van Otterloo, S., van der Hoek, W., Wooldridge, M.: Knowledge as strategic ability. Electronic Lecture Notes in Theoretical Computer Science **85**(2) (2003)
329. Owen, D., Menzies, T.: Lurch: a lightweight alternative to model checking. In: Software Engineering and Knowledge Engineering (SEKE), pp. 158–165 (2003)
330. Owen, D.R.: Combining complementary formal verification strategies to improve performance and accuracy. Ph.D. thesis, West Virginia University, Lane Department of Computer Science and Electrical Engineering (2007)
331. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: Combining specification, proof checking, and model checking. In: R. Alur, T.A. Henzinger (eds.) Computer-Aided Verification, CAV '96, *Lecture Notes in Computer Science*, vol. 1102, pp. 411–414. Springer-Verlag, New Brunswick, NJ (1996)
332. Paulson, L.C.: A Generic Theorem Prover, *Lecture Notes in Computer Science*, vol. 828. Springer-Verlag (1994)
333. Pauly, M.: A modal logic for coalitional power in games. Journal of Logic and Computation **12**(1), 149–166 (2002)
334. Pearson, C.: Numerical Methods in Engineering and Science. CRC Press (1986)
335. Pecheur, C., Raimondi, F.: Symbolic model checking of logics with actions. In: Proc. of Model Checking and Artificial Intelligence, pp. 113–128 (2007)
336. Pednault, E.P.: ADL and the State-Transition Model of Action. Journal of Logic and Computation **4**(5), 467–512 (1994)
337. Penczek, W., Lomuscio, A.: Verifying epistemic properties of multi-agent systems via model checking. Fundamenta Informaticae **55**(2), 167–185 (2003)
338. Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., Cimatti, A.: Formal analysis of hardware requirements. In: Proceedings of the 43rd annual conference on Design automation (DAC '06) (2006)
339. Pinto, J., Reiter, R.: Reasoning about time in the situation calculus. Ann. Math. Artif. Intell **14**, 251–268 (1995)
340. Plotkin, G.D.: A Structural Approach to Operational Semantics. Tech. Rep. DAIMI FN-19, University of Aarhus (1981)
341. Pnueli, A.: The temporal logic of programs. In: Proceedings of FOCS, pp. 46–57 (1977)
342. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 179–190. ACM, New York, NY, USA (1989)
343. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. chap. 6, pp. 149–174
344. Port, R., van Gelder, T. (eds.): Mind as Motion: Explorations in the Dynamics of Cognition. MIT Press (1995)
345. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging multi-agent systems using design artifacts: The case of interaction protocols. In: In Proceedings of AAMAS-02, pp. 960–967 (2002)
346. Poutakidis, D., Padgham, L., Winikoff, M.: An exploration of bugs and debugging in multi-agent systems. In: In Proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS), pp. 628–632. ACM Press (2003)
347. Pratt, V.: Semantical considerations on Floyd-Hoare logic. In: Proc. of the 17th IEEE Symp. on Foundations of Computer Science, pp. 109–121 (1976)

348. Purdy, W.: Fluted formulas and the limits of decidability. Journal of Symbolic Logic **61**, 608–620 (1996)
349. Pěchouček, M., Mařík, V.: Industrial deployment of multi-agent technologies: review and selected case studies. Journal of Autonomous Agents and Multi-Agent Systems **17**, 397–431 (2008). DOI 10.1007/s10458-008-9050-0
350. Pynadath, D.V., Tambe, M.: The Communicative Multiagent Team Decision Problem: Analyzing Teamwork Theories and Models. Journal of Artificial Intelligence Research **16**, 389–423 (2002)
351. Pynadath, D.V., Tambe, M., Chauvat, N., Cavedon, L.: Towards Team-Oriented Programming. In: Intelligent Agents VI — Proc. 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL), *Lecture Notes in Artificial Intelligence*, vol. 1757, pp. 233–247. Springer-Verlag (1999)
352. Raimondi, F.: Model checking multi-agent systems. Ph.D. thesis, University College London (2006)
353. Raimondi, F., Lomuscio, A.: Automatic verification of deontic interpreted systems by model checking via OBDD's. In: R. de Mántaras, L. Saitta (eds.) Proceedings of ECAI, pp. 53–57 (2004)
354. Raimondi, F., Lomuscio, A.: Verification of multiagent systems via ordered binary decision diagrams: an algorithm and its implementation. In: Proc. of the International Joint Conference on AAMAS, pp. 630–637 (2004)
355. Raimondi, F., Lomuscio, A.: Automatic Verification of Multi-agent Systems by Model Checking via Ordered Binary Decision Diagrams. Journal of Applied Logic **5**(2), 235–251 (2007)
356. Rao, A., Georgeff, M.: A model-theoretic approach to the verification of situated reasoning systems. In: Proc. of IJCAI, pp. 318–324 (1993)
357. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: W.V. de Velde, J. Perrame (eds.) Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), pp. 42–55. Springer Verlag, LNAI 1038 (1996)
358. Rao, A.S., Georgeff, M.: BDI Agents: From Theory to Practice. In: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS), pp. 312–319. San Francisco, CA (1995)
359. Rao, A.S., Georgeff, M.P.: Modeling Agents within a BDI-Architecture. In: Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR&R). Morgan Kaufmann, Cambridge, Massachusetts (1991)
360. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: J. Allen, R. Fikes, E. Sandewall (eds.) Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91), pp. 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991)
361. Rao, A.S., Georgeff, M.P.: An Abstract Architecture for Rational Agents. In: Proc. International Conference on Knowledge Representation and Reasoning (KR&R), pp. 439–449 (1992)
362. Rao, A.S., Georgeff, M.P.: Decision Procedures for BDI Logics. Journal of Logic and Computation **8**(3), 293–342 (1998)
363. Reeve, G., Reeves, S.: Experiences using Z animation tools. Working Paper 01/3. Department of Computer Science, University of Waikato (2001). URL http://www.cs.waikato.ac.nz/pubs/wp/2001/
364. Reiter, R.: The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In: Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, pp. 359–380. Academic Press (1991)
365. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge, MA (2001)
366. Ricci, A., Omicini, A., Denti, E.: Activity theory as a framework for mas coordination. In: Procs. of ESAW'02, pp. 96–110 (2002)

367. Ricci, A., Viroli, M., Cimadamore, M.: Prototyping Concurrent Systems with Agents and Artifacts: Framework and Core Calculus. Electronic Notes in Theoretical Computer Science **194**(4), 111–132 (2008)
368. Ricci, A., Viroli, M., Omicini, A.: Give agents their artifacts: the A&A approach for engineering working environments in mas. In: E.H. Durfee, M. Yokoo, M.N. Huhns, O. Shehory (eds.) AAMAS, p. 150. IFAAMAS (2007)
369. Richardson, D.J., Clarke, L.A.: Partition analysis: A method combining testing and verification. IEEE Transactions on Software Engineering **11**(12), 1477–1490 (1985). DOI 10.1109/TSE.1985.231892
370. van Riemsdijk, M., de Boer, F., Dastani, M., Meyer, J.-J. Ch.: Prototyping 3apl in the maude term rewriting language. In: Proceedings of the seventh International Workshop on Computational Logic in Multi-Agent Systems (CLIMA 2006), *LNAI*, vol. 4371. Springer (2007)
371. van Riemsdijk, M.B., de Boer, F.S., Dastani, M., Meyer, J.-J. Ch.: Prototyping 3apl in the maude term rewriting language. In: AAMAS, pp. 1279–1281 (2006)
372. van Riemsdijk, M.B., Meyer, J.-J. Ch., de Boer, F.S.: Semantics of plan revision in intelligent agents. Theoretical Computer Science **351**(2), 240–257 (2006)
373. Rosner, R.: Modular synthesis of reactive systems. Ph.D. thesis, Weizmann Institute of Science (1992)
374. Rouff, C., Hinchey, M., Rash, J., Truszkowski, W., Gordon-Spears, D. (eds.): Agent Technology from a Formal Perspective. NASA Monographs in Systems and Software Engineering. Springer-Verlag, New York, USA (2006)
375. Ruiter, D.: A basic classification of legal institutions. Ratio Juris **10(4)**, 357–371 (1997)
376. Rushby, J.: Analyzing cockpit interfaces using formal methods. In: H. Bowman (ed.) Proceedings of FM-Elsewhere, *Electronic Notes in Theoretical Computer Science*, vol. 43, pp. 1–14. Elsevier, Pisa, Italy (2000). 10.1016/S1571-0661(04)80891-0
377. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering & System Safety **75**(2), 167 – 177 (2002). DOI 10.1016/S0951-8320(01)00092-8
378. Rushby, J.: A safety-case approach for certifying adaptive systems. In: AIAA Infotech@Aerospace Conference (2009)
379. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall (1995)
380. Russell, S., Norvig, P.: Artificial Intelligence. A Modern Approach. Prentice Hall International (2001)
381. Sadri, F., Toni, F., Torroni, P.: Dialogues for negotiation: agent varieties and dialogue sequences. In: Proc. of the International workshop on Agents, Theories, Architectures and Languages, pp. 405–421 (2001)
382. Safra, S.: Complexity of automata on infinite objects. Ph.D. thesis, Rehovot, Israel (1989). URL citeseer.ist.psu.edu/safra89complexity.html
383. Scherl, R.B., Levesque, H.J.: Knowledge, action, and the frame problem. Artificial Intelligence **144**(1–2), 1–39 (2003)
384. Schewe, S.: ATL* satisfiability is 2ExpTime-complete. In: Proceedings of ICALP 2008, *Lecture Notes in Computer Science*, vol. 5126, pp. 373–385. Springer-Verlag (2008)
385. Schild, K.: On the relationship between BDI-logics and standard logics of concurrency. Autonomous agents and multi-agent systems **3**, 259–283 (2000)
386. Schmidt, R.A.: PDL-TABLEAU (2003). http://www.cs.man.ac.uk/~schmidt/pdl-tableau
387. Schnoebelen, P.: The complexity of temporal model checking. In: Advances in Modal Logics, Proceedings of AiML 2002. World Scientific (2003)
388. Schobbens, P.Y.: Alternating-time logic with imperfect recall. Electronic Notes in Theoretical Computer Science **85**(2) (2004)
389. Schurr, N., Marecki, J., Lewis, J.P., Tambe, M., Scerri, P.: The defacto system: Coordinating human-agent teams for the future of disaster response. In: R.H. Bordini, M. Dastani, J. Dix, A.E. Fallah-Seghrouchni (eds.) Multi-Agent Programming: Languages, Platforms and Applications, *Multiagent Systems, Artificial Societies, and Simulated Organizations*, vol. 15, pp. 197–215. Springer (2005)

390. Serbanuta, T.F., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics (extended abstract). Electronic Notes in Theoretical Computer Science **192**(1), 125–141 (2007)
391. Shapiro, S.: Specification and verification of multiagent systems using the Cognitive Agents Specification Language (CASL). Ph.D. thesis, Department of Computer Science, University of Toronto (2005)
392. Shapiro, S., Lespérance, Y.: Modeling multiagent systems with the Cognitive Agents Specification Language — a feature interaction resolution application. In: C. Castelfranchi, Y. Lespérance (eds.) Intelligent Agents Volume VII — Proceedings of the 2000 Workshop on Agent Theories, Architectures, and Languages (ATAL-2000), *LNAI*, vol. 1986, pp. 244–259. Springer-Verlag, Berlin (2001)
393. Shapiro, S., Lesperance, Y., Levesque, H.: The cognitive agent specification language and verification environment. In: this volume, Chapter 9
394. Shapiro, S., Lespérance, Y., Levesque, H.J.: Specifying communicative multi-agent systems. In: W. Wobcke, M. Pagnucco, C. Zhang (eds.) Agents and Multi-Agent Systems — Formalisms, Methodologies, and Applications, *LNAI*, vol. 1441, pp. 1–14. Springer-Verlag, Berlin (1998)
395. Shapiro, S., Lespérance, Y., Levesque, H.J.: The cognitive agents specification language and verification environment for multiagent systems. In: Proceedings of the first international joint conference on autonomous agents and multiagent systems (AAMAS'02), pp. 19–26 (2002)
396. Shapiro, S., Lespérance, Y., Levesque, H.J.: Goal change in the situation calculus. Journal of Logic and Computation **17**(5), 983–1018 (2007)
397. Shapiro, S., Pagnucco, M., Lespérance, Y., Levesque, H.J.: Iterated belief change in the situation calculus. In: A.G. Cohn, F. Giunchiglia, B. Selman (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000), pp. 527–538. Morgan Kaufmann, San Francisco, CA (2000)
398. Sharpanskykh, A., Treur, J.: Verifying interlevel relations within multi-agent systems. In: Proceedings of the 17th European Conference on Artificial Intelligence, ECAI'06, pp. 290–294. IOS Press (2006)
399. Sharpanskykh, A., Treur, J.: Relating cognitive process models to behavioural models of agents. In: Proceedings of the 8th IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT'08, pp. 330–335. IEEE Computer Society Press (2008)
400. Shoham, Y., Tennenholtz, M.: On social laws for artificial agent societies: Off-line design. Artificial Intelligence **73**(1-2), 231–252 (1995)
401. Shoham, Y., Tennenholtz, M.: On the emergence of social conventions: Modeling, analysis and simulations. Artificial Intelligence **94**(1–2), 139–166 (1997)
402. Sierhuis, M.: Modeling and Simulating Work Practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design. Ph.D. thesis, Social Science and Informatics (SWI), University of Amsterdam, SIKS Dissertation Series No. 2001-10, Amsterdam, The Netherlands (2001)
403. Sierhuis, M.: Multiagent Modeling and Simulation in Human-Robot Mission Operations (2006). (http://ic.arc.nasa.gov/ic/publications)
404. Singh, M.: Agent communication languages: rethinking the principles. IEEE Computer **31**(2), 40–47 (1998)
405. Sirbu, M.: Credits and debits on the internet. IEEE Spectrum **34**(2), 23–29 (1997)
406. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. Journal of ACM **32**(3), 733–749 (1985)
407. Smith, R.G., Davis, R.: Frameworks for Cooperation in Distributed Problem Solving. IEEE Transactions on Systems, Man, and Cybernetics **11**(1) (1980)
408. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. In: Proceedings of the International Joint Conference on Theory and Practice of Software Development, pp. 369–383 (1989)
409. Strauss, A.: Negotiations: Varieties, Contexts, Processes and Social Order. San Francisco, Jossey-Bass (1978)

410. Sudeikat, J., Braubach, L., Pokahr, A., Lamersdorf, W., Renz, W.: Validation of BDI agents. ProMAS 2006 pp. 185–200 (2007)
411. Tambe, M.: Teamwork in Real-world Dynamic Environments. In: Proc. 1st International Conference on Multi–Agent Systems (ICMAS). MIT Press (1995)
412. Tennenholtz, M.: On stable social laws and qualitative equilibria. Artificial Intelligence **102**(1), 1–20 (1998)
413. Thiébaux, S., Hoffmann, J., Nebel, B.: In defense of PDDL axioms. Artificial Intelligence **168**, 38–69 (2005)
414. Tidhar, G.: Team-Oriented Programming: Preliminary Report. Tech. Rep. 1993-41, Australian Artificial Intelligence Institute (1993)
415. Tinnemeier, N., Dastani, M., Meyer, J.-J. Ch.: Orwell's nightmare for agents? programming multi-agent organisations. In: Proc. of ProMAS'08 (2008)
416. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages **3**(3), 121–189 (1995)
417. van der Torre, L., Tan, Y.: Diagnosis and decision making in normative reasoning. Artificial Intelligence and Law **7**(1), 51–67 (1999)
418. Treur, J.: Past-future separation and normal forms in temporal predicate logic specifications. Journal of Algorithms in Cognition, Informatics and Logic **64** (2009). Doi http://dx.doi.org/10.1016/j.jalgor.2009.02.008 (in press)
419. Tsai, W.T., Vishnuvajjala, R., Zhang, D.: Verification and validation of knowledge-based systems. IEEE Transactions on Knowledge and Data Engineering **11**(1), 202–212 (1999). DOI 10.1109/69.755629
420. van Ditmarsch, H., Kooi, B., van der Hoek, W.: Dynamic Epistemic Logic, *Synthese Library Series*, vol. 337. Springer (2007)
421. van Riemsdijk, M.B., Astefanoaei, L., de Boer, F.S.: Using the Maude term rewriting language for agent development with formal foundations. In: this volume, Chapter 11
422. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986), pp. 332–344. IEEE Computer Society Press (1986)
423. Vázquez-Salceda, J., Dignum, V., Dignum, F.: Organizing Multiagent Systems. Journal of Autonomous Agents and Multi-Agent Systems **11**(3), 307–360 (2005)
424. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Tech. rep., Universidad Complutense de Madrid, Madrid (2003)
425. Vigueras, G., Botía, J.A.: Tracking causality by visualization of multi-agent interactions using causality graphs. ProMAS 2007 pp. 190–204 (2008)
426. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering **10**(2), 203–232 (2003)
427. Visser, W., Pasareanu, C.S., Khurshid, S.: Test Input Generation with Java PathFinder. In: Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 97–107 (2004)
428. Walther, D., Lutz, C., Wolter, F., Wooldridge, M.: ATL satisfiability is indeed EXPTIME-complete. Journal of Logic and Computation **16**(6), 765–787 (2006)
429. Walton, C.D.: Verifiable Agent Dialogues. Journal of Applied Logic **5**(2), 197–213 (2007)
430. Walton, D.: Model checking agent dialogues. In: Declarative Agent Languages and Technologies, pp. 132–147 (2005)
431. Webster, M.P., Dennis, L.A., Fisher, M.: Model-Checking Auctions, Coalitions and Trust. Tech. Rep. ULCS-09-004, University of Liverpool, Department of Computer Science (2009). `http://www.csc.liv.ac.uk/research/`
432. Wilke, T.: CTL+ is exponentially more succint than CTL. In: Proceedings of FST&TCS '99, *LNCS*, vol. 1738, pp. 110–121 (1999)
433. Winikoff, M.: JACK^{TM} intelligent agents: An industrial strength platform. In: Multi-Agent Programming: Languages, Platforms and Applications. Kluwer (2005)
434. Winikoff, M.: Implementing flexible and robust agent interactions using distributed commitment machines. Multiagent and Grid Systems **2**(4), 365–381 (2006)

435. Winikoff, M.: Implementing Commitment-Based Interactions. In: Proc. 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1–8. ACM, New York, NY, USA (2007)
436. Winikoff, M., Cranefield, S.: On the testability of BDI agent systems. Information Science Discussion Paper Series 2008/03, University of Otago, Dunedin, New Zealand (2008). http://www.business.otago.ac.nz/infosci/pubs/papers/dpsall.htm
437. Winkelhagen, L., Dastani, M., Broersen, J.: Beliefs in agent implementation. In: Proceedings DALT 2005, LNCS 3904. Springer (2006)
438. Wooldridge, M.: Reasoning about Rational Agents. MIT Press (2000)
439. Wooldridge, M.: Introduction to Multiagent Systems. John Wiley & Sons, Inc. (2002)
440. Wooldridge, M., Fisher, M., Huget, M.P., Parsons, S.: Model checking multi-agent systems with MABLE. In: C. Castelfranchi, W.L. Johnson (eds.) Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 952–959. ACM Press (2002)
441. Wooldridge, M., Fisher, M., Huget, M.P., Parsons, S.: Model checking multi-agent systems with MABLE. In: Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 952–959 (2002). DOI 10.1145/544862.544965
442. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A Brief Survey of Program Slicing. SIGSOFT Software Engineering Notes **30**(2), 1–36 (2005)
443. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: Applying event calculus planning using commitments. In: Proceedings of the 1st Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), pp. 527–534 (2002)
444. Young, M., Taylor, R.: Rethinking the taxonomy of fault detection techniques. In: 11th International Conference on Software Engineering, pp. 52–63 (1989)
445. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems. International Journal of Software Engineering and Knowledge Engineering **11**(3), 303–328 (2001)
446. Zhang, D., Cleaveland, R., Stark, E.: The integrated cwb-nc/pioatool for functional verification and performance analysis of concurrent systems. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 431–436 (2003)
447. Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing for agent systems. In: Second International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 10–18 (2007)
448. Zhou, Z.Q., Huang, D., Tse, T., Yang, Z., Huang, H., Chen, T.: Metamorphic testing and its applications. In: Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004) (2004). Published as Hong Kong University (HKU) Computer Science (CS) Technical Report TR-2004-12