Chapter 0

Introduction

Objective

Hardware and software systems are growing rapidly in scale and functionality. From smartcards to air-traffic controllers, computers are being deployed everywhere. As the complexity of a design grows, so does the likelihood of subtle errors it contains. While the complexity of designs has been growing rapidly, the underlying design methodology has evolved only marginally. Consequently, assuring reliability has become one of the key engineering challenges to sustain the ongoing computer revolution. While reliability has always been a concern in system design, some current trends are worth noting:

- Computers are used increasingly in safety-critical applications such as medical treatment and mission control. Presence of bugs in such applications has unacceptable consequences.
- Bugs found at later stages of design can be very expensive, an extreme case of which is illustrated by the notorious floating-point division error in Intel's Pentium processor.
- There is an ever-increasing pressure to reduce time-to-market. This calls for maximum automation of all stages of the design process, including debugging.
- The current trend in design of embedded systems is towards greater use of programmable components. This shifts the focus from low-level optimizations to high-level designs.

Motivated by these concerns, computer-aided verification aims to provide tools that help in detecting logical errors in early stages of design.

Formal verification

Formal methods seek to establish a mathematical proof that a system works correctly. A formal approach provides (1) a modeling language to describe the system, (2) a specification language to describe the correctness requirements, and (3) an analysis technique to verify that the system meets its specification. The model describes the *possible* behaviors of the system, and the specification describes the *desired* behaviors of the system. The statement *the model P satisfies the specification* φ is now a mathematical statement, to be proved or disproved using the analysis technique. The following two are distinguishing hallmarks of formal verification, as opposed to traditional techniques such as testing and simulation:

1. **Formal:** The intuitive correctness claim is made mathematically precise.
2. **Verification:** The goal of the analysis is to prove or disprove the correctness claim. It is not adequate to check a representative sample of possible behaviors as in simulation, rather a guarantee that *all* behaviors satisfy the specification is required.

Automated verification

Primary focus of this book is on analysis that can be performed algorithmically. Typically, such analysis is performed by an exhaustive simulation of the model on all possible inputs. The analysis is performed by a software tool, called a *verifier*, which determines whether the model satisfies the specification, and returns the answer Yes or No along with relevant diagnostic information. While automation has been central to the rising industrial interest in formal verification, automated analysis is not always possible. First, the analysis problem is typically undecidable. For instance, given a C program, determining whether it terminates or not, is a classical undecidable problem. On the other hand, if all the variables of a model are known to be of finite types, then the number of possible states of a system is finite, and the typical analysis problem is decidable. Second, decidability, in itself, does not imply feasibility of analysis. As we shall study, even the simplest analysis problem is computationally hard. In spite of the above mentioned difficulties, there is still a great deal of interesting analysis that can be performed by a verifier. Many recent advances have led to heuristics that make analysis feasible for interesting classes of systems. Furthermore, there is a great deal of flexibility in setting up the analysis problem, and thus, when the original analysis problem is infeasible, it can be simplified so that a verifier can solve it, and still provide useful feedback.

An alternative to automated verification is *interactive verification*. In interactive verification, the analysis problem is formulated as proving a theorem in a mathematical proof system, and the designer attempts to construct the proof of the theorem using a theorem prover as an aid. For instance, one can formulate the correctness of a sorting program using Floyd-Hoare logic, and use a theorem prover such as HOL to prove it. While this approach is very general, it requires considerable expertise, and the involved manual effort has proved to be an obstacle so far.

Reactive systems

The more conventional view of a computer program is the *functional* view: a program computes a function from inputs to outputs. For example, given a list of numbers as the input, a sorting program computes another list of numbers as the output. A more general view is the *reactive* view: a system is interacting with its environment accepting requests and producing responses. For instance, a computer network may be viewed as a reactive system that is accepting packets from different sources and delivering them at different destinations. In the reactive view, the computation of a system may not terminate. Such a reactive view is more appropriate to understand a variety of systems such as an operating system, a microprocessor, a telecommunications switch, and the world-wide web. In this book, we are concerned with modeling, specification, and analysis of reactive systems.

Modeling

The first step in formal verification is to describe the system in the chosen modeling language. A modeling language can be thought as a very high-level concurrent programming language. At this point, it is worthwhile to note the following features of modeling which distinguish it from programming:

- The purpose of a model, in the context of formal verification, is to describe control and interaction. In modeling, we usually avoid describing complex data structures and data manipulation. A modeling language provides a simple set of operations to build complex modules from simpler ones. This simplicity makes analysis more feasible, and helps to focus attention on how the components interact.

- A modeling language typically supports nondeterminism explicitly. For example, a model of a lossy buffer can be described using a choice: when a message is received by the buffer, the message is either discarded or stored. Consequently, unlike a program, a model may exhibit many possible behaviors even after all the inputs are specified.

- In modeling, the designer describes not only the system, but also the *environment* in which the system is supposed to operate. For instance, in a model of a traffic controller, the designer describes the controller together with its environment, namely, cars approaching the intersection from different directions. Explicit modeling of the environment is essential for meaningful analysis of the system. The model of the environment captures the assumptions about the manner in which the system is to be used.

- A system can be modeled at many different levels of abstraction. Thus, the same system may have different models. A model may be incomplete, and can specify only some of the components. Unlike programming, implementability is not the central issue in modeling.

Finite-state machines are widely used to describe control flow. In practice, many extensions of finite-state machines are used. *Extended* finite-state machines support variables. To describe a system consisting of interacting components, finite-state machines need to be equipped with a communication mechanism. For instance, in *Mealy machines* the edges are annotated with input and output symbols. Our modeling language is *reactive modules* which allows description of extended finite-state machines with communication mechanism rich enough to describe different types of interactions.

Requirements

For formal analysis, a designer needs to specify the correctness requirements separately from the model of the system. Requirements capture what the system is intended to do. Typical requirements are of two kinds. The first kind stipulates that the system should always operate within a *safe* set, or something *bad* never happens. For instance, for a traffic controller, the lights for cross-traffic in perpendicular directions should never be green simultaneously. The second kind stipulates that the system discharges its *obligations*, or something *good* will eventually happen. For instance, for a traffic controller, it is essential that a particular light does not stay red forever, and turns green eventually.

Requirements can be expressed formally in a mathematical logic. Pnueli proposed the use of temporal logic to specify requirements concerning behavior of a reactive system over time. Just as a formula of a propositional logic is interpreted over a valuation that assigns truth values to boolean variables, a formula of a temporal logic is interpreted over a sequence (or more generally, a tree) of valuations that capture the truth values of boolean variables at different instances of time. Temporal logic can express many requirements succinctly and naturally, and is well-suited for algorithmic analysis.

For writing specifications, an alternative to temporal logic is automata. In the automata-theoretic approach to verification, the system is viewed as a *generator* of a formal language, the specification is an *acceptor* defining a formal language, and the verification problem reduces to a language inclusion question: whether every behavior generated by the system automaton is accepted by the specification automaton. The theory of *ω-automata*—automata over infinite sequences—provides an elegant conceptual framework to study verification.

Model checking

Model checking means checking that the model satisfies the specification. The analysis is performed algorithmically by searching the state-space of the model. The term was coined by Clarke and Emerson in 1981 in the context of checking a finite model against requirements specified in a temporal logic called *Computation Tree Logic* (CTL). Today the term applies more generally: models need not be finite-state, and requirements can be written in a variety of other languages.

Model checking is computationally expensive even if we restrict attention to simple requirements. The problem is rooted in the fact that the number of states of a system grows exponentially with the number of variables used to describe it. This is the so-called *state-space explosion* problem. We will study a variety of techniques that alleviate this problem. In particular, *symbolic* model checking has proved to be quite effective in analyzing systems of practical interest.

Given a model and a requirement as input, a model checker does not simply answer YES or NO, rather, when the model does not satisfy the requirement, it produces a counter-example, an evidence for the failure. This diagnostic information is extremely useful for debugging purposes. Indeed, model checking is typically an iterative process. The designer starts with a model of the system, checks a variety of requirements, and uses the feedback to modify the model.

Hierarchical verification

In an ideal approach to design, the design begins with a very simple initial model. Model checking is used to debug the model. As the designer gains confidence in its logical correctness, the model is made more complex by adding details, and more requirements can be analyzed. During the refinement step, the designer would like to ensure some consistency between the models before and after adding the details. The relationship between two different models can be made mathematically precise by defining a *refinement* preorder over the set of models. Different formal methods advocate different views regarding when one model should be considered to be a refinement of another, that is, they differ in the semantics of the refinement relation. The *refinement checking* problem, then, is to verify that a detailed model is a refinement of the abstract model. Observe that refinement checking is like model checking where the same language is used to describe the model as well as the requirement. A hierarchical approach to design and analysis corresponds to constructing models at several different levels of abstraction, and establishing each level to be a refinement of the next higher level.

The problem of refinement checking, while decidable for finite-state systems, is computationally intractable. However, the designer's intuition regarding the correspondence between the abstract and detailed model can be exploited to establish the refinement claim. Of particular interest to us will be the *compositional* methods in which the structure of the descriptions of the two models is used to decompose the required refinement claim into subclaims regarding refinement relationships among components of the two models.

Hardware verification

Model checking has been most successful in hardware verification. In 1992, the model checker SMV was used to pinpoint logical errors in the cache coherence protocol described in the IEEE Futurebus+ Standard. This, and numerous subsequent, case-studies attracted attention of hardware industry eager to enhance capabilities of design automation tools. Model checking seems suitable to debug intricate aspects of microprocessor designs. Today semiconductor companies such as Lucent, IBM, Intel, Motorola, and Siemens, have internal verification groups aimed at integrating formal verification in design flow, while CAD-tool vendors such as Cadence and Synopsis are exploring ways to add verification capability to design tools. Some verification tools are already commercially available, for instance, *FormalCheck* of Lucent Technologies.

An important reason for the success of model checking in hardware verification is the ease with which it fits into the existing design methodology. Hardware description languages such as VHDL and Verilog are extremely popular, and designers routinely use simulation and synthesis tools available for these languages. While these languages have not been designed with computer-aided verification in mind, a significant subset can be subjected to analysis.

Software verification

High-level design is somewhat uncommon in software development, and consequently, model checking has had limited influence in this domain. An important application has been design of protocols used in safety-critical applications such as aviation. Unlike in hardware design, there is no commonly accepted standard language that has precise mathematical semantics and is amenable to analysis. Increasing popularity of synchronous languages such as Esterel and Lustre, and of *Statecharts*, a graphical formalism to describe hierarchically structured state machines, suggest a promising future.

Recently, a model checking effort at Microsoft Research to check C source code has been successful. The tool, which is being commercialized, checks if third-party driver software conforms to the Windows interfacing requirements. This shows that in certain control-intensive applications, model checking can be very effective also in the software domain.

Limitations

Computer-aided verification is only one of the many weapons available to a designer to ensure reliability. Let us note a few limitations to understand what it can do and what it cannot do.

- The application domain of computer-aided verification is control-intensive concurrent systems. It seems unsuitable to analyze, for instance, a database query manager, or a word processor.
- A model checker analyzes a model, and not the system itself. Even when the modeling language coincides with the implementation language (e.g. as in hardware design using VHDL), to make analysis feasible, a variety of simplifications are used. Consequently, the gap between the model and its implementation remains. This implies that the greatest strength of computer-aided verification is detecting bugs, rather than certifying absence of bugs.
- Formalizing requirements is a challenging task common to all formal methods. The designer can enumerate several requirements that the system is supposed to satisfy, but usually cannot be sure that the list is complete. The gap between the intuitive understanding of correctness and its formalization in the specification language reasserts the applicability of model checking as a *falsification*, rather than certification, tool.
- With new heuristics, and with increasing speed and available memory on modern computers, we can hope to apply computer-aided verification to analyze systems beyond the scope of today's tools. However, the high computational complexity of the analysis problem is, and will remain, a hurdle.

Contents

Chapter 1

Reactive Modules

This chapter introduces a *modeling language*, RML (Reactive Module Language), for describing the architecture and behavior of hardware and software systems. Modeling languages are to be used by hardware designers, software engineers, and CAD tools during the early stages of the design process for describing and, more importantly, analyzing blueprints of a system. Thus, unlike an implementation language such as a hardware description language or a programming language, a modeling language need not provide extensive mechanisms for structuring control flow and manipulating data. Rather, a modeling language must have the following four essential characteristics. First, it must facilitate high-level, partial system descriptions by supporting nondeterminism. Second, it must facilitate the description of interactions between systems and system components by supporting concurrency. Third, it must facilitate the rapid prototyping and simulation of system descriptions by supporting an execution model. Fourth, it must facilitate the formal analysis of system behaviors by supporting a precise mathematical semantics. The first and fourth characteristics distinguish RML from many common implementation languages; the second and third characteristics distinguish RML from many common requirements specification languages.

1.1 Definition of Reactive Modules

We model systems as reactive modules. Reactive modules resemble molecules, in that they interact with each other to form composite objects. Reactive modules are built from atoms, and atoms are built from variables —the elementary particles of systems. Our presentation proceeds bottom-up, from variables to atoms, modules, and the composition of modules.

1.1.1 Variables

We consider systems that are *discrete*, *deadlock-free*, and *nondeterministic*. A discrete system is a collection of variables that, over time, change their values in a sequence of rounds. The first round is an *initialization round*, and all subsequent rounds are *update rounds*. In the initialization round, the values of all variables are initialized, and in every update round, the values of all variables are updated. Deadlock-freedom means that in the initialization round, there is at least one option for initializing each variable, and in every update round, there is at least one option for updating each variable. Consequently, every update round can be followed by another update round. Nondeterminism means that in the initialization round, there may be several options for initializing a variable, and in an update round, there may be several options for updating a variable. Consequently, two exact copies of a system can, over time, exhibit very different behaviors.

Initial commands and update commands

We define the behavior of a variable using two guarded commands —an *initial command* and an *update command*. While unprimed symbols, such as x, refer to the value of a variable at the beginning of a round, primed symbols, such as x', refer to the value of the same variable at the end of a round. A value of x at the end of the initialization round is called an *initial value* of x. The possible initial values for a variable are defined by an initial command. For example, the initial command

```
**init**
   ⫿ $true \rightarrow x' := 0$
   ⫿ $true \rightarrow x' := 1$
```

asserts that 0 and 1 are the possible initial values of x. A value of x at the beginning of an update round is called a *current value* of x, and a value of x at the end of an update round is called a *next value* of x. In every update round, the possible next values for x may depend on the current value of x, and on the current values of other variables. The possible next values for a variable are defined by an update command. For example, the update command

```
update
  ‖ $y = 0 \rightarrow x' := x + 1$
  ‖ $y \neq 0 \rightarrow x' := x - 1$
  ‖ $\mathit{true} \;\; \rightarrow x' := x$
```

asserts that in every update round, if the current value of y is 0, then the value of x is either incremented or stays unchanged, and if the current value of y is different from 0, then the value of x is either decremented or stays unchanged.

Example 1.1 [Scheduler] Consider a scheduler that, in every round, assigns a processor to one of two tasks. The nonnegative-integer variable task_1 indicates the amount of processor time, measured in rounds, which is necessary to complete the first task. Similarly, the nonnegative-integer variable task_2 indicates the amount of processor time which is necessary to complete the second task. The ternary variable *proc* has the value 0 if in the most recent round, the processor has been idle; *proc* has the value 1 if the processor has been assigned to the first task; and *proc* has the value 2 if the processor has been assigned to the second task. The initial command

```
init
  ‖ $\mathit{true} \rightarrow \mathit{proc}' := 0$
```

asserts that initially the processor is idle. The update command

```
update
  ‖ $\mathit{task}_1 = 0 \wedge \mathit{task}_2 = 0 \rightarrow \mathit{proc}' := 0$
  ‖ $\mathit{task}_1 > 0$                    $\rightarrow \mathit{proc}' := 1$
  ‖ $\mathit{task}_1 = 0 \wedge \mathit{task}_2 > 0 \rightarrow \mathit{proc}' := 2$
```

asserts that the scheduler always gives priority to the first task. ■

Simultaneous updates

Within a round, some variables are initialized or updated simultaneously, and some variables are initialized or updated sequentially. We insist that if two variables x and y are initialized or updated simultaneously in some round, then x and y are initialized and updated simultaneously in every round. A set of variables that are initialized simultaneously in the initialization round, and updated simultaneously in every update round, are said to form an *atom.* It is often convenient to name atoms, in which case the constituent variables of an atom are referred to as the variables that are *controlled* by the atom. The behaviors of all variables that are controlled by one atom are defined using a single initial command and a single update command. For example, for the atom controlling the two variables x and y, the initial command

```
init
  ‖ $\mathit{true} \rightarrow x' := 0;\; y' := 1$
  ‖ $\mathit{true} \rightarrow x' := 1;\; y' := 0$
```

asserts that the possible initial values of x and y are 0 and 1, and that the initial values of x and y are different. For the same atom, the update command

update
$[\!]\ x \leq y \rightarrow x' := x+1;\ y' := y-1$
$[\!]\ x \geq y \rightarrow x' := x-1;\ y' := y+1$

asserts that in every update round, depending on the current values of x and y, either x is incremented and y is decremented, or vice versa.

Example 1.2 [Scheduler] Consider a scheduler similar to Example 1.1, except that the scheduler alternates priorities between both tasks. If in a given round the processor is assigned to the first task, then in the next round the second task is given priority over the first task, and vice versa. The binary variable *prior* indicates which of the two tasks will be given priority in the upcoming round. The variables *proc* and *prior* form an atom; that is, the processor assignment and the priority information are updated simultaneously. Assuming that initially either task may be given priority, we have the initial command

init
$[\!]\ \mathit{true} \rightarrow \mathit{proc}' := 0;\ \mathit{prior}' := 1$
$[\!]\ \mathit{true} \rightarrow \mathit{proc}' := 0;\ \mathit{prior}' := 2.$

Assuming that a task retains priority until it is given the processor, we have the update command

update
$[\!]\ \mathit{task}_1 = 0 \wedge \mathit{task}_2 = 0 \rightarrow \mathit{proc}' := 0$
$[\!]\ \mathit{prior} = 1 \wedge \mathit{task}_1 > 0 \rightarrow \mathit{proc}' := 1;\ \mathit{prior}' := 2$
$[\!]\ \mathit{prior} = 1 \wedge \mathit{task}_1 = 0 \wedge \mathit{task}_2 > 0 \rightarrow \mathit{proc}' := 2$
$[\!]\ \mathit{prior} = 2 \wedge \mathit{task}_2 > 0 \rightarrow \mathit{proc}' := 2;\ \mathit{prior}' := 1$
$[\!]\ \mathit{prior} = 2 \wedge \mathit{task}_2 = 0 \wedge \mathit{task}_1 > 0 \rightarrow \mathit{proc}' := 1.$

Note that in the first, third, and fifth guarded assignments, the value of the variable *prior* stays unchanged; that is, $\mathit{prior}' := \mathit{prior}$. ■

Sequential updates

We insist that if a variable x is initialized or updated before a variable z in some round, then x is initialized and updated before z in every round. If x is initialized before z in the initialization round, and x is updated before z in every update round, then the variable x is said to be *awaited* by the variable z. If z awaits x, then the possible initial values for z may depend on the initial value of x, and in every update round, the possible next values for z may depend on the next value of x. For example, assuming that z awaits both x and y, the initial command

init
$[\!]\ \mathit{true} \rightarrow z' := x' + y'$

asserts that the initial value of z is equal to the sum of the initial values of x and y, and the update command

```
**update**
  $\| \; y' = 0 \rightarrow z' := x$
  $\| \; y' \neq 0 \rightarrow z' := x'$
```

asserts that in every update round, if the next value of y is 0, then the next value of z is equal to the current value of x, and otherwise the next value of z is equal to the next value of x.

Example 1.3 [Scheduler] In every update round of the scheduler example, the indicators $task_1$ and $task_2$ for pending work are updated after the processor is assigned to one of the two tasks; that is, both indicators await the processor assignment *proc*. The indicator $task_1$ is decremented in every round in which the processor is assigned to the first task, and the indicator $task_2$ is decremented in every round in which the processor is assigned to the second task. In addition, new work for a task may arrive in any round, and it arrives in blocks of 5 units. The nonnegative-integer variable new_1 indicates the amount of new work that has arrived in the most recent round for the first task, and the nonnegative-integer variable new_2 indicates the amount of new work that has arrived for the second task. The initial command and the update command for new_1 are identical, and we write

```
**initupdate**
  $\| \; true \rightarrow new'_1 := 0$
  $\| \; true \rightarrow new'_1 := 5$
```

to avoid duplication. The variable $task_1$ awaits both new_1 and $proc_1$, and its behavior is defined by the commands

```
**init**
  $\| \; true \rightarrow task'_1 := new'_1$
**update**
  $\| \; proc' = 1 \rightarrow task'_1 := task_1 + new'_1 - 1$
  $\| \; proc' \neq 1 \rightarrow task'_1 := task_1 + new'_1.$
```

The behaviors of the variables new_2 and $task_2$ are defined by similar commands.
■

1.1.2 Atoms

The initialization round and every update round consist of several subrounds, one for each atom. For an atom U, in the U-subround of the initialization round, the controlled variables of U are initialized simultaneously, as defined by the initial command of U. In the U-subround of an update round, the controlled variables of U are updated simultaneously, as defined by the update command of U. If the possible next values for some controlled variable of U depend on the current value of a variable x, then x is said to be *read* by the atom U. Read variables occur in the update command of U as unprimed symbols. If the possible initial values for some controlled variable of U depend on the initial value of x, or if the possible next values for some controlled variable of U depend on the next value of x, then the variable x is *awaited* by the atom U. Awaited variables occur in the initial and update commands of U as primed symbols. A variable can be both read and awaited by an atom, or read and controlled, but for obvious reasons, a variable cannot be awaited and controlled.

ATOM

Let X be a finite set of typed variables. An X*-atom* U consists of an atom declaration and an atom body. The declaration of U consists of a nonempty set $\mathsf{ctr}X_U \subseteq X$ of *controlled variables*, a set $\mathsf{read}X_U \subseteq X$ of *read variables*, and a set $\mathsf{await}X_U \subseteq X\backslash\mathsf{ctr}X_U$ of *awaited variables.* The body of U consists of an *initial command* init_U and an *update command* update_U. The initial command init_U is a guarded command from $\mathsf{await}X'_U$ to $\mathsf{ctr}X'_U$. The update command update_U is a guarded command from $\mathsf{read}X_U \cup \mathsf{await}X'_U$ to $\mathsf{ctr}X'_U$.

Remark 1.1 [Atom variables] All variables of an X-atom are taken from the underlying set X of variables. If X and Y are two sets of variables with $X \subseteq Y$, then every X-atom is also a Y-atom. ■

An important special case of atoms are the *deterministic* atoms. For a deterministic atom, the initial values of all controlled variables are uniquely determined by the initial values of the awaited variables, and in every update round, the next values of all controlled variables are uniquely determined by the current values of the read variables and the next values of the awaited variables.

DETERMINISTIC ATOM

The atom U is *deterministic* if both the initial command init_U and the update command update_U are deterministic. Otherwise, U is a *nondeterministic* atom.

The consistency of atoms

We insist on two consistency requirements for atoms, which ensure that a collection of variables that are controlled by several atoms can be initialized and updated unambiguously. First, we require that no variable be controlled by more than one atom. This prevents the assignment of multiple, inconsistent values to a variable. Second, we require that there be no circular await dependencies between variables. The await dependencies between the controlled variables and the awaited variables of an atom constrain the possible temporal orderings of the subrounds within a round. We allow only await dependencies that permit, within every round, at least one possible ordering of the subrounds. Consider two variables, x and y, which are controlled, respectively, by the two atoms U_x and U_y. If x is an awaited variable of U_y, then the initial value of y may depend on the initial value of x, and the next value of y may depend on the next value of x. Therefore x must be initialized and updated before y; that is, the U_x-subround must go before the U_y-subround in the initialization round and in every update round. It follows that y must not be an awaited variable of U_x. More generally, the atom U_1 *precedes* the atom U_n if there is a chain $U_2, \ldots, U_{n-1}$ of atoms such that for all $2 \leq i \leq n$, some controlled variable of U_{i-1} is an awaited variable of U_i. If U precedes V, then the U-subround must go before the V-subround in every round. Therefore it must not happen that U precedes V and V precedes U.

Atom consistency

Let X be a finite set of typed variables, let $\mathcal{U}$ be a set of X-atoms, and let x and y be two variables in X. The variable y *awaits* the variable x, written $x \prec_{\mathcal{U}} y$, if some atom in $\mathcal{U}$ controls y and awaits x. The set $\mathcal{U}$ is *consistent* if (1) no variable is controlled by more than one atom in $\mathcal{U}$, and (2) the transitive closure $\prec_{\mathcal{U}}^{+}$ of the await relation on the variables in X is asymmetric. Given two atoms U and V in $\mathcal{U}$, the atom U *precedes* the atom V, written $U \prec\!\!\prec_{\mathcal{U}} V$, if there is a variable x controlled by U and a variable y controlled by V such that $x \prec_{\mathcal{U}}^{+} y$.

Proposition 1.1 [Partial order of atoms] *If $\mathcal{U}$ is a consistent set of atoms, then the precedence relation $\prec\!\!\prec_{\mathcal{U}}$ is a strict partial order on $\mathcal{U}$.*

Exercise 1.1 {T2} [Proof of Proposition 1.1] (a) Prove Proposition 1.1. (b) Show that the definition of consistency cannot be relaxed; that is, prove that if the precedence relation $\prec\!\!\prec_{\mathcal{U}}$ of a set $\mathcal{U}$ of atoms is asymmetric, then condition (2) for the consistency of $\mathcal{U}$ is satisfied. ■

The execution of atoms

For a consistent set $\mathcal{U}$ of atoms, the linearizations of the partial order $\prec\!\!\prec_{\mathcal{U}}$ determine the possible sequences of subrounds within a round. These linearizations are called the *execution orders* for $\mathcal{U}$. Every consistent set of atoms has at least one execution order, and possibly several.

Execution order

Let X be a finite set of typed variables, and let $\mathcal{U}$ be a consistent set of X-atoms. An *execution order* for $\mathcal{U}$ is a sequence $U_1, \ldots, U_n$ of the atoms in $\mathcal{U}$ which does not violate the precedence relation $\prec\!\!\prec_{\mathcal{U}}$; that is, for all $1 \leq i, j \leq n$, if $U_i \prec\!\!\prec_{\mathcal{U}} U_j$, then $i < j$.

A system is *closed* if it controls the behavior of all its variables. We model a closed system with the set X of variables as a consistent set $\mathcal{U}$ of X-atoms so that each variable in X is controlled by some atom in $\mathcal{U}$. Such a set $\mathcal{U}$ of atoms is *executed* by carrying out, in the initialization round, all initial commands of $\mathcal{U}$ in some execution order, and by carrying out, in every update round, all update commands of $\mathcal{U}$ in some execution order. The outcome of the execution is called an *initialized trajectory* of $\mathcal{U}$: it gives, for each variable in X, a sequence of values, one for every round. If $\mathcal{U}$ contains some nondeterministic atoms, then for any given number of rounds, there can be many initialized trajectories. By contrast, the fact that $\mathcal{U}$ may have several execution orders does by itself not give rise to multiple trajectories. In particular, if all atoms in $\mathcal{U}$ are deterministic, then for any given number of rounds, there is a unique initialized trajectory.

Exercise 1.2 {`T2`} [Execution of atoms] Let X be a finite set of typed variables, and let $\mathcal{U}$ be a consistent set of X-atoms so that each variable in X is controlled by some atom in $\mathcal{U}$. Show that the possible outcomes of executing $\mathcal{U}$ do not depend on the execution orders that are chosen during the execution of $\mathcal{U}$; that is, prove that every initialized trajectory of $\mathcal{U}$ can be obtained by choosing an arbitrary execution order for $\mathcal{U}$ and maintaining the chosen execution order in every round. ■

Example 1.4 [Scheduler] The scheduler from Example 1.3 is a closed system with six variables, new_1, new_2, $task_1$, $task_2$, $proc$, and $prior$, which are arranged in the five atoms $A1$–$A5$ shown in Figure 1.1. In RML, each atom is written as an atom name followed by an atom declaration and an atom body. In atom declarations, the keywords **reads** or **awaits** are omitted if the sets of read or awaited variables are empty. The atoms $A3$ and $A4$ are deterministic; the atoms $A1$, $A2$, and $A5$ are nondeterministic. The set $\{A1, \ldots, A5\}$ of atoms is consistent, because $new_1 \prec task_1$, $proc \prec task_1$, $new_2 \prec task_2$, and $proc \prec task_2$ are the only await dependencies. There are many execution orders, including $A1, A2, A5, A3, A4$ and $A5, A2, A4, A1, A3$. Figure 1.2 shows two, arbitrarily chosen, initialized trajectories with 15 update rounds each. The first of these two trajectories is depicted graphically, in the form of a so-called *timing diagram*, in Figure 1.3. The vertical dotted lines of the timing diagram represent boundaries between rounds. Note, for instance, that the variable $task_1$ changes its initial value, in the first update round, only after both new_1 and $proc$ have changed their initial values. ■

Combinational and sequential atoms

In the initialization round, the possible initial values for the controlled variables of an atom depend, in some way, on the initial values of the awaited variables. If in every update round, the possible next values for the controlled variables depend in the same way on the next values of the awaited variables, then the

```
**atom** $A1$ **controls** $new_1$
  **initupdate**
    $\|$ $true \rightarrow new'_1 := 0$
    $\|$ $true \rightarrow new'_1 := 5$

**atom** $A2$ **controls** $new_2$
  **initupdate**
    $\|$ $true \rightarrow new'_2 := 0$
    $\|$ $true \rightarrow new'_2 := 5$

**atom** $A3$ **controls** $task_1$ **reads** $task_1$ **awaits** $new_1, proc$
  **init**
    $\|$ $true \rightarrow task'_1 := new'_1$
  **update**
    $\|$ $proc' = 1 \rightarrow task'_1 := task_1 + new'_1 - 1$
    $\|$ $proc' \neq 1 \rightarrow task'_1 := task_1 + new'_1$

**atom** $A4$ **controls** $task_2$ **reads** $task_2$ **awaits** $new_2, proc$
  **init**
    $\|$ $true \rightarrow task'_2 := new'_2$
  **update**
    $\|$ $proc' = 2 \rightarrow task'_2 := task_2 + new'_2 - 1$
    $\|$ $proc' \neq 2 \rightarrow task'_2 := task_2 + new'_2$

**atom** $A5$ **controls** $proc, prior$ **reads** $task_1, task_2, prior$
  **init**
    $\|$ $true \rightarrow proc' := 0;\ prior' := 1$
    $\|$ $true \rightarrow proc' := 0;\ prior' := 2$
  **update**
    $\|$ $task_1 = 0 \wedge task_2 = 0$                     $\rightarrow proc' := 0$
    $\|$ $prior = 1 \wedge task_1 > 0$                      $\rightarrow proc' := 1;\ prior' := 2$
    $\|$ $prior = 1 \wedge task_1 = 0 \wedge task_2 > 0$    $\rightarrow proc' := 2$
    $\|$ $prior = 2 \wedge task_2 > 0$                      $\rightarrow proc' := 2;\ prior' := 1$
    $\|$ $prior = 2 \wedge task_2 = 0 \wedge task_1 > 0$    $\rightarrow proc' := 1$
```

Figure 1.1: The five scheduler atoms

new_1	5	0	0	0	0	0	0	0	0	0	5	0	0	0	5
new_2	0	0	0	0	5	0	0	0	0	0	0	5	0	0	0
$task_1$	5	4	3	2	2	1	1	0	0	0	4	4	3	3	7
$task_2$	0	0	0	0	4	4	3	3	2	1	1	5	5	4	4
proc	0	1	1	1	2	1	2	1	2	2	1	2	1	2	1
prior	1	2	2	2	1	2	1	2	1	1	2	1	2	1	2

new_1	5	0	0	0	0	0	0	0	0	0	0	0	5	0	0
new_2	5	0	0	0	0	0	0	0	0	0	0	0	5	0	0
$task_1$	5	5	4	4	3	3	2	2	1	1	0	0	4	4	3
$task_2$	5	4	4	3	3	2	2	1	1	0	0	0	5	4	4
proc	0	2	1	2	1	2	1	2	1	2	0	0	1	2	1
prior	2	1	2	1	2	1	2	1	2	1	1	1	2	1	2

Figure 1.2: Two initialized trajectories of the scheduler

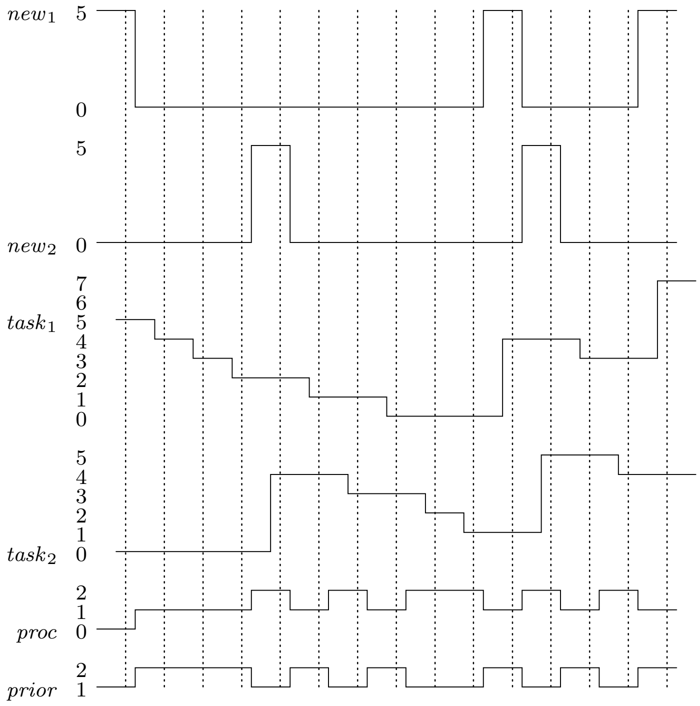


Figure 1.3: The timing diagram for the first trajectory from above

atom is called *combinational*. Thus, a combinational atom is a (generally nondeterministic) function that, given the values of the awaited variables at the end of an initialization or update round, computes the possible values for the controlled variables at the end of the round. In particular, for combinational atoms, the possible next values of some controlled variable x cannot depend on the current value of any variable, including x itself. By contrast, atoms that distinguish between initial and update rounds are called *sequential*.

Combinational vs. sequential atom

An atom U is *combinational* if (1) the set $\mathsf{read}X_U$ of read variables is empty, and (2) the initial command init_U and update command update_U are identical. Otherwise, U is a *sequential* atom.

Example 1.5 [Zero-delay vs. unit-delay copying] Given two variables y and x of the same type, we want y to duplicate the behavior of x. The combinational atom

```
atom $CombCopy$ controls $y$ awaits $x$
  initupdate
    $[\!]$ $true \to y' := x'$
```

copies the value of x into y without delay. In the initial round, the atom waits for x being initialized, and assigns the initial value of x to y. In every update round, the atom waits for x being updated, and assigns the next value of x to y. Consequently, both y and x have the same value at the end of every round. The sequential atom

```
atom $SeqCopy$ controls $y$ reads $x$
  update
    $[\!]$ $true \to y' := x$
```

copies the value of x into y with a delay of one round (the initial command is irrelevant for the purposes of this example). In every update round, the atom assigns the current value of x to y. Consequently, the value of y at the end of every update round is the same as the value of x at the beginning of the round. In Rml, combinational atoms can be recognized by the keyword **initupdate**. For example, the atoms $A1$ and $A2$ from Example 1.4 are combinational. ■

Lazy and eager atoms

An atom *sleeps* in an update round if the values of all controlled variables stay unchanged. An atom that may sleep in every update round is called *lazy*. The progress of a lazy atom cannot be enforced, because the atom may put off the next modification of the controlled variables for any number of rounds. By contrast, if certain current values of read variables or next values of awaited variables force an immediate change in value for a controlled variable, then the atom is called *eager*.

Lazy vs. eager atom

Given a finite set X of typed variables, the *sleep assignment for* X is the guarded assignment γ from X to X' with the guard $p_\gamma = \mathit{true}$ and the assignment $e_\gamma^{x'} = x$ for each variable $x' \in X'$. An atom U is *lazy* if the update command update_U contains the sleep assignment for the set $\mathsf{ctr}X_U$ of controlled variables. Otherwise, U is an *eager* atom.

Example 1.6 [Continuous vs. occasional copying] Both atoms *CombCopy* and *SeqCopy* from Example 1.5 are eager. In the first case, all modifications of y follow immediately, within the same round, the corresponding modifications of x; in the second case, the modifications of y are delayed by exactly one round. By contrast, the lazy atom

```
**atom** *LazyCopy* **controls** $y$ **reads** $y$ **awaits** $x$
  **update**
    $[\!]$ $\mathit{true} \to y' := x'$
    $[\!]$ $\mathit{true} \to y' := y$
```

copies the value of x into y at arbitrary times (the initial command is irrelevant for the purposes of this example). In every update round, either the value of y stays unchanged, or it is updated to the next value of x. Consequently, some values of x may not be copied into y. In Rml, the atom prefix **lazy** can be used instead of the sleep assignment for the update command. For example, the atom *LazyCopy* can alternatively be specified as

```
**lazy atom** *LazyCopy* **controls** $y$ **reads** $y$ **awaits** $x$
  **update**
    $[\!]$ $\mathit{true} \to y' := x'$.
```

Note that the variable y is read, even though, because the keyword **lazy** is used, y does not literally occur in the update command as an unprimed symbol. ■

Remark 1.2 [Lazy implies sequential and, mostly, nondeterministic] Every lazy atom U reads its controlled variables; that is, $\mathsf{ctr}X_U \subseteq \mathsf{read}X_U$. It follows that all lazy atoms are sequential. Furthermore, with the exception of (trivial) atoms whose update commands contain only the sleep assignment, lazy atoms are nondeterministic. ■

Passive and active atoms

During an update round, an atom can notice changes in the values of awaited variables. If the awaited variable is also read, then the atom can directly compare the current value with the next value. If the awaited variable is not read, then the atom can remember the next value from the previous round, by storing it in a controlled variable, and compare it with the next value from the present round. Therefore, every change in the value of an awaited variable is an *observable event.* An atom is called *passive* if it may sleep in every update round in which no observable event occurs; that is, the atom may sleep whenever the values of all awaited variables stay unchanged. By contrast, if the value of a controlled variable is changed in certain update rounds independent of observable events, then the atom is called *active.* The active atoms are round-driven and the passive atoms are event-driven: while the progress of an active atom can be enforced by the expiration of rounds, the progress of a passive atom can be enforced only by other atoms that modify awaited variables.

PASSIVE VS. ACTIVE ATOM

Given two finite sets X and Y of typed variables, the *conditional sleep assignment for X with respect to Y* is the guarded assignment γ from $X \cup Y \cup Y'$ to X' with the guard $p_\gamma = (Y' = Y)$ and the assignment $e_\gamma^{x'} = x$ for each variable $x' \in X'$. An atom U is *passive* if U is either combinational, or lazy, or the update command update_U contains the conditional sleep assignment for the set $\mathsf{ctr}X_U$ of controlled variables with respect to the set $\mathsf{await}X_U$ of awaited variables. Otherwise, U is an *active* atom.

Remark 1.3 [Passive includes combinational and lazy] By definition, all combinational and lazy atoms are passive. This reflects the fact that conditional sleep assignments are redundant for combinational and lazy atoms: if the conditional sleep assignment is added to the update command of a combinational or lazy atom, then the behavior of the atom remains the same. For a lazy atom, this is trivially so. For a combinational atom, this is because if the values of the awaited variables do not change in an update round, then the atom may compute the same next values for the controlled variables as in the previous round. ■

Example 1.7 [Round vs. event counting] If the behavior of the nonnegative-integer variable n is defined by the active atom

```
atom $ActiveCount$ controls $n$ reads $n$
  init
    $[\!] \; true \rightarrow n' := 0$
  update
    $[\!] \; true \rightarrow n' := n + 1$
```

then the value of n at the end of the i-th update round is i. Thus, active atoms can count the number of rounds that expire. For example, an active atom may count the number of rounds that expire between two consecutive changes in the value of a variable x that is controlled by another atom. By contrast, passive atoms do not have the ability to count rounds; they can count only the number of observable events, such as the number of changes in the value of x. Specifically, if the behavior of n is defined by the passive atom

```
**atom** $PassiveCount$ **controls** $n$ **reads** $n, x$ **awaits** $x$
  **init**
    $[\!]\ true \rightarrow n' := 0$
  **update**
    $[\!]\ x' \neq x \rightarrow n' := n + 1$
    $[\!]\ x' = x \rightarrow n' := n$
```

then the value of n at the end of the i-th update round is $j \leq i$, where j is the number of times that the value of x has changed during the first i update rounds. In Rml, the atom prefix **passive** can be used instead of the conditional sleep assignment for the update command. The atom *PassiveCount* is not a good example for illustrating the use of the keyword **passive**, however, because the conditional sleep assignment $x' = x \rightarrow n' := n$ coincides with the default assignment of the update command (if all guards are false, then the controlled variables stay unchanged), and therefore can be omitted with or without prefixing the atom description. A better example for the use of the keyword **passive** will follow in the next section. Now, for the record: the atoms *ActiveCount* (trivially) and *PassiveCount* are both sequential and eager. The combinational and lazy copiers *CombCopy* and *LazyCopy* from Examples 1.5 and 1.6 are passive (trivially), and the sequential, eager copier *SeqCopy* is active. This is because *SeqCopy* needs to be sensitive to the expiration of rounds in order to delay copying by exactly one round. ■

Remark 1.4 [Classification of atoms] The atoms can be partitioned into four pairwise disjoint classes: the combinational atoms, the lazy atoms, the active atoms, and the atoms that are sequential, eager, and passive. ■

1.1.3 Modules

A *reactive module* is a system, or system component, that interacts with other systems, or other components, which, collectively, make up the *environment* of the module. The behavior of some variables is controlled by the module, and the behavior of other variables is controlled by the environment. We refer to the former as the *controlled variables* of the module, and to the latter as the *environment variables.* The controlled variables are partitioned into atoms, and so are the environment variables. In the initialization round and in every update round, the module and the environment take turns in the form of subrounds. In each subround of the initialization round, either the module initializes an atom of controlled variables, or the environment initializes an atom of environment variables. In each subround of an update round, either the module updates an atom of controlled variables, or the environment updates an atom of environment variables. Deadlock-freedom requires that, in the initialization round, the module is prepared to initialize its variables for all possible initial values of the environment variables, and in every update round, the module is prepared to update its variables for all possible current and next values of the environment variables.

In addition to being partitioned into atoms, the controlled variables are classified as to whether or not their values can be observed by the environment, and the environment variables are classified as to whether or not their values can be observed by the module. If a controlled variable is visible to the environment, then the updating of the environment variables may depend on the values of the controlled variable. Symmetrically, if an environment variable is visible to the module, then the updating of the controlled variables may depend on the values of the environment variable. Thus, a module description refers to three classes of variables —private, interface, and external. Each *private variable* can be read and modified by the module, and neither read nor modified by the environment. Each *interface variable* can be read by both the module and the environment, and modified by the module only. Each *external variable* can be read by both the module and the environment, and modified by the environment only. In other words: the module controls the private variables and the interface variables; the environment observes the interface variables and the external variables. The fourth class of variables —environment variables that are not visible to the module— is, naturally, not part of the module description.

MODULE

A (*reactive*) *module* P consists of a variable declaration and a set atoms_P of atoms. The *variable declaration* of P consists of three pairwise disjoint, finite sets of typed variables —the set $\mathsf{priv}X_P$ of *private variables*, the set $\mathsf{intf}X_P$ of *interface variables*, and the set $\mathsf{extl}X_P$ of *external variables*. We refer to $\mathsf{ctr}X_P = \mathsf{priv}X_P \cup \mathsf{intf}X_P$ as the *controlled variables* of P, to $\mathsf{obs}X_P = \mathsf{intf}X_P \cup \mathsf{extl}X_P$ as the *observable variables*, and to $X_P = \mathsf{ctr}X_P \cup \mathsf{obs}X_P$ as the *module variables*. The set atoms_P is a consistent set of X_P-atoms so that each variable $x \in X_P$ is controlled by some atom in atoms_P iff x is a controlled variable of P.

Terminology. For the controlled variables of a module P, by definition, $\mathsf{ctr}X_P = (\cup\, U \in \mathsf{atoms}_P \mid \mathsf{ctr}X_U)$. Similarly, we refer to $\mathsf{read}X_P = (\cup\, U \in \mathsf{atoms}_P \mid \mathsf{read}X_U)$ as the *read variables* of the module P, to $\mathsf{await}X_P = (\cup\, U \in \mathsf{atoms}_P \mid \mathsf{await}X_U)$ as the *awaited variables* of P, and to $\prec_P = \prec_{\mathsf{atoms}_P}$ as the *await relation* of P. The execution orders for atoms_P are called *execution orders* of P. ■

Important special cases of modules are the *finite*, the *closed*, and the *deterministic* modules. A module is finite if all module variables can assume only finitely many values. A module is closed if the behavior of the controlled variables is not influenced by the behavior of any environment variables (although the behavior of some environment variables may be influenced by the behavior of the controlled variables). A module is deterministic if the behavior of the environment variables uniquely determines the behavior of the controlled variables.

FINITE, CLOSED, AND DETERMINISTIC MODULES

The module P is *finite* if all module variables in X_P have finite types; otherwise, P is an *infinite* module. The module P is *closed* if the set $\mathsf{extl}X_P$ of external variables is empty; otherwise, P is an *open* module. The module P is *deterministic* if all atoms in atoms_P are deterministic; otherwise, P is a *nondeterministic* module.

The execution of modules

A module P is *executed* by dividing every round into two phases. In the first phase of a round, the external variables of P are initialized or updated nondeterministically: each external variable obtains an arbitrary value of the appropriate type. In the second phase of the round, the controlled variables are initialized or updated by carrying out the initial or update commands of P in some execution order. The first phase ensures that all initial and next values of external variables are available should they be needed in the second phase. The atom consistency of P ensures, by Proposition 1.1, the existence of an execution order for the second phase. The outcome of the execution is an *initialized trajectory* of P, which gives a sequence of values for each variable in X_P. By Exercise 1.2, the choice of execution order does not influence the outcome of the execution. However, since the external variables are initialized and updated nondeterministically, and since the initial and update commands may be nondeterministic, a module can have many initialized trajectories. Only modules that are both closed and deterministic have, for any given number of rounds, a unique initialized trajectory.

The observable part of an initialized trajectory of the module P, which gives a sequence of values for each variable in $\mathsf{obs}X_P$, is called a *trace* of P. Thus, every trace of P shows a possible observable behavior of P over time. Since different initialized trajectories (outcomes of executions) may give rise to the same trace (observable behavior), even modules that are both closed and deterministic can have many traces of a given length. Formal definitions of trajectories and traces will be given in Chapters 2 and 5.

Example 1.8 [Scheduler] The scheduler from Example 1.4 can be built from the three modules whose RML descriptions are given in Figure 1.4 without atom bodies. The modules *Task1* and *Task2* are closed; the module *Scheduler* is open. For illustration, we execute the module *Scheduler* in isolation. There are

```
module Task1 is
  interface $new_1 : \mathbb{N}$
  atom A1 controls $new_1$

module Task2 is
  interface $new_2 : \mathbb{N}$
  atom A2 controls $new_2$

module Scheduler is
  private $prior : \{1, 2\}$
  interface $task_1, task_2 : \mathbb{N};\ proc : \{0, 1, 2\}$
  external $new_1, new_2 : \mathbb{N}$
  atom A3 controls $task_1$ reads $task_1$ awaits $new_1, proc$
  atom A4 controls $task_2$ reads $task_2$ awaits $new_2, proc$
  atom A5 controls $proc, prior$ reads $task_1, task_2, prior$
```

Figure 1.4: The three scheduler modules

two execution orders, $A5, A3, A4$ and $A5, A4, A3$. In the first phase of the initial round, the external variables new_1 and new_2 are assigned arbitrary nonnegative integers, and in the second phase, the initial commands of the three atoms are executed in one of the two execution orders. In the first phase of every update round, the external variables new_1 and new_2 are assigned arbitrary new nonnegative integers, and in the second phase, the update commands of the three atoms are executed in some execution order. Since all initial and update commands are deterministic except for the initial value of the variable *prior*, for any two sequences of values for the external variables new_1 and new_2, and any initial value of *prior*, the module *Scheduler* has a unique initialized trajectory. The two trajectories of Figure 1.2 are initialized trajectories of the module *Scheduler*, and a third initialized trajectory is shown in Figure 1.5. In the third trajectory, the values of the external variables new_1 and new_2 are updated arbitrarily, in a manner that is not compliant with the modules *Task1* and *Task2*. If the values of the private variable *prior* are omitted from an initialized trajectory, we obtain a trace of the module *Scheduler*. ■

Block diagrams

We depict the structure of modules graphically using *block diagrams.* The block diagram for a module consists of delay elements and gates which are connected by wires, and of a module boundary. Each controlled variable is represented by a delay element whose output wire carries, in every update round, the current value of the variable. Each atom is represented by a gate whose output wires

new_1	1	2	0	0	1	0	3	2	0	0	0	0	5	0	0
new_2	0	2	0	0	0	0	5	0	0	0	0	0	5	0	0
$task_1$	1	2	2	1	2	1	4	5	5	4	4	3	8	7	7
$task_2$	0	2	1	1	0	0	4	4	3	3	2	2	6	6	5
$proc$	0	1	2	1	2	1	2	1	2	1	2	1	2	1	2
$prior$	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1

new_1	1	2	0	0	1	0	3	2	0	0	0	0	5	0	0
new_2	0	2	0	0	0	0	5	0	0	0	0	0	5	0	0
$task_1$	1	2	2	1	2	1	4	5	5	4	4	3	8	7	7
$task_2$	0	2	1	1	0	0	4	4	3	3	2	2	6	6	5
$proc$	0	1	2	1	2	1	2	1	2	1	2	1	2	1	2

Figure 1.5: An initialized trajectory of the module *Scheduler* and the corresponding trace

carry the next values of the variables that are controlled by the atom. The output wires of a gate are connected with the corresponding delay elements, where the updated values of the variables are stored for the next round. Thus there are two wires for each variable —one that carries the current value (delay output) and one that carries the next value (delay input) of the variable. The wires from delay elements to gates represent read dependencies between variables, and the wires from gates to gates represent await dependencies. Since the precedence relation on the atoms (gates) is asymmetric (Proposition 1.1), every wire cycle contains at least one delay element.

The delay elements and the gates of a module are circumscribed by a dotted line that denotes the module boundary. Each interface variable x is represented by two output wires, labeled x and x', that penetrate the module boundary from the inside to the outside. In every update round, the unprimed output wire carries the current value of the variable x, and the primed output wire carries the next value of x. Each external variable y is represented by one or two input wires, labeled y and y', that penetrate the module boundary from the outside to the inside. Since the module may use only the current value of y, or only the next value of y, the primed input wire or the unprimed input wire can be absent.

Example 1.9 [Scheduler] The block diagrams for the modules *Task1*, *Task2*, and *Scheduler* from Example 1.8 are shown in Figure 1.6. ■

Remark 1.5 [Block diagrams as types] The block diagram for a module contains the same information as the variable declarations and the atom declarations of the module. We call this information the *type* of the module. The

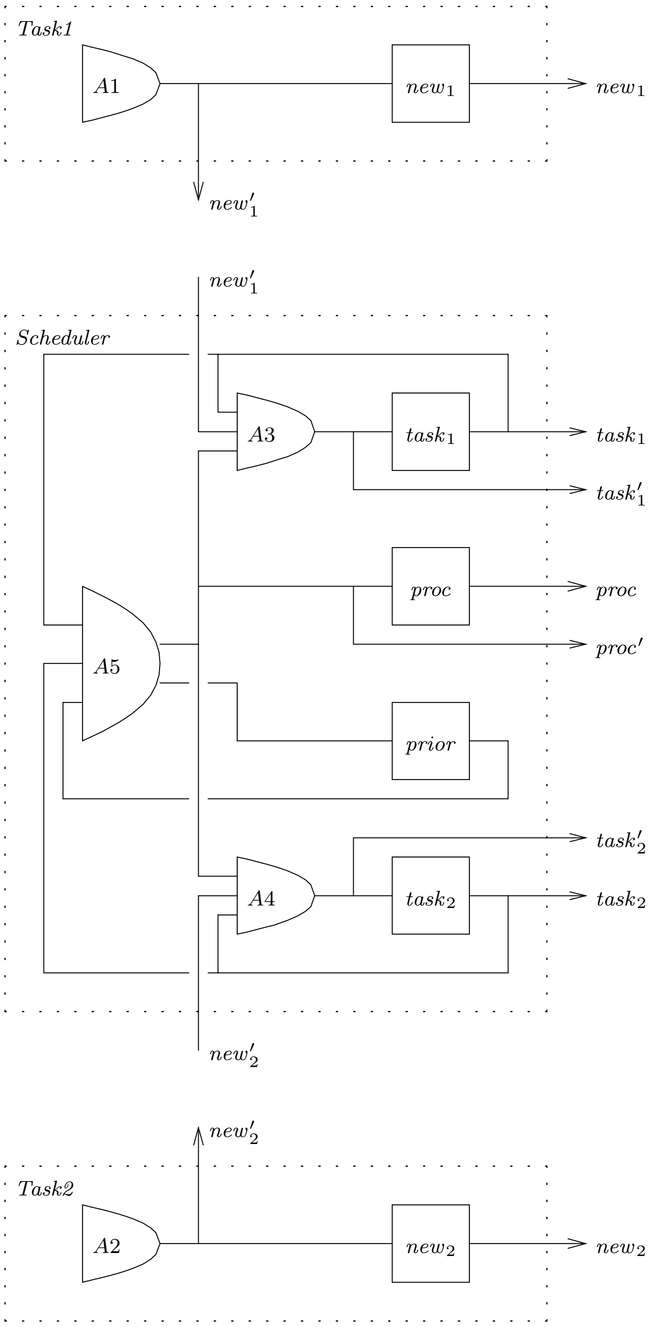


Figure 1.6: Block diagrams for the scheduler modules

type of a module does not include the initial and the update commands of the module. In particular, from the type of a module, it can be concluded if the module is finite, or closed, but not if the module is deterministic. ■

Asynchronous and synchronous modules

A module *stutters* in an update round if the values of all interface variables stay unchanged. A module that may stutter in every update round is called *asynchronous*. The environment cannot enforce observable progress of an asynchronous module. While an asynchronous module can privately record all updates of external variables, all updates of interface variables proceed at a speed that is independent of the environment speed. By contrast, a module that interacts with the environment synchronously may agree to modify an interface variable dependent on, and within the same round as, the modification of an external variable.

> ASYNCHRONOUS VS. SYNCHRONOUS MODULE
>
> The module P is *asynchronous* if all interface variables in $\mathsf{intf}X_P$ are controlled by lazy atoms. Otherwise, P is a *synchronous* module.

Example 1.10 [Zero-delay vs. unit-delay vs. buffered squaring] The module

```
module SyncSquare is
  interface out: $\mathbb{N}$
  external in: $\mathbb{N}$
  atom controls out awaits in
    initupdate
      $[\!] \; true \rightarrow out' := (in')^2$
```

waits, in every round, for the next value of the external nonnegative-integer variable *in*, computes the square, and displays the result in the interface variable *out*, all within the same round. This is done by a single, combinational atom. Thus, *SyncSquare* is an operator that transforms an input stream of nonnegative integers into an output stream of corresponding squares. The operator is synchronous, because every output value is produced in the very round in which the corresponding input value arrives. The module

```
module DelayedSyncSquare is
  interface out: $\mathbb{N}_\perp$
  external in: $\mathbb{N}$
  atom controls out awaits in
    init
      $[\!] \; true \rightarrow out' := \perp$
    update
      $[\!] \; true \rightarrow out' := in^2$
```

requires exactly one round to produce the square of an input value (initially, the output value is the undefined value $\perp$). Since a new output value cannot be delayed arbitrarily, the module *DelayedSyncSquare* is again synchronous.

By contrast, the asynchronous module *AsyncSquare* from Figure 1.7 implements an operator that requires an arbitrary number of rounds (possibly 0) for producing the square of an input value. From one update round to the next, the unprocessed input values are stored in the private queue *buffer*. If consecutive unprocessed input values are equal, only one representative is stored in *buffer*, and the square is computed only once. The queue *buffer* is updated in every round in which a new unprocessed input value arrives, so that no input values are lost. New output values, on the other hand, are produced after arbitrary delays. The module *AsyncSquare* therefore has two atoms. The atom *ComputeOut*, which controls *out*, is lazy: in every update round, it either computes a square and displays the result in the interface variable *out*, or it sleeps. If the queue *buffer* is empty, the square is computed for the external variable *in*; otherwise, the square is computed for the first element of the queue. The atom *StoreIn*, which controls *buffer*, is eager: in every update round, it waits both for the next input value and for the action taken by the atom *ComputeOut*, and reacts as follows. Whenever the input value changes and the queue *buffer* is nonempty —i.e., there are already some unprocessed input values— the new input value is added to the queue. The same happens if the input value changes and the queue is empty, but the output value does not change —i.e., the atom *ComputeOut* has decided to sleep. Whenever the output value changes (*ComputeOut* has decided to compute a square) and *buffer* is nonempty (the square is computed for the first element of the queue), the first element is removed from the queue. Note that, because the keyword **passive** is used, the update command of *StoreIn* contains both the conditional sleep assignment

$$[\!]\ \mathit{in}' = \mathit{in} \wedge \mathit{out}' = \mathit{out} \rightarrow \mathit{buffer}' := \mathit{buffer}$$

and the default assignment

$$[\!]\ \mathit{out}' \neq \mathit{out} \wedge \mathit{IsEmpty}(\mathit{buffer}) \rightarrow \mathit{buffer}' := \mathit{buffer}.$$

Figure 1.8 shows one, arbitrarily chosen, initialized trajectory for each of the three modules *SyncSquare*, *DelayedSyncSquare*, and *AsyncSquare*. For the modules *SyncSquare* and *DelayedSyncSquare*, which have no private variables, the traces coincide with the initialized trajectories. For the module *AsyncSquare*, we obtain the traces by omitting the values of the private queue *buffer* from the initialized trajectories. Every trace of the synchronous modules *SyncSquare* and *DelayedSyncSquare* is also a trace of the asynchronous module *AsyncSquare*, but the converse is not true. While *SyncSquare* and *DelayedSyncSquare* each have exactly one trace for any given sequence of input values, *AsyncSquare* may have infinitely many. ■

```
module $AsyncSquare$ is
  private $buffer$ : queue of $\mathbb{N}$
  interface $out$ : $\mathbb{N}_\perp$
  external $in$ : $\mathbb{N}$

  lazy atom $ComputeOut$
    controls $out$
    reads $out, buffer$
    awaits $in$
    init
      $[\!]\ true \to out' := (in')^2$
      $[\!]\ true \to out' := \perp$
    update
      $[\!]\ IsEmpty(buffer) \quad \to out' := (in')^2$
      $[\!]\ \neg IsEmpty(buffer) \to out' := Front(buffer)^2$

  passive atom $StoreIn$
    controls $buffer$
    reads $in, out, buffer$
    awaits $in, out$
    init
      $[\!]\ out' \neq \perp \to buffer' := EmptyQueue$
      $[\!]\ out' = \perp \to buffer' := Enqueue(in', EmptyQueue)$
    update
      $[\!]\ \begin{bmatrix} \wedge\ in' \neq in \\ \wedge\ out' \neq out \\ \wedge\ \neg IsEmpty(buffer) \end{bmatrix} \to buffer' := Enqueue(in, Dequeue(buffer))$
      $[\!]\ \begin{bmatrix} \wedge\ in' = in \\ \wedge\ out' \neq out \\ \wedge\ \neg IsEmpty(buffer) \end{bmatrix} \to buffer' := Dequeue(buffer)$
      $[\!]\ \begin{bmatrix} \wedge\ in' \neq in \\ \wedge\ out' = out \end{bmatrix} \to buffer' := Enqueue(in, buffer)$
```

Figure 1.7: Asynchronous squaring

in	1	2	2	3	3	3	4	4	4	4	5	6	7	8	9
out	1	4	4	9	9	9	16	16	16	16	25	36	49	64	81

in	1	2	2	3	3	3	4	4	4	4	5	6	7	8	9
out	$\perp$	1	4	4	9	9	9	16	16	16	16	25	36	49	64

in	1	2	2	3	3	3	4	4	4	4	5	6	7	8	9
out	$\perp$	$\perp$	1	4	9	9	9	9	9	16	25	25	25	25	36
buffer	1	1, 2	2	3			4	4	4			6	6, 7	6, 7, 8	7, 8, 9

Figure 1.8: Three initialized trajectories of the modules *SyncSquare* (top), *DelayedSyncSquare* (middle), and *AsyncSquare* (bottom)

Exercise 1.3 {`P3`} [Squaring inputs] Following Example 1.10, you are asked to implement two more operators that transform an input stream of nonnegative integers into an output stream of corresponding squares. (a) Define a module *AsyncSquare2* which, like *AsyncSquare*, requires an arbitrary number (possibly 0) of rounds to produce a square but, unlike *AsyncSquare*, computes the square of each individual input value, even if it is identical to the previous input value. Give an initialized trajectory of *AsyncSquare* such that the corresponding trace is not a trace of *AsyncSquare2*. (b) Define a module *SyncSquare2* which requires at least 2 and at most 5 rounds to produce a square. (c) Draw the block diagrams for the four modules *SyncSquare*, *SyncSquare2*, *AsyncSquare*, and *AsyncSquare2*. ■

Passive and active modules

A module *sleeps* in an update round if the values of all controlled variables stay unchanged. The environment *stutters* in an update round if the values of all external variables stay unchanged. A module that may sleep in every update round in which the environment stutters is called *passive*. The environment can enforce the progress of a passive module only by modifying external variables.

Passive vs. active module

The module P is *passive* if all atoms in atoms_P are passive. Otherwise, P is an *active* module.

Remark 1.6 [Progress of passive modules] As long as the environment stutters, a passive module may sleep: for a passive module we obtain an initialized trajectory of any given length by simply repeating initial values for all variables. In particular, a closed, passive module may sleep in every update round. The following exercise presents a generalization of this remark. ■

Exercise 1.4 {T2} [Stutter closure for passive modules] Consider a module P, and for each variable in X_P, consider a sequence of $k \geq 1$ values. Construct sequences of $k+1$ values by repeating the last value of each sequence of length k. (a) Prove that if P is passive and the sequences of length k form an initialized trajectory of P, then also the sequences of length $k+1$ form an initialized trajectory of P. (b) In part (a), can you replace "passive" by "asynchronous" and "initialized trajectory" by "trace"? ■

Exercise 1.5 {P1} [Classification of modules] A module may be asynchronous and passive, synchronous and passive, asynchronous and active, or synchronous and active. Give examples for all four classes of modules. ■

Private determinism

For a deterministic module, the behavior of the controlled variables, both private and interface, is uniquely determined by the behavior of the external variables. Thus, for a nondeterministic module, nondeterminism may manifest itself in the initialization and updating of private variables, or of interface variables, or both. If all nondeterminism is limited to interface variables, we call it *private determinism*: a module exhibits private determinism if the behavior of the private variables is uniquely determined by the behavior of the observable variables, both interface and external. It follows that for every privately deterministic module, there is a one-to-one correspondence between the initialized trajectories (i.e., the outcomes of executions) and the traces (i.e., the observable behaviors).

PRIVATE DETERMINISM

The module P has *private determinism* if all private variables in $\mathsf{priv}X_P$ are controlled by deterministic atoms. Otherwise, P has *private nondeterminism.*

Example 1.11 [Determinism vs. private determinism vs. nondeterminism] The modules *SyncSquare* and *DelayedSyncSquare* of Example 1.10 are deterministic. The module *AsyncSquare* is nondeterministic, because it requires an arbitrary number of rounds for processing an input value. However, *AsyncSquare* has private determinism, because the initial value of the private queue *buffer* is uniquely determined by the initial values of the observable variables *in* and *out*, and in every update round, the next value of *buffer* is uniquely determined by the current value of *buffer* and the current and next values of *in* and *out*. In this way, given a trace of *AsyncSquare*, we can construct a unique corresponding initialized trajectory. By contrast, the module *LossyAsyncSquare* of Figure 1.9, which implements asynchronous squaring using a lossy queue, does not have private determinism. The module *LossyAsyncSquare* shares the atom *ComputeOut* with the module *AsyncSquare*, but differs in the atom that controls the private

```
module $LossyAsyncSquare$ is
  private $buffer$ : queue of $\mathbb{N}$
  interface $out$ : $\mathbb{N}_\perp$
  external $in$ : $\mathbb{N}$

  lazy atom $ComputeOut$
    controls $out$
    reads $out, buffer$
    awaits $in$

  passive atom $LossyStoreIn$
    controls $buffer$
    reads $in, out, buffer$
    awaits $in, out$
    init
      $[\!] \; true \to buffer' := EmptyQueue$
      $[\!] \; out' = \perp \to buffer' := Enqueue(in', EmptyQueue)$
    update
      $[\!] \; \begin{bmatrix} \wedge \; in' \neq in \\ \wedge \; out' \neq out \\ \wedge \; \neg IsEmpty(buffer) \end{bmatrix} \to buffer' := Enqueue(in, Dequeue(buffer))$
      $[\!] \; \begin{bmatrix} \wedge \; out' \neq out \\ \wedge \; \neg IsEmpty(buffer) \end{bmatrix} \to buffer' := Dequeue(buffer)$
      $[\!] \; \begin{bmatrix} \wedge \; in' \neq in \\ \wedge \; out' = out \end{bmatrix} \to buffer' := Enqueue(in, buffer)$
```

Figure 1.9: Lossy asynchronous squaring

in	1	2	2	3	3	3	4	5	6	7
out	$\bot$	1	1	1	1	1	1	9	9	9
buffer	1			3	3	3	$3, 4$	$4, 5$	$4, 5$	$4, 5$

in	1	2	2	3	3	3	4	5	6	7
out	$\bot$	1	1	1	1	1	1	9	9	9
buffer	1			3	3	3	3		6	$6, 7$

Figure 1.10: Two observably equivalent initialized trajectories of the module $LossyAsyncSquare$

queue *buffer*: whenever the input value changes, it may or may not be added to *buffer*, and thus, some input values can be lost. Furthermore, in any given update round, whether the next input value is lost or not is independent of the current value of *buffer* and independent of the current and next values of *in* and *out*. Figure 1.10 shows two distinct initialized trajectories of the module *LossyAsyncSquare* which give rise to the same trace. ■

1.2 Operations on Reactive Modules

We build complex modules from simple modules using three operations — parallel composition, variable renaming, and variable hiding.

1.2.1 Parallel Composition

The composition operation combines two modules into a single module whose behavior captures the interaction between the two component modules. Two modules can be composed only if their variable declarations are mutually consistent, and if the combined await dependencies of the two modules are not circular.

Module compatibility

The two modules P and Q are *compatible* if (1a) $\mathsf{priv}X_P$ and X_Q are disjoint, (1b) X_P and $\mathsf{priv}X_Q$ are disjoint, (1c) $\mathsf{intf}X_P$ and $\mathsf{intf}X_Q$ are disjoint, and (2) the transitive closure $(\prec_P \cup \prec_Q)^+$ is asymmetric.

Remark 1.7 [Independent modules] If the module variables of two modules are disjoint, then the two modules are compatible. ■

The composition operation is defined for compatible modules.

MODULE COMPOSITION

If P and Q are two compatible modules, then the (*parallel*) *composition* $P\|Q$ is the module such that

- each private variable of a component module is a private variable of the compound module: $\mathsf{priv}X_{P\|Q} = \mathsf{priv}X_P \cup \mathsf{priv}X_Q$;
- each interface variable of a component module is an interface variable of the compound module: $\mathsf{intf}X_{P\|Q} = \mathsf{intf}X_P \cup \mathsf{intf}X_Q$;
- each external variable of a component module is an external variable of the compound module, provided it is not an interface variable of the other component: $\mathsf{extl}X_{P\|Q} = (\mathsf{extl}X_P \cup \mathsf{extl}X_Q)\backslash\mathsf{intf}X_{P\|Q}$;
- each atom of a component module is an atom of the compound module: $\mathsf{atoms}_{P\|Q} = \mathsf{atoms}_P \cup \mathsf{atoms}_Q$.

Remark 1.8 [Composing several modules] The composition operation on modules is commutative and associative. In RML, we therefore omit parentheses when writing module expressions such as $P\|Q\|R$. ■

Example 1.12 [Scheduler] By composing the three modules from Example 1.8 we obtain the module

module *SchedulerSystem* **is** *Task1* ‖ *Task2* ‖ *Scheduler*.

The type of the compound module is

module *SchedulerSystem* **is**
  private *prior* : $\{1,2\}$
  interface $new_1, new_2 : \mathbb{N}$; $task_1, task_2 : \mathbb{N}$; $proc : \{0,1,2\}$
  atom $A1$ **controls** new_1
  atom $A2$ **controls** new_2
  atom $A3$ **controls** $task_1$ **reads** $task_1$ **awaits** $new_1, proc$
  atom $A4$ **controls** $task_2$ **reads** $task_2$ **awaits** $new_2, proc$
  atom $A5$ **controls** $proc, prior$ **reads** $task_1, task_2, prior$

and the corresponding block diagram is shown in Figure 1.11. In the pictorial representation of parallel composition, each primed output wire of one component module is connected with all primed input wires of other component modules that represent the same variable, and (not occurring in the scheduler example) each unprimed output wire is connected with the corresponding unprimed input wires. The module *SchedulerSystem* is infinite, closed, nondeterministic, synchronous (all atoms are eager), and active. While every initialized trajectory of the compound module *SchedulerSystem* is also an initialized trajectory of the component module *Scheduler*, the converse is not true. This is

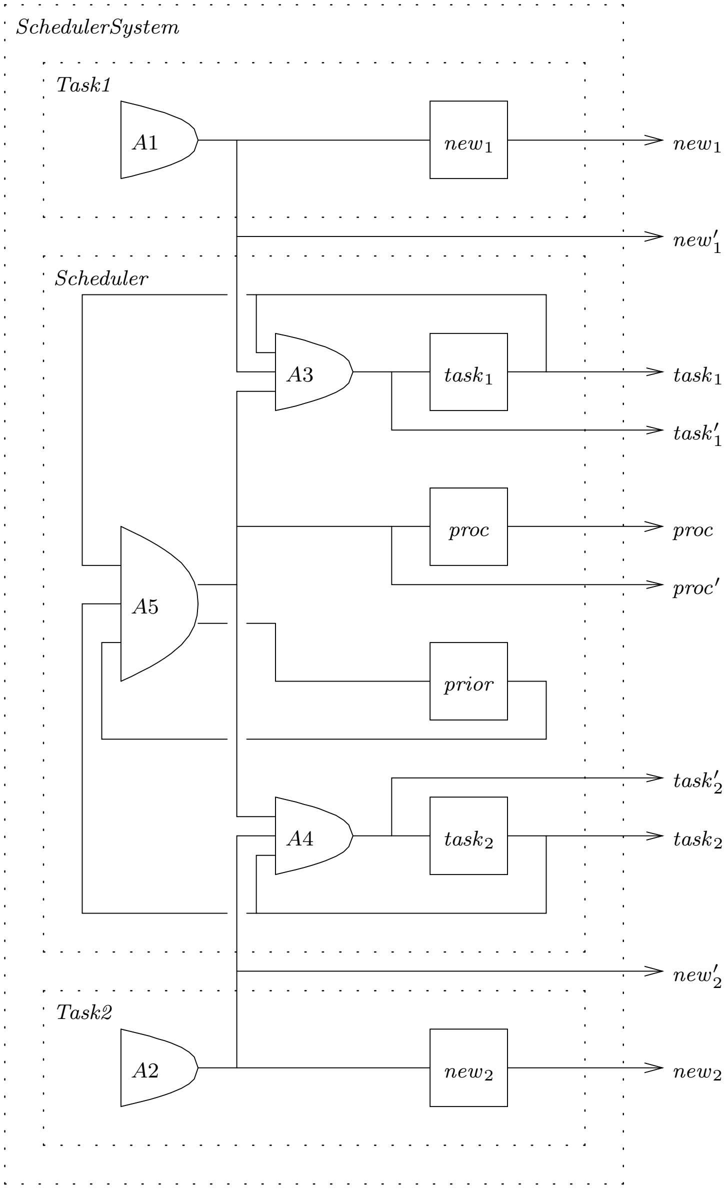


Figure 1.11: Block diagram for the scheduler system

because the component modules *Task1* and *Task2* constrain the behavior of the variables new_1 and new_2, which are external to *Scheduler*. For example, the two trajectories of Figure 1.2 are initialized trajectories of *SchedulerSystem*; the trajectory of Figure 1.5 is not. In Chapter 2 we will formally construct the trajectories of a compound module from the trajectories of the component modules. ■

Remark 1.9 [Module properties under parallel composition] The composition of two modules is finite iff both component modules are finite. The composition of two open modules may be closed. The composition of two modules is deterministic (or has private determinism) iff both component modules are deterministic (or have private determinism). The composition of two modules is asynchronous (or passive) iff both component modules are asynchronous (or passive). ■

Abstract block diagrams

In block diagrams, we may choose to hide the internal structure of a module and view it as a black box with input and output wires. If the atom structure of a module is suppressed, we draw the module boundary as a solid line instead of a dotted line. In order to compose such *abstract block diagrams*, every module needs to be annotated with information about the await dependencies between variables. The amount of compatibility information that is both necessary and sufficient is captured by the following definition. Given a module P, a *derived await dependency* $x \prec^d_P y$ of P consists of an external variable x and an interface variable y such that $x \prec^+_P y$. The derived await dependency $x \prec^d_P y$ indicates that the initial value of the interface variable y may depend on the initial value of the external variable x, and in every update round, the next value of y may depend on the next value of x. Therefore, P cannot be composed with a module Q with external variable y, interface variable x, and $y \prec^d_Q x$.

Exercise 1.6 {T2} [Derived await dependencies] Consider two modules P and Q whose variables satisfy conditions (1a)–(1c) for module compatibility. (a) Show that the derived await dependencies of the two modules contain exactly the information that is necessary and sufficient for determining compatibility; that is, prove that P and Q are compatible iff the transitive closure $(\prec^d_P \cup \prec^d_Q)^+$ is asymmetric. (b) Assuming P and Q are compatible, construct the derived await dependencies of the compound module $P \| Q$ from the derived await dependencies of the component modules. ■

A block diagram for a module P must show either its internal structure or its derived await dependencies. Since it is immaterial if a derived await dependency is an actual await dependency (contained in $\prec_P$) or only "derived" from other await dependencies (contained in $\prec^+_P$), we often omit the superscript d from the symbol $\prec^d$.

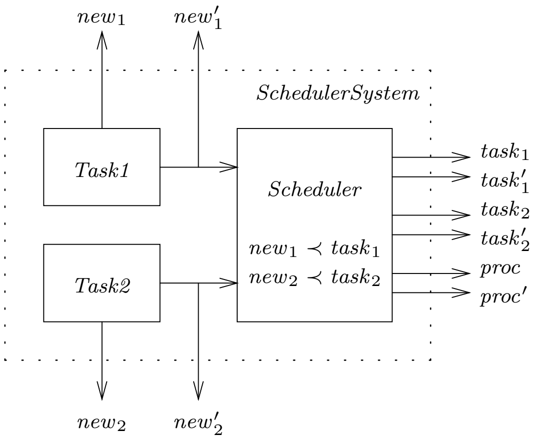


Figure 1.12: Abstract block diagram for the scheduler system

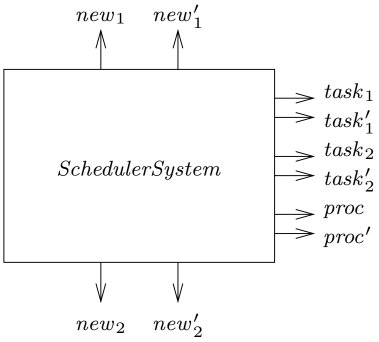


Figure 1.13: Very abstract block diagram for the scheduler system

Example 1.13 [Scheduler] The block diagram of Figure 1.12 does not show the atom structure of the component modules for the scheduler system from Example 1.12. The module *Scheduler* has two derived await dependencies, $new_1 \prec task_1$ and $new_2 \prec task_2$; for example, it cannot be composed with a module that awaits $task_1$ and controls new_1. The block diagram of Figure 1.13 further abstracts the internal structure of the compound module and views the entire scheduler system as a black box. The module *SchedulerSystem* is closed, and therefore has no derived await dependencies. ■

1.2.2 Variable Renaming

Before composing two modules, it may be necessary to rename private variables in order to make the two modules compatible. Variable renaming is also useful for identifying an interface variable of one module with an external variable of another module, and for creating multiple copies of a module.

VARIABLE RENAMING

Let X be a finite set of typed variables, and let ρ be a renaming for X. By ρ' we denote the renaming for $X \cup X'$ such that for all variables $x \in X$, $x[\rho'] = x[\rho]$ and $x'[\rho'] = x[\rho]'$. Given an X-atom U, the *renamed atom* $U[\rho]$ is the $X[\rho]$-atom with the set $\mathsf{ctr}X_U[\rho]$ of controlled variables, the set $\mathsf{read}X_U[\rho]$ of read variables, the set $\mathsf{await}X_U[\rho]$ of awaited variables, the initial command $\mathsf{init}_U[\rho]$, and the update command $\mathsf{update}_U[\rho]$. Given a module P, and a renaming ρ for the set X_P of module variables, the *renamed module* $P[\rho]$ is the module with the set $\mathsf{priv}X_P[\rho]$ of private variables, the set $\mathsf{intf}X_P[\rho]$ of interface variables, the set $\mathsf{extl}X_P[\rho]$ of external variables, and the set $\{U[\rho] \mid U \in \mathsf{atoms}_P\}$ of atoms.

Example 1.14 [Scheduler] From the generic task module

```
module Task is
  interface new : ℕ
  atom controls new
    init update
      [] true → new′ := 0
      [] true → new′ := 5
```

we can construct the two task modules of the scheduler system from Example 1.12 by renaming. In RML, we write

```
module Task1 is Task[new := new₁]
module Task2 is Task[new := new₂].
```

The interface variable *new* is renamed to create two distinct copies of the module *Task*. ■

Remark 1.10 [Module properties under variable renaming] Variable renaming preserves the cardinality (finite vs. infinite), closure (closed vs. open), determinism (deterministic vs. privately deterministic vs. nondeterministic), synchronicity (asynchronous vs. synchronous), and round-sensitivity (passive vs. active) properties of modules. ■

Implicit renaming of private variables

Two modules P and Q can be composed only if (1a) the private variables of P are disjoint from the module variables of Q, and (1b) the private variables of Q are disjoint from the module variables of P. If, however, we are not interested in the internal structures of P and Q, then we may not know the names of their private variables. To allow the parallel composition of modules in such circumstances, we treat the private variables of a module as "dummies," like the bound variables of a quantified formula (these variables can be renamed freely, and must be renamed suitably for safe substitution, before a free variable is replaced by an expression). In particular, in Rml we do not distinguish between modules that differ only in the names of private variables. Whenever we write $P \| Q$, we do not insist on conditions (1a) and (1b) for module compatibility, but rather assume that, implicitly, the private variables of P and Q are renamed suitably before the two modules are composed. (We still do insist, of course, on condition (1c) that the interface variables of P and Q are disjoint, and on condition (2) that the derived await dependencies of P and Q can be combined without introducing dependency cycles.)

1.2.3 Variable Hiding

The hiding of interface variables allows us to construct module abstractions of varying degrees of detail. For instance, after composing two modules, it may be appropriate to convert some interface variables to private variables, so that they are used only for the interaction of the component modules, and are no longer visible to the environment of the compound module.

Variable hiding

Given a module P, and an interface variable x in $\mathsf{intf}X_P$, by *hiding* x in P we obtain the module **hide** x **in** P with the set $\mathsf{priv}X_P \cup \{x\}$ of private variables, the set $\mathsf{intf}X_P \backslash \{x\}$ of interface variables, the set $\mathsf{extl}X_P$ of external variables, and the set atoms_P of atoms.

Remark 1.11 [Hiding several variables] In Rml, we write **hide** x_1, x_2 **in** P as an abbreviation for the module **hide** x_1 **in** (**hide** x_2 **in** P), which is identical to the module **hide** x_2 **in** (**hide** x_1 **in** P). ■

Example 1.15 [Scheduler] If we hide the interface variables task_1 and task_2 in the scheduler system from Example 1.14, we obtain the module

module *SchedulerSystem2* **is hide** $\mathit{task}_1, \mathit{task}_2$ **in** *SchedulerSystem*

whose block diagram is shown in Figure 1.14. Our conventions for the pictorial representation of variable renaming and variable hiding are evident from the figure. Hiding preserves the initialized trajectories of a module, but not the traces. A trace for the module *SchedulerSystem2* gives values only to the observable (interface) variables new_1, new_2, and proc. ■

Remark 1.12 [Module properties under variable hiding] Variable hiding preserves the cardinality, closure, and round-sensitivity properties of modules. Hiding preserves determinism, but may not preserve private determinism. By hiding a variable in a synchronous module we may obtain an asynchronous module. ■

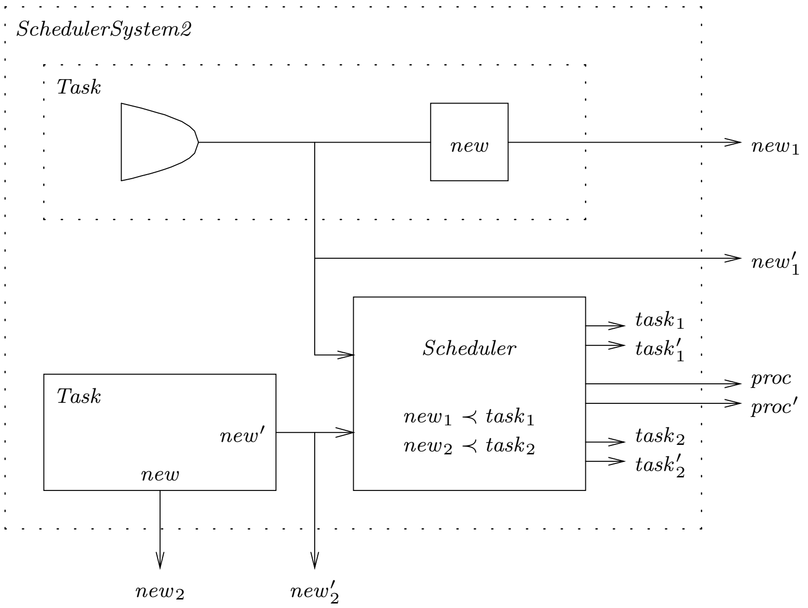


Figure 1.14: Block diagram for the scheduler system with renaming and hiding

Remark 1.13 [Abstract block diagrams as abstract types] Every block diagram for a module P, no matter how abstract, contains four pieces of information: the read external variables, the awaited external variables, the interface variables, and the derived await dependencies of P. We call this information the *abstract type* of the module. Since the names of private variables are immaterial, the abstract types of two modules suffice for determining if the two modules are compatible. Given a complex module that is built from simple modules using the three operations of composing, renaming, and hiding, and given the (abstract) types of all simple modules, we can infer the (abstract) type of the complex module. It is for this reason that the operations of composing, renaming, and hiding can be performed also on block diagrams. ■

1.3 Examples of Reactive Modules

We draw on examples from several application domains —synchronous and asynchronous hardware, concurrent programs with read-shared variables, and distributed programs with synchronous and asynchronous message passing.

1.3.1 Synchronous Circuits

Synchronous circuits are built from logic gates and memory cells that are driven by a sequence of clock ticks. Each logic gate computes a boolean value once per clock cycle, and each memory cell stores a boolean value from one clock cycle to the next. We model each logic gate and each memory cell as a reactive module so that every update round represents a clock cycle. The wires that connect the logic gates and the memory cells are modeled as boolean variables. As is customary in circuit design, we denote the values of wires by 0 and 1 instead of *false* and *true*, respectively. We construct synchronous circuits from three basic building blocks: as basic logic gates we use the Not gate and the And gate, and as basic memory cell we use the latch (set-reset flip-flop). These building blocks are then combined to circuits by applying the three module operations of parallel composition, variable renaming, and variable hiding.

Combinational circuits

Figure 1.15 defines three deterministic, synchronous, passive modules for modeling Not, And, and Or gates. The module *SyncNot* models a zero-delay Not gate, which takes a boolean input and produces a boolean output. The input is modeled as an external variable, *in*, because it is modified by the environment and visible to the gate. The output is modeled as an interface variable, *out*, because it is modified by the gate and visible to the environment. In the initial round, the Not gate waits for the input value to be initialized before computing the initial output value, by negating the initial input value. In every update round, the Not gate waits for the input value to be updated before computing the next output value, by negating the updated input value. The module *SyncNot* is passive, because the output changes only if the input changes, and synchronous, because the output changes in the very round (clock cycle) in which the input changes (zero delay). The module *SyncAnd* models a zero-delay And gate in similar fashion. The And gate takes two boolean inputs, represented by the external variables in_1 and in_2, and produces a boolean output, represented by the interface variable *out*. In the initial round, both input values must be initialized before the gate issues the initial output value. In every update round, both input values must be updated before the gate issues the next output value with zero delay.

From Not and And gates we can build all combinational circuits. For example, by de Morgan's law, a zero-delay Or gate can be defined by composing a zero-delay And gate with three zero-delay Not gates that negate both inputs and the output of the And gate. The resulting module *SyncOr* has the same abstract type as the module *SyncAnd* —two awaited boolean inputs represented by the external variables in_1 and in_2, and a boolean output represented by the interface variable *out* which depends on both inputs (the long dashes in Figure 1.15 indicate Rml commentary). The private variables z_1, z_2, and z_3 of *SyncOr*

```
module $SyncNot$ is
  interface $out : \mathbb{B}$
  external $in : \mathbb{B}$
  atom controls $out$ awaits $in$
    initupdate
      $[\!] \; in' = 0 \to out' := 1$
      $[\!] \; in' = 1 \to out' := 0$

module $SyncAnd$ is
  interface $out : \mathbb{B}$
  external $in_1, in_2 : \mathbb{B}$
  atom controls $out$ awaits $in_1, in_2$
    initupdate
      $[\!] \; in'_1 = 0 \qquad\qquad\quad \to out' := 0$
      $[\!] \; in'_2 = 0 \qquad\qquad\quad \to out' := 0$
      $[\!] \; in'_1 = 1 \wedge in'_2 = 1 \to out' := 1$

module $SyncOr$ is
  —interface $out$
  —external $in_1, in_2$
  hide $z_1, z_2, z_3$ in
    $\| \; SyncAnd[in_1, in_2, out := z_1, z_2, z_3]$
    $\| \; SyncNot[in, out := in_1, z_1]$
    $\| \; SyncNot[in, out := in_2, z_2]$
    $\| \; SyncNot[in, out := z_3, out]$
```

Figure 1.15: Synchronous Not, And, and Or gates

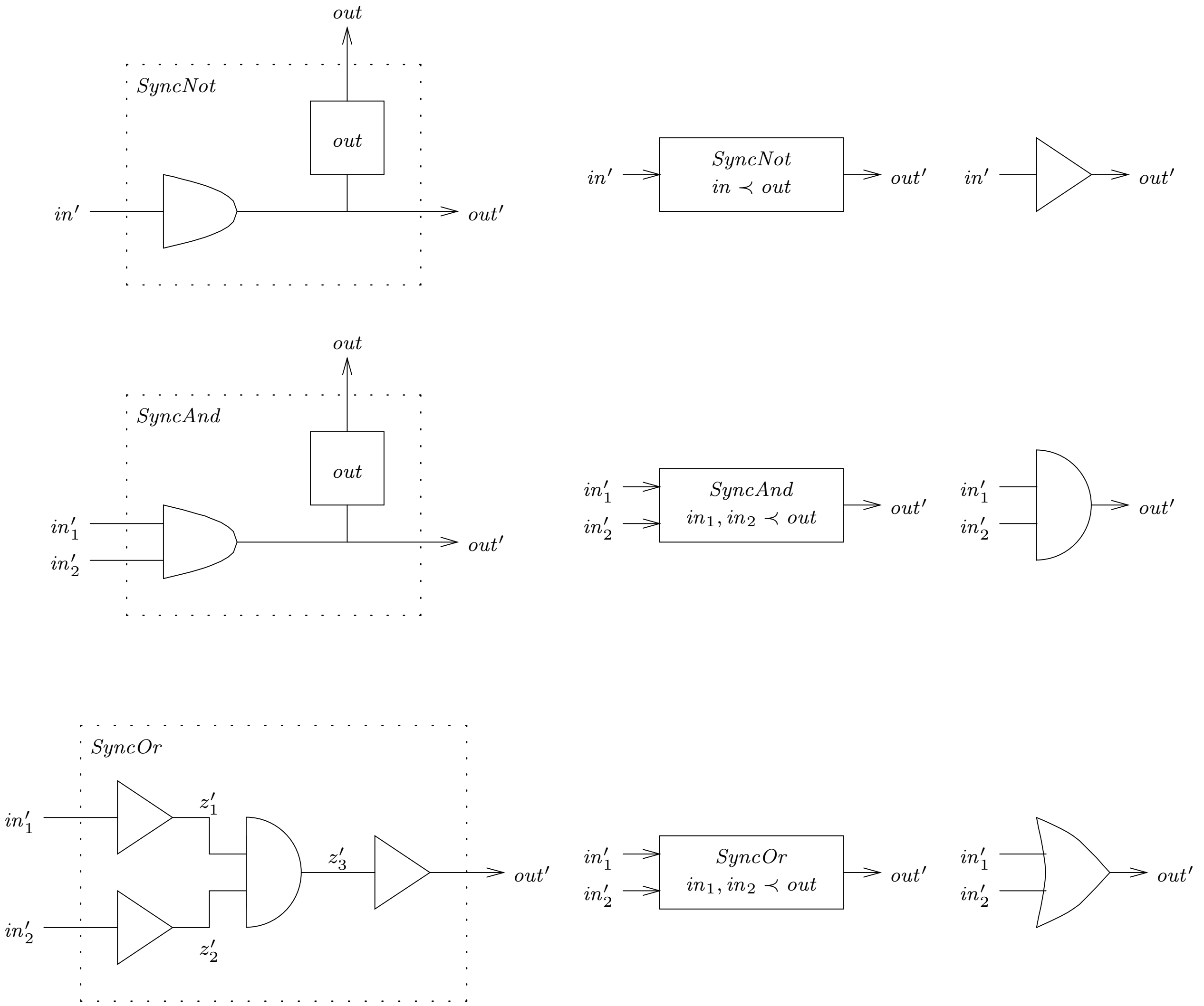


Figure 1.16: Block diagrams for the synchronous Not, And, and Or gates

represent internal wires that connect the four component gates. The values of these internal wires can be neither read nor modified by the environment. Since *out* waits for z_3, which waits for both z_1 and z_2, which wait for in_1 and in_2, respectively, we obtain the two derived await dependencies $in_1 \prec out$ and $in_2 \prec out$; that is, every update of the output must be preceded by updates of both inputs. The module *SyncOr* is again passive (all atoms are combinational) and synchronous (an eager atom controls the output).

Figure 1.16 shows, in the left column, detailed block diagrams for the zero-delay Not, And, and Or gates, and in the center column, corresponding abstract block diagrams. We omit the unprimed output wires from the abstract block diagrams of logic gates, because they are not used for building circuits (to save the value of a wire from one clock cycle of a synchronous circuit to the subsequent clock cycle, the value must be latched). We abbreviate the abstract block diagrams of the logic gates using module boundaries of different shapes which resemble the standard gate symbols. This allows us to suppress the derived await dependencies. The abbreviations are shown in the right column of Figure 1.16.

Sequential circuits

Figure 1.17 defines a nondeterministic, synchronous, active module for modeling a unit-delay latch. The latch takes two boolean inputs, represented by the external variables *set* and *reset*, and produces a boolean output, represented by the interface variable *out*. Unlike the logic gates, the latch has a boolean state, which is represented by the private variable *state*. The latch behaves like a Moore machine. In every update round, the latch first issues its state as output and then waits for the updated input values to compute its next state. If the updated value of *set* is 1 and the updated value of *reset* is 0, then the latch changes its state to 1. If the updated value of *set* is 0 and the updated value of *reset* is 1, then the latch changes its state to 0. If both updated input values are 0, then the state of the latch (which is equal to the already updated output out') does not change. If both updated input values are 1, then the next state of the latch is arbitrary —it may be either 0 or 1 (in this case, two of the guards apply, and the value of *state* is updated nondeterministically). What remains to be specified are the initial values of *out* and *state*. The initial output of the latch is arbitrary (this is a second source of nondeterminism). The initial state of the latch is computed combinationally from the initial values of *out*, *set*, and *reset* as during update rounds (in particular, if both *set* and *reset* are initially 0, then the initial value of *state* is determined by the initial value of *out*).

The resulting module *SyncLatch* is active, because in every round a new output is issued independently of any input change, and synchronous, because after an input change the output changes in the very next round (unit delay). Figure 1.18 shows the detailed block diagram for the latch and, below, an abstract

```
module SyncLatch is
  private state : $\mathbb{B}$
  interface out : $\mathbb{B}$
  external set, reset : $\mathbb{B}$

  atom ComputeOutput controls out reads state
    init
      $[\!]$ $true \rightarrow out' := \mathbb{B}$
    update
      $[\!]$ $true \rightarrow out' := state$

  atom ComputeNextState controls state awaits out, set, reset
    initupdate
      $[\!]$ $set' = 1$                        $\rightarrow state' := 1$
      $[\!]$ $reset' = 1$                      $\rightarrow state' := 0$
      $[\!]$ $set' = 0 \wedge reset' = 0$      $\rightarrow state' := out'$
```

Figure 1.17: Synchronous latch

block diagram. As with logic gates, we omit the unprimed output wire from the abstract block diagram of the latch, because it is not used for building circuits. Note that while zero-delay logic gates have (derived) await dependencies between inputs and outputs, the unit-delay latch does not. For this it was necessary to model the latch with two atoms, each controlling one variable, rather than with a single atom controlling both variables: in the module *SyncLatch*, the state variable *state* waits for the input variables *set* and *reset*; the output variable *out* does not. This decoupling of the output computation, which requires no inputs, from the next-state computation, which requires both inputs, into separate subrounds is essential for composing latches with logic gates which, in every round (clock cycle), provide the latch inputs dependent on the latch outputs.

Example 1.16 [Binary counter] As an example of a sequential circuit, we design a three-bit binary counter. The counter takes two boolean inputs, represented by the external variables *start* and *inc*, for starting and incrementing the counter. The counter value ranges from 0 to 7, and is represented by three bits. We do not make any assumption about the initial counter value. A start command resets the counter value to 0 and overrides any increment command that is issued in the same round. An increment command increases the counter value by 1. If the counter value is 7, the increment command changes the counter value to 0. In every round, the counter issues its value as output —the low bit on the interface variable out_0, the middle bit on the interface variable out_1, and the high bit on the interface variable out_2. (While combinational circuits are

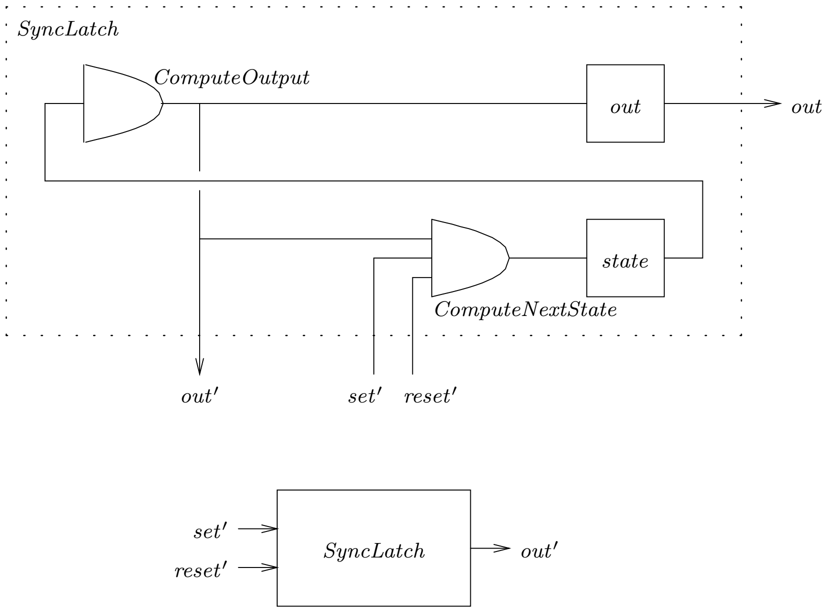


Figure 1.18: Block diagrams for the synchronous latch

passive, sequential circuits are active.)

Figure 1.19 shows a possible design of the three-bit counter from three one-bit counters (for clarity, in Rml we can annotate the component modules of a compound module with the names of the observable variables even if the variables are not renamed). Note that $carry_0$ waits for both *start* and *inc*, then $carry_1$ waits for $carry_0$, and $carry_2$ waits for $carry_1$. It follows that all three bits of the counter are updated in a single round (clock cycle). Figure 1.20 shows block diagrams for the one-bit counter *Sync1BitCounter* and the three-bit counter *Sync3BitCounter*. The module *Sync3BitCounter* has no derived await dependencies; it is finite, open, nondeterministic (because the initial counter value is arbitrary), privately deterministic, synchronous, and active. Figure 1.21 shows an initial trajectory of *Sync3BitCounter* and, for some of the variables, the corresponding timing diagram. (The private variables of the three one-bit counters have been renamed implicitly; for instance, z has been renamed to z_0, z_1, and z_2.) ■

Exercise 1.7 {P3} [Synchronous circuits] (a) Define a passive module *SyncNor* that models a zero-delay Nor gate. Use the variable names in_1 and in_2 for

```
module $Sync1BitCounter$ is
  —interface $out, carry$
  —external $start, inc$
  hide $set, reset, z$ in
    ‖ $SyncLatch[set, reset, out]$
    ‖ $SyncAnd[in_1, in_2, out := out, inc, carry]$
    ‖ $SyncOr[in_1, in_2, out := carry, start, reset]$
    ‖ $SyncNot[in, out := reset, z]$
    ‖ $SyncAnd[in_1, in_2, out := inc, z, set]$

module $Sync3BitCounter$ is
  —interface $out_0, out_1, out_2$
  —external $start, inc$
  hide $carry_0, carry_1, carry_2$ in
    ‖ $Sync1BitCounter[start, inc, out, carry := start, inc, out_0, carry_0]$
    ‖ $Sync1BitCounter[start, inc, out, carry := start, carry_0, out_1, carry_1]$
    ‖ $Sync1BitCounter[start, inc, out, carry := start, carry_1, out_2, carry_2]$
```

Figure 1.19: One-bit and three-bit binary counters

input, and use *out* for output. (b) Why is

```
hide $z$ in
  ‖ $SyncNor[in_1, in_2, out := set, z, out]$
  ‖ $SyncNor[in_1, in_2, out := reset, out, z]$
```

not a legal definition of a module? (c) Consider the module

```
module $SyncDelay$ is
  private $state : \mathbb{B}$
  interface $out : \mathbb{B}$
  external $in : \mathbb{B}$
  atom $ComputeOutput$ controls $out$ reads $state$
  atom $ComputeNextState$ controls $state$ awaits $in$
    initupdate
      $[\!] \ true \rightarrow state' := in'$
```

which shares the atom *ComputeOutput* with the module *SyncLatch* from Figure 1.17. Give a few initialized trajectories of the module *SyncDelay*. Then characterize, in precise words, the set of all initialized trajectories of *SyncDelay*. Is the module *SyncDelay* finite? Closed? Deterministic? Privately determinis-

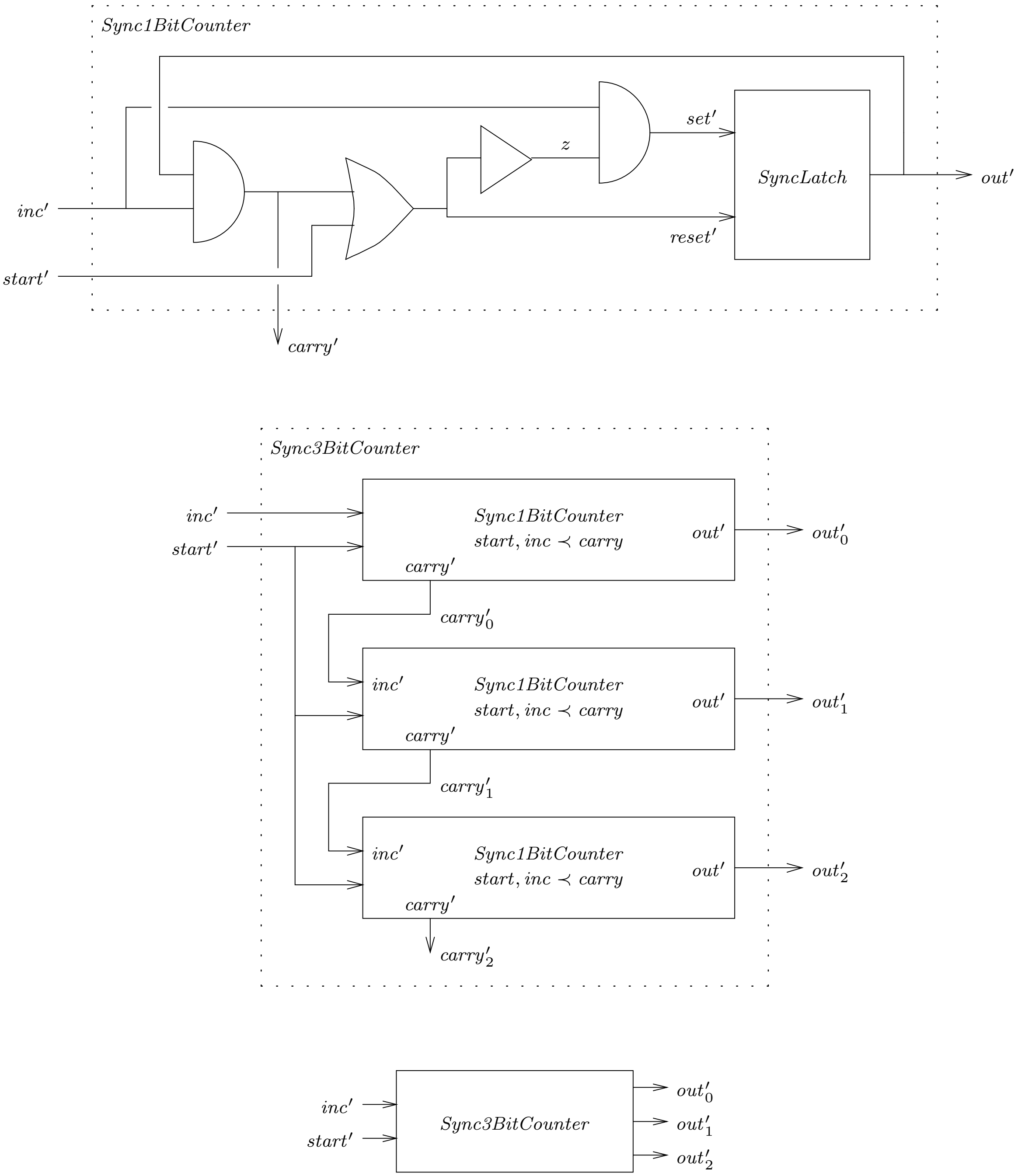


Figure 1.20: Block diagrams for the one-bit and three-bit binary counters

inc	1	0	1	1	1	1	1	1	0	0	0	1	1	0	1
start	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
z_0	0	1	1	0	0	1	0	1	1	1	1	0	1	0	1
set_0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	1
$reset_0$	1	0	0	1	1	0	1	0	0	0	0	1	0	1	0
$carry_0$	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0
z_1	0	1	1	1	0	1	1	1	1	1	1	0	1	0	1
set_1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
$reset_1$	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0
$carry_1$	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
z_2	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1
set_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$reset_2$	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0
$carry_2$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
out_0	1	0	0	1	0	0	1	0	1	1	1	1	0	1	0
out_1	1	0	0	0	1	0	0	1	1	1	1	1	0	0	0
out_2	0	1	1	1	1	0	0	0	0	0	0	0	1	1	0

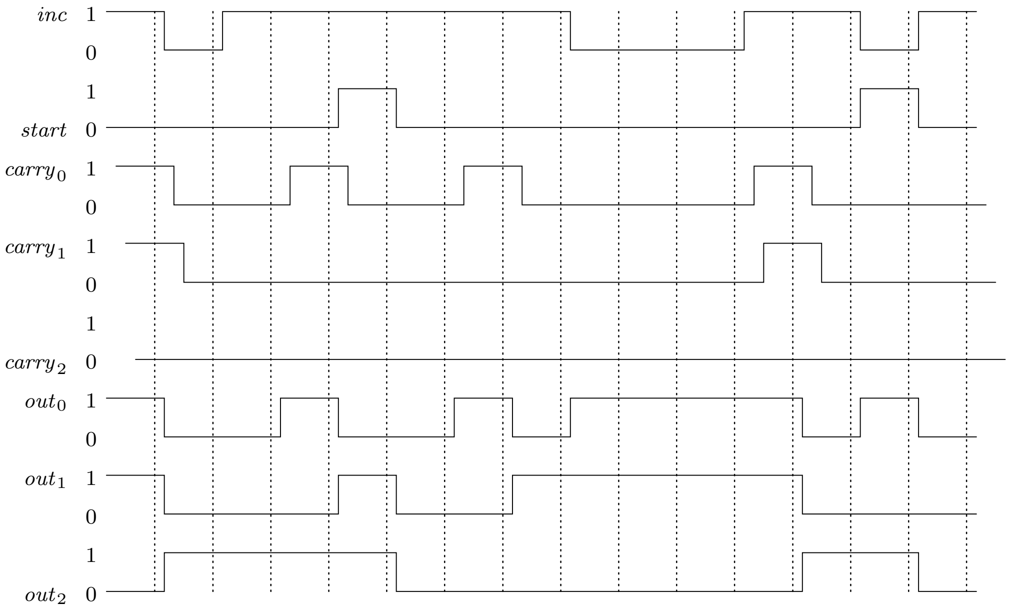


Figure 1.21: An initialized trajectory of the module *Sync3BitCounter*

tic? Asynchronous? Passive? (d) Draw block diagrams for the module

```
module $SyncLatch2$ is
  —interface $out$
  —external $set, reset$
  hide $z_1, z_2, z_3$ in
    $\|$ $SyncNor[in_1, in_2, out := set, z_3, z_1]$
    $\|$ $SyncNor[in_1, in_2, out := reset, out, z_2]$
    $\|$ $SyncDelay[in, out := z_1, out]$
    $\|$ $SyncDelay[in, out := z_2, z_3]$
```

at three different levels of abstraction. Compare the abstract type of the module *SyncLatch2* with the abstract type of the module *SyncLatch*. Give an initialized trajectory of *SyncLatch2* and draw its timing diagram. How do the traces of *SyncLatch2* differ from the traces of *SyncLatch*? Can the traces of *SyncLatch* be matched by removing one of the component modules from the compound module *SyncLatch2*? ■

1.3.2 Shared-variables Protocols

We refer to concurrent programs that communicate through read-shared variables as *processes*. We model each process as a reactive module with a single atom. The sequential control of a process is often encoded by a controlled variable called *pc*, which stands for "program counter." The interface variables of a process can be read by other processes. All inter-process communication occurs in this way, by processes reading (rather than awaiting) external variables, which are controlled by other processes. Hence, there are no awaited variables: in every update round, each process determines the next values of the controlled variables based solely on the current values of variables. Processes are combined by applying the three module operations of parallel composition, variable renaming, and variable hiding. In the synchronous case, all processes proceed in lock-step —one step per update round. In the asynchronous case, each process may or may not proceed in any given update round.

The mutual-exclusion problem

A paradigmatic problem in concurrent programming is the mutual-exclusion problem, which asks for a programming solution to ensure that no two processes simultaneously access a common resource, such as an I/O device or a write-shared variable. We illustrate the use of reactive modules for modeling concurrent programs by modeling two protocols —one synchronous, the other asynchronous— which solve the mutual-exclusion problem. We restrict our attention to the two-process case. Without loss of generality, we assume that each process has a so-called "critical section," which contains all accesses to the common resource. The interface variable pc_1 of the first process indicates if the process control is outside the critical section ($pc_1 = outC$), requesting to enter the critical section ($pc_1 = reqC$), or inside the critical section ($pc_1 = inC$). The interface variable pc_2 of the second process indicates the status of the second process in the same manner. Each process starts outside its critical section: the initial command for pc_1 is

```
init
  $[\!] \ true \to pc_1' := outC$
```

and similarly for pc_2. Each process may remain outside its critical section for an arbitrary number of rounds, and it may remain inside the critical section for an arbitrary number of rounds. In other words, each process may request to enter the critical section at any time, and it may leave the critical section at any time. We model these assumptions using nondeterminism: the update command for pc_1 contains the guarded assignments

```
update
  $[\!] \ pc_1 = outC \to$
  $[\!] \ pc_1 = outC \to pc_1' := reqC$
  $[\!] \ pc_1 = inC \quad \to$
  $[\!] \ pc_1 = inC \quad \to pc_1' := outC$
```

and similarly for pc_2. Since the program counters pc_1 and pc_2 are interface variables, in every update round, each process "knows" about the current status of the other process. Our task is to add guarded assignments that permit each process to enter its critical section, by updating pc_1 or pc_2 from $reqC$ to inC, in a controlled fashion. We say that the i-th process has the *opportunity* to enter the critical section if the guard is true for some guarded command that sets pc_i to inC. (If a process has the opportunity to enter the critical section, the process does not necessarily need to enter, because it may have additional nondeterministic choices.)

In a correct solution to the mutual-exclusion problem, the parallel composition of both processes has to meet several requirements. First and foremost is the requirement of *mutual exclusion*: it must not happen, ever, that both processes are inside their critical sections simultaneously. The mutual-exclusion requirement can be enforced easily, say, by never permitting the second process to enter the critical section. This, however, is not a satisfactory solution and is ruled out by the following, second requirement, which is called *accessibility* if either of the processes requests to enter the critical section, then in the current or some future round, the process will have the opportunity to enter; furthermore, this opportunity will present itself no matter how the other process behaves —i.e., how it resolves its nondeterministic choices— as long as the other process does not stay inside the critical section forever.

Synchronous mutual exclusion

The protocol *SyncMutex* of Figure 1.22 provides a synchronous solution to the mutual-exclusion problem. The first process, Q_1, proceeds into its critical section when the second process is not in its critical section (but may be requesting to enter), and the second process, Q_2, proceeds into its critical section when the first process is outside its critical section (and not requesting to enter). In particular, if both processes are trying to enter their critical sections in the same round, only the first process will succeed. In that case, the second process will enter its critical section as soon as the first process leaves its critical section. This guarantees accessibility. Both processes proceed synchronously, because if a process tries to enter its critical section, then it will proceed into the critical section in the first round in which it is permitted to do so.

Exercise 1.8 {P2} [Synchronous mutual exclusion] The module *SyncMutex* is active, because if a process is permitted to enter its critical section in the first round in which it tries to enter, then the process proceeds into the critical section in the same round, without waiting for a change in the value of the external variable. Modify the protocol *SyncMutex* to obtain a synchronous, passive solution to the mutual-exclusion problem. (Do not use additional variables.) ■

Asynchronous mutual exclusion

In the asynchronous model of concurrent programming, all processes proceed at independent, and possibly varying, speeds. The assumption of *speed independence* abstracts details about the execution of a concurrent program: it captures parallel implementations on multiple processors of unknown speeds, as well as time-sharing implementations on a single processor with an unknown scheduling policy. In reactive modules, each speed-independent process is specified by a lazy atom. Then, in every update round, each speed-independent process may either *proceed* (i.e., the values of some controlled variables change) or *sleep* (i.e., the values of all controlled variables stay unchanged). For concurrent programs with read-shared variables, the assumption of speed independence is captured formally by the following definition.

Speed-independent process set

A *speed-independent process* is a lazy atom without awaited variables. A *speed-independent process set* is a module all of whose atoms are speed-independent processes.

Remark 1.14 [Properties of speed-independent process sets] Every speed-independent process set is both asynchronous and passive. The speed-independent process sets are closed under parallel composition, variable renaming, and variable hiding; that is, if these operations are applied to speed-independent process sets, then the results are again speed-independent process sets. ■

```
module $Q_1$ is
  interface $pc_1 \colon \{outC, reqC, inC\}$
  external $pc_2 \colon \{outC, reqC, inC\}$
  atom controls $pc_1$ reads $pc_1, pc_2$
    init
      $[\!] \; true \to pc_1' := outC$
    update
      $[\!] \; pc_1 = outC$                        $\to$
      $[\!] \; pc_1 = outC$                        $\to pc_1' := reqC$
      $[\!] \; pc_1 = reqC \wedge pc_2 \neq inC$   $\to pc_1' := inC$
      $[\!] \; pc_1 = inC$                         $\to$
      $[\!] \; pc_1 = inC$                         $\to pc_1' := outC$

module $Q_2$ is
  interface $pc_2 \colon \{outC, reqC, inC\}$
  external $pc_1 \colon \{outC, reqC, inC\}$
  atom controls $pc_2$ reads $pc_1, pc_2$
    init
      $[\!] \; true \to pc_2' := outC$
    update
      $[\!] \; pc_2 = outC$                        $\to$
      $[\!] \; pc_2 = outC$                        $\to pc_2' := reqC$
      $[\!] \; pc_2 = reqC \wedge pc_1 = outC$     $\to pc_2' := inC$
      $[\!] \; pc_2 = inC$                         $\to$
      $[\!] \; pc_2 = inC$                         $\to pc_2' := outC$

module $SyncMutex$ is $Q_1 \parallel Q_2$
```

Figure 1.22: Synchronous mutual exclusion

```
module $P_1$ is
  interface $pc_1\colon \{outC, reqC, inC\};\ x_1\colon \mathbb{B}$
  external $pc_2\colon \{outC, reqC, inC\};\ x_2\colon \mathbb{B}$
  lazy atom controls $pc_1, x_1$ reads $pc_1, pc_2, x_1, x_2$
    init
      $\|\ true \to pc_1' := outC;\ x_1' := \mathbb{B}$
    update
      $\|\ pc_1 = outC$                                      $\to pc_1' := reqC;\ x_1' := x_2$
      $\|\ pc_1 = reqC \wedge (pc_2 = outC \vee x_1 \neq x_2)$ $\to pc_1' := inC$
      $\|\ pc_1 = inC$                                       $\to pc_1' := outC$

module $P_2$ is
  interface $pc_2\colon \{outC, reqC, inC\};\ x_2\colon \mathbb{B}$
  external $pc_1\colon \{outC, reqC, inC\};\ x_1\colon \mathbb{B}$
  lazy atom controls $pc_2, x_2$ reads $pc_1, pc_2, x_1, x_2$
    init
      $\|\ true \to pc_2' := outC;\ x_2' := \mathbb{B}$
    update
      $\|\ pc_2 = outC$                                      $\to pc_2' := reqC;\ x_2' := \neg x_1$
      $\|\ pc_2 = reqC \wedge (pc_1 = outC \vee x_1 = x_2)$    $\to pc_2' := inC$
      $\|\ pc_2 = inC$                                       $\to pc_2' := outC$

module $Pete$ is hide $x_1, x_2$ in $P_1 \,\|\, P_2$
```

Figure 1.23: Asynchronous mutual exclusion

	initial	P_2	P_1, P_2			P_2	P_2	P_1, P_2	P_1	P_2
pc_1	*outC*	*outC*	*reqC*	*reqC*	*reqC*	*reqC*	*reqC*	*inC*	*outC*	*outC*
pc_2	*outC*	*reqC*	*inC*	*inC*	*inC*	*outC*	*reqC*	*reqC*	*reqC*	*inC*
x_1	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
x_2	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*

Figure 1.24: An initialized trajectory of the module *Pete*

The previous, synchronous solution to the mutual-exclusion problem violates speed independence. An asynchronous solution, which must take the form of a speed-independent process set, is more difficult to devise and understand. The protocol *Pete* of Figure 1.23 provides the asynchronous solution due to Peterson, in which each process employs an additional boolean interface variable (x_1 and x_2, respectively). If both processes are trying to enter their critical sections in the same round, then the first process can succeed if $x_1 \neq x_2$, and the second process can succeed if $x_1 = x_2$. This guarantees the mutual-exclusion requirement. However, in contrast to the simple protocol *SyncMutex*, it is not obvious that *Pete* meets the accessibility requirement; indeed, much of this book will be devoted towards developing algorithms for checking if a finite module like *Pete* meets a requirement like accessibility. Figure 1.24 shows a sample initialized trajectory of *Pete*. Since the two atoms of *Pete* are speed-independent processes, in any given update round, either none, one, the other, or both atoms may sleep. The first line of Figure 1.24 indicates for every update round which processes proceed.

Exercise 1.9 {`T3`} [Accessibility for Peterson's protocol] Prove that the module *Pete* meets the accessibilty requirement. (As with the proofs required by other exercises, your aim need not be a derivation in some formal calculus, but an argument that is sufficiently rigorous and detailed as to convince the reader and, more importantly, yourself.) ■

Exercise 1.10 {`P3`} [Three-process mutual exclusion] You are asked to generalize Peterson's protocol to the case of three processes: first specify the three-process mutual-exclusion problem; then present your solution in the form of a finite module which is a speed-independent three-process set. Give an initialized trajectory of your protocol along which each process enters the critical section at least once (annotate the trajectory, as in Figure 1.24, with the processes that proceed during the update rounds). ■

Exercise 1.11 {`P3`} [Interleaving model] Peterson's protocol was originally designed under the *interleaving assumption* that in every update round at least one of the two processes sleeps. The interleaving assumption is stronger (more restrictive) than the assumption of speed independence, which permits update rounds in which both processes proceed. (a) Implement Peterson's original protocol using three synchronous, passive modules, two of which represent the two processes. The third module represents a scheduler which, in every update round, nondeterministically determines which of the two processes sleeps. The decision of the scheduler is communicated to the processes through an auxiliary variable that is controlled by the scheduler and awaited by the processes. After parallel composition, hide the auxiliary variable to obtain an asynchronous protocol which has the same abstract type as *Pete* and strictly fewer traces. Give a trace of *Pete* which is not a trace of your new protocol. (b) Every asynchronous protocol that meets the mutual-exclusion requirement under the assumption of speed independence also meets the mutual-exclusion requirement under the interleaving assumption. Can you find an asynchronous protocol, in the form of a speed-independent two-process set, which violates the mutual-exclusion requirement, but does so only along initialized trajectories that contain at least one update round in which both processes proceed? ■

1.3.3 Message-passing Protocols

We refer to distributed programs that communicate through messages as *agents*. The transmission of messages is governed by message-passing protocols, which ensure that all messages that are sent are also received. We illustrate the use of reactive modules for modeling distributed programs by modeling several protocols for passing messages between agents.

Event variables

We refer to every change in the value of a variable x as an x *event*. Thus, in every update round, an x event either happens or does not happen; the x events may happen as rarely as never, or as often as once per update round. In this spirit, in reactive modules we model the happening of a *pure event* — a happening without value, such as an individual clock tick or the fact that a message is being transmitted from a sender to a receiver— by toggling a boolean variable. Suppose, for example, that the boolean variable *tick* is used for modeling clock ticks. Then, the pure event "clock tick" happens whenever the value of *tick* changes either from *true* to *false*, or from *false* to *true*. In those update rounds in which the next value of *tick* is equal to the current value, no clock tick happens. In other words, the clock ticks are represented by the *tick* events.

If a boolean variable x is used to model pure events, then we are interested in all changes to the value of x, but the actual value of x at the beginning or end of any given round is irrelevant. Hence, Rml provides a special type, denoted $\mathbb{E}$, for the modeling of pure events. The variables of type $\mathbb{E}$ are called *event variables*. Each event variable ranges over the set $\mathbb{B}$ of boolean values, but compared to boolean variables, the initialization and updating of event variables is strongly restricted. The initialization of event variables is implicit: each event variable is initialized nondeterministically to either *true* or *false*. In update commands, an event variable x can occur only in the following two ways. First, the Rml expression $x!$ stands for the assignment $x' := \neg x$, which issues an x event. (If $x!$ is absent from a guarded assignment, then $x' := x$ by default, and no x event is issued.) It follows that the atom that controls x must read x. Second, the Rml expression $x?$ stands for the boolean expression $x' \neq x$, which checks if an x event is happening. It follows that an atom that does not control x, reads x if and only if it awaits x. Given a module P, we write $\mathsf{event}X_P$ for the set of event variables of P.

Synchronous communication

We are given two agents —a sender and a receiver. The sender produces a message, then sends the message to the receiver and produces another message, etc. The receiver, concurrently, waits to receive a message, then consumes the message and waits to receive another message, etc. We model each agent as a module that cannot observe the control variables of the other agent. The private variable *pc* of the sender indicates if the agent is producing a message (*pc* = *produce*) or attempting to send a message (*pc* = *send*). The sender starts by producing a message:

```
init
  ‖ $true \rightarrow pc' := produce$
```

Messages are produced by the lazy atom *Producer*, which requires an unknown number of rounds to produce a message. Once a message is produced, the producer issues a $done_P$ event (which is private to the sender) and the produced message is shown as msg_P (which initially is undefined). We assume that messages have the finite type $\mathbb{M}$, and that any stream of messages from the finite set $\mathbb{M}$ may be produced. We model these assumptions using nondeterminism:

```
lazy atom Producer controls $done_P, msg_P$ reads $pc, done_P$
  init
    ‖ $true \rightarrow msg'_P := \bot$
  update
    ‖ $pc = produce \rightarrow done_P!;\ msg'_P := \mathbb{M}$
```

Once a message has been produced, the sender is ready to send the message:

```
update
  ‖ $pc = produce \wedge done_P? \rightarrow pc' := send$
```

The private variable *pc* of the receiver indicates if the agent is waiting to receive a message (*pc* = *receive*) or consuming a message (*pc* = *consume*). The receiver starts by waiting to receive a message:

```
init
  ▯ $true \rightarrow pc' := receive$
```

The received message is stored in the private variable msg_R. Messages are consumed by the lazy atom *Consumer*, which requires an unknown number of rounds to consume a message. Once a message is consumed, the consumer issues a $done_C$ event (which is private to the receiver) and the consumed message is shown as msg_C (which initially is undefined):

```
lazy atom Consumer controls $done_C, msg_C$ reads $pc, done_C, msg_R$
  init
    ▯ $true \rightarrow msg'_C := \bot$
  update
    ▯ $pc = consume \rightarrow done_C!;\ msg'_C := msg_R$
```

Once a message has been consumed, the receiver waits to receive another message:

```
update
  ▯ $pc = consume \wedge done_C? \rightarrow pc' := receive$
```

Our task is to add guarded assignments that permit the sender to send a message, by updating *pc* from *send* to *produce*, and guarded assignments that permit the receiver to receive a message, by updating *pc* from *receive* to *consume*, in a controlled fashion. Roughly speaking, when composing both agents, the stream of consumed messages msg_C should contain the same message values, in the same order, as the stream of produced messages msg_P. Formal requirements for message-passing protocols will be stated in Chapter ??.

The protocol *SyncMsg* of Figure 1.25 has the sender and the receiver synchronize to transmit a message; that is, when ready to send a message, the sender is blocked until the receiver becomes ready to receive, and when ready to receive a message, the receiver is blocked until the sender transmits a message. The synchronization of both agents is achieved by two-way handshaking in three subrounds (or “stages”) within a single update round. The first subround belongs to the atom *Stage1* of the receiver. If the receiver is ready to receive a message, it asynchronously issues an interface *ready* event to signal its readiness to the sender. The second subround belongs to the atom *Stage2* of the sender. If the sender sees an external *ready* event and is ready to send a message, it synchronously issues an interface *transmit* event to signal a transmission, and it offers the message which is to be transmitted in the interface variable msg_S. The third subround belongs to the atom *Stage3* of the receiver. If the receiver sees an external *transmit* event, it copies the message from the external variable msg_S to the private variable msg_R. The three-stage, two-way handshaking structure

```
module $SyncSender$ is
  private $pc\colon \{produce, send\}$;  $done_P\colon \mathbb{E}$
  interface $transmit\colon \mathbb{E}$;  $msg_S\colon \mathbb{M}$;  $msg_P\colon \mathbb{M}_\perp$
  external $ready\colon \mathbb{E}$

  passive atom $Stage2$
    controls $pc, transmit, msg_S$
    reads $pc, done_P, ready, transmit, msg_P$
    awaits $done_P, ready$
    init
      $\| \; true \rightarrow pc' := produce;\; msg'_S := \mathbb{M}$
    update
      $\| \; pc = produce \wedge done_P? \rightarrow pc' := send$
      $\| \; pc = send \wedge ready? \qquad \rightarrow transmit!;\; msg'_S := msg_P;\; pc' := produce$

  lazy atom $Producer$ controls $done_P, msg_P$ reads $pc, done_P$

module $Receiver$ is
  private $pc\colon \{receive, consume\}$;  $msg_R\colon \mathbb{M}$;  $done_C\colon \mathbb{E}$
  interface $ready\colon \mathbb{E}$;  $msg_C\colon \mathbb{M}_\perp$
  external $transmit\colon \mathbb{E}$;  $msg_S\colon \mathbb{M}$

  passive atom $Stage3$
    controls $pc, msg_R$
    reads $pc, transmit, done_C$
    awaits $transmit, msg_S, done_C$
    init
      $\| \; true \rightarrow pc' := receive;\; msg'_R := \mathbb{M}$
    update
      $\| \; pc = receive \wedge transmit? \rightarrow msg'_R := msg'_S;\; pc' := consume$
      $\| \; pc = consume \wedge done_C? \rightarrow pc' := receive$

  lazy atom $Stage1$ controls $ready$ reads $pc, ready$
    update
      $\| \; pc = receive \rightarrow ready!$

  lazy atom $Consumer$ controls $done_C, msg_C$ reads $pc, done_C, msg_R$

module $SyncMsg$ is
  —interface $msg_P, msg_C$
  hide $ready, transmit, msg_S$ in
    $\|$ $SyncSender$
    $\|$ $Receiver$
```

Figure 1.25: Synchronous message passing

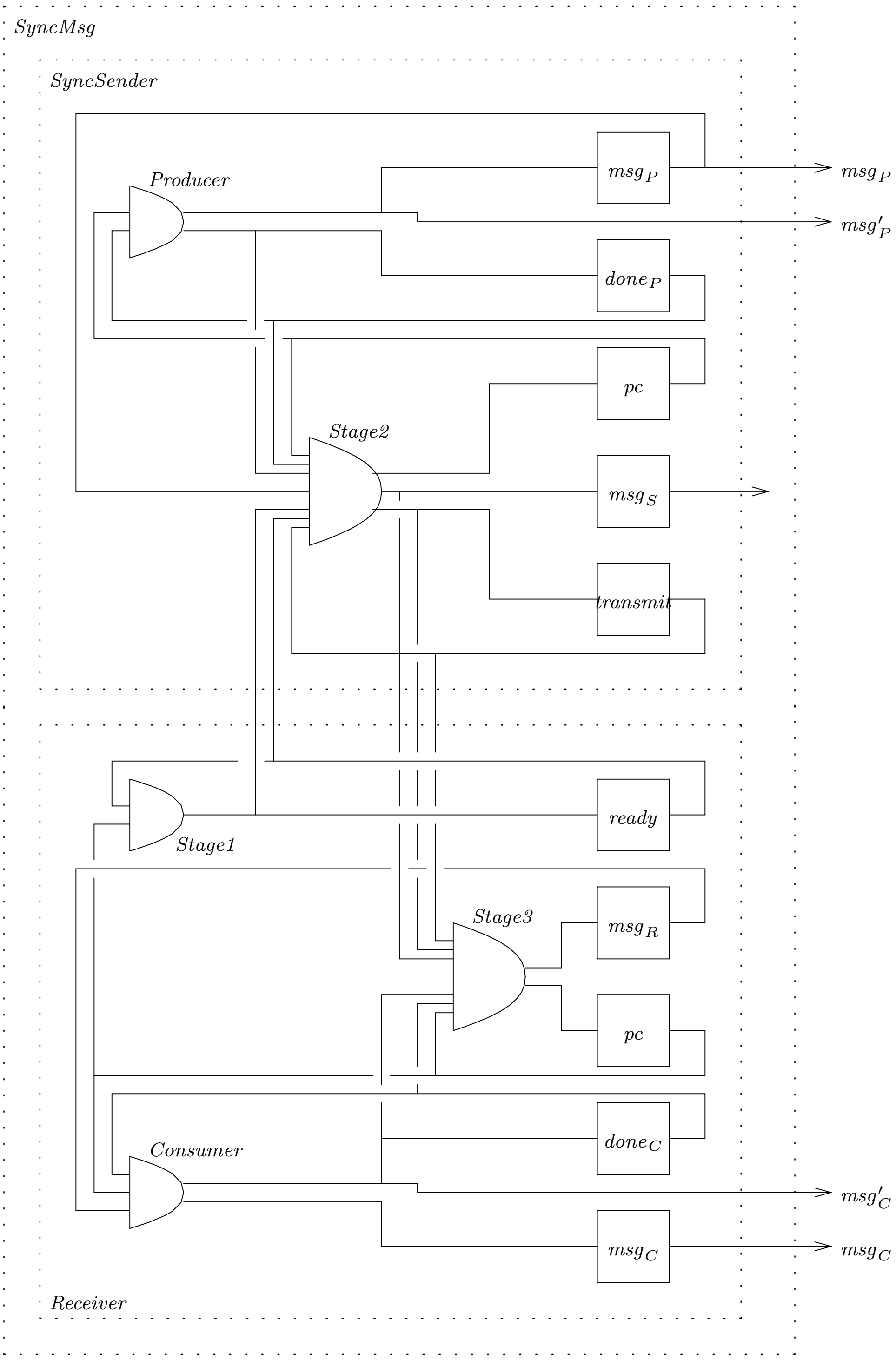


Figure 1.26: Block diagram for synchronous message passing

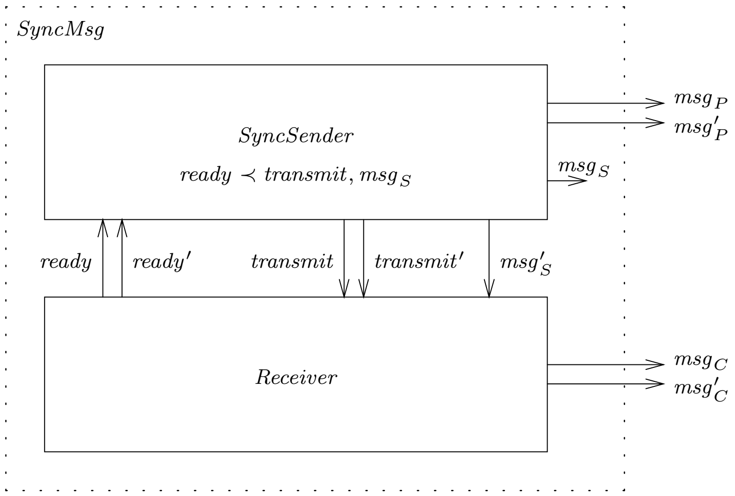


Figure 1.27: Abstract block diagram for synchronous message passing

of the protocol *SyncMsg* can be seen in the block diagram of Figure 1.26 and in the abstract block diagram of Figure 1.27.

Both agents *SyncSender* and *Receiver* are passive, because the sender may sleep in any given update round except when the receiver signals *ready*, and the receiver may sleep in any given update round except when the sender signals *transmit*. The sender is synchronous, because it signals *transmit* in the very round in which the receiver signals *ready*; the receiver is asynchronous. The entire protocol *SyncMsg*, after hiding the variables *transmit* and msg_S, is both asynchronous and passive. Figure 1.28 shows a sample initialized trajectory of the module *SyncMsg*, assuming that the possible values of messages are $\mathbb{M} = \{\text{A}, \text{B}, \text{C}\}$. (The two program counters have been renamed implicitly, and their values are abbreviated. Instead of giving the values of event variables, the figure indicates when the corresponding events happen.) The corresponding trace of *SyncMsg* consists of a stream of produced messages msg_P and a stream of consumed messages msg_C.

Exercise 1.12 {P2} [Synchronous message passing] (a) Give a few additional initialized trajectories of the module *SyncMsg* and the corresponding traces. Then characterize, in precise words, the set of all traces of *SyncMsg*. (b) Define a module that has the same abstract type and the same traces as the module *SyncMsg*, but as few private variables as possible. ■

Exercise 1.13 {P3} [Dining philosophers] Suppose that there are two rooms and n philosophers. In one room, the philosophers think; in the other room,

pc_S	p	p	s	s	p	s	s	s	p	p	p	s	s	p	s
transmit					◇				◇					◇	
msg_S	C	C	C	C	A	A	A	A	B	B	B	B	B	C	C
$done_P$			◇			◇						◇			◇
msg_P	$\perp$	$\perp$	A	A	A	B	B	B	B	B	B	C	C	C	B
pc_R	r	r	r	r	c	c	c	r	c	r	r	r	r	c	r
ready		◇			◇				◇		◇	◇		◇	
msg_R	B	B	B	B	A	A	A	A	B	B	B	B	B	C	C
$done_C$								◇		◇					◇
msg_C	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	A	A	B	B	B	B	B	C

msg_P	$\perp$	$\perp$	A	A	A	B	B	B	B	B	B	C	C	C	B
msg_C	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	A	A	B	B	B	B	B	C

Figure 1.28: An initialized trajectory of the module *SyncMsg* and the corresponding trace

they eat while seated at a round table. Every philosopher owns one of the n chairs at the table. There is one chopstick between each of the n plates, and every philosopher uses both the left and the right chopstick for eating (it follows that at most $\lfloor n/2 \rfloor$ philosophers can be eating at the same time). Every philosopher begins by thinking and, when hungry, enters the dining room. There, the philosopher sits down at the table at the designated chair, picks up the chopstick to the left (or waits until it becomes available), and then the chopstick to the right (or waits). Once in control of both chopsticks, the philosopher eats, then releases both chopsticks, leaves the dining room, thinks, and returns when hungry again.

The passive module *Stick* of Figure 1.29 implements a chopstick. The private variable *pc* indicates if the chopstick is available ($pc = \mathit{free}$), picked up by the philosopher to the left ($pc = \mathit{left}$), or picked up by the philosopher to the right ($pc = \mathit{right}$). An external req_L event indicates that the philosopher to the left requests the chopstick, an interface $grant_L$ event indicates that the philosopher to the left picks up the chopstick, and an external $release_L$ event indicates that the philosopher to the left releases the chopstick. The event variables req_R, $grant_R$, and $release_R$ refer to the philosopher to the right. (a) Define a passive module *Phil* which implements a philosopher and, using multiple, renamed copies of *Phil* and *Stick*, define a compound module *Dine4* which implements the dining-philosophers scenario for $n = 4$. Illustrate the communication structure of the module *Dine4* by drawing block diagrams for *Phil*, *Stick*, and *Dine4* at suitable levels of abstraction. (b) Give an initialized trajectory of your module *Dine4* which ends up in a situation where all 4 philosophers sit at the table,

```
module $Stick$ is
  private $pc\colon \{free, left, right\}$
  interface $grant_L, grant_R\colon \mathbb{E}$
  external $req_L, release_L, req_R, release_R\colon \mathbb{E}$
  passive atom
    controls $pc, grant_L, grant_R$
    reads $pc, req_L, grant_L, release_L, req_R, grant_R, release_R$
    awaits $req_L, release_L, req_R, release_R$
    init
      $[\!] \; true \to pc' := free$
    update
      $[\!] \; pc = free \wedge req_L?$         $\to grant_L!;\ pc' := left$
      $[\!] \; pc = free \wedge req_R?$         $\to grant_R!;\ pc' := right$
      $[\!] \; pc = left \wedge release_L?$     $\to pc' := free$
      $[\!] \; pc = right \wedge release_R?$    $\to pc' := free$
```

Figure 1.29: A chopstick for the dining philosophers

have picked up one chopstick, and wait for the other chopstick to become available. There are several ways to prevent this deadlock situation. (b1) Have each philosopher pick up both chopsticks simultaneously (or wait until both chopsticks become available). (b2) Add to the entrance of the dining room a guard that admits at most $n - 1 = 3$ philosophers into the dining room at any given time. Define a passive module *Guard* and draw the abstract block diagram for the dining-philosophers scenario with a guard. (Hide all communication between the philosophers and the guard so that the resulting module has the same abstract type as the module *Dine4*.) ■

Exercise 1.14 {P3} [Write-shared variables] Consider a concurrent program with two processes, R_1 and R_2, both of which have read and write access to a boolean variable x. When R_1 wishes to read the value of x, it issues an interface $read_1$ event and expects, depending on the current value of x, either an external *return_true* event or an external *return_false* event within the same round. Similarly, when R_2 wishes to read x, it issues an interface $read_2$ event and expects an external *return_true* or *return_false* event within the same round. When R_1 wishes to assign the value i to x, for $i \in \{true, false\}$, it issues an interface $write_1\_i$ event. Similarly, when R_2 wishes to assign the value i to x, it issues an interface $write_2\_i$ event. If both R_1 and R_2 issue conflicting write requests, then x is updated nondeterministically to one of the written values. (a) Define a passive module R_x for modeling the shared variable x. Your module should have the private boolean variable x, the two interface event variables *return_true* and *return_false*, and the external event variables $read_1$, $read_2$, $write_1\_true$,

$write_1\_false$, $write_2\_true$, and $write_2\_false$. (b) Give an alternative implementation *Pete2* of Peterson's mutual-exclusion protocol (Figure 1.23) which uses instead of the two read-shared boolean variables x_1 and x_2 the single write-shared boolean variable x. Define two modules R_1 and R_2 for modeling the two processes of the protocol, encoding $x_1 = x_2$ by $x = true$ and $x_1 \neq x_2$ by $x = false$. The resulting module

module *Pete2* **is hide** ... **in** $R_1 \parallel R_2 \parallel R_x$

should have the same abstract type and the same traces as the module *Pete*. (c) How would you change the definition of R_x in part (a) if x is a nonnegative-integer variable rather than a boolean variable? ■

Asynchronous communication

While the synchronous message-passing protocol *SyncMsg* of Figure 1.26 performs a two-way handshake within a single round, the asynchronous protocol *AsyncMsg* of Figure 1.30 uses many rounds for a single handshake. The two protocols have identical receiver agents. Every send-receive cycle of *AsyncMsg* consists of four phases —a message production phase, an agent coordination phase, a message transmission phase, and a message consumption phase— each consisting of any number of update rounds. During the message production phase, the sender ($pc = produce$) takes an unknown number of rounds to produce a message. During the agent coordination phase, the sender ($pc = wait$) waits for an external *ready* event, which signals the readiness of the receiver to receive a message. The receiver ($pc = receive$) takes an unknown number of rounds to issue an interface *ready* event. During the message transmission phase, $pc = send$ for the sender and $pc = receive$ for the receiver. The sender takes an unknown number of rounds to transmit the message, asynchronously issuing an interface *transmit* event and simultaneously offering the message in the interface variable msg_S. The receiver, ready to receive, sees the external *transmit* event and copies the message from the external variable msg_S to the private variable msg_R. During the message consumption phase, the receiver ($pc = consume$) takes an unknown number of rounds to consume the message. The message consumption phase overlaps with the ensuing message production phase, which initiates a new send-receive cycle.

Exercise 1.15 {P2} [Asynchronous message passing] (a) Draw block diagrams for the module *AsyncMsg* at several levels of abstraction. (b) Give a few initialized trajectories of the module *AsyncMsg* and the corresponding traces. Then characterize, in precise words, the set of all traces of *AsyncMsg*. (c) How do the traces of the module *AsyncMsg* differ from the traces of the module *SyncMsg* from Exercise 1.12? ■

Exercise 1.16 {P4} [Faulty communication] Suppose that the delivery of messages may be delayed in a communication medium, and that messages may be

```
module $AsyncSender$ is
  private $pc\colon \{produce, send, wait\}$;  $done_P\colon \mathbb{E}$
  interface $transmit\colon \mathbb{E}$;  $msg_S\colon \mathbb{M}$;  $msg_P\colon \mathbb{M}_\perp$
  external $ready\colon \mathbb{E}$

  passive atom $ProgramCounter$
    controls $pc$
    reads $pc, done_P, ready, transmit$
    awaits $done_P, ready, transmit$
    init
      [] $true \rightarrow pc' := produce$
    update
      [] $pc = produce \wedge done_P? \rightarrow pc' := wait$
      [] $pc = wait \wedge ready?$      $\rightarrow pc' := send$
      [] $pc = send \wedge transmit?$  $\rightarrow pc' := produce$

  lazy atom $Transmitter$
    controls $transmit, msg_S$
    reads $pc, transmit, msg_P$
    init
      [] $true \rightarrow msg'_S := \mathbb{M}$
    update
      [] $pc = send \rightarrow transmit!$;  $msg'_S := msg_P$

  lazy atom $Producer$ controls $done_P, msg_P$ reads $pc, done_P$

module $AsyncMsg$ is
  —interface $msg_P, msg_C$
  hide $ready, transmit, msg_S$ in
    $\|$ $AsyncSender$
    $\|$ $Receiver$
```

Figure 1.30: Asynchronous message passing

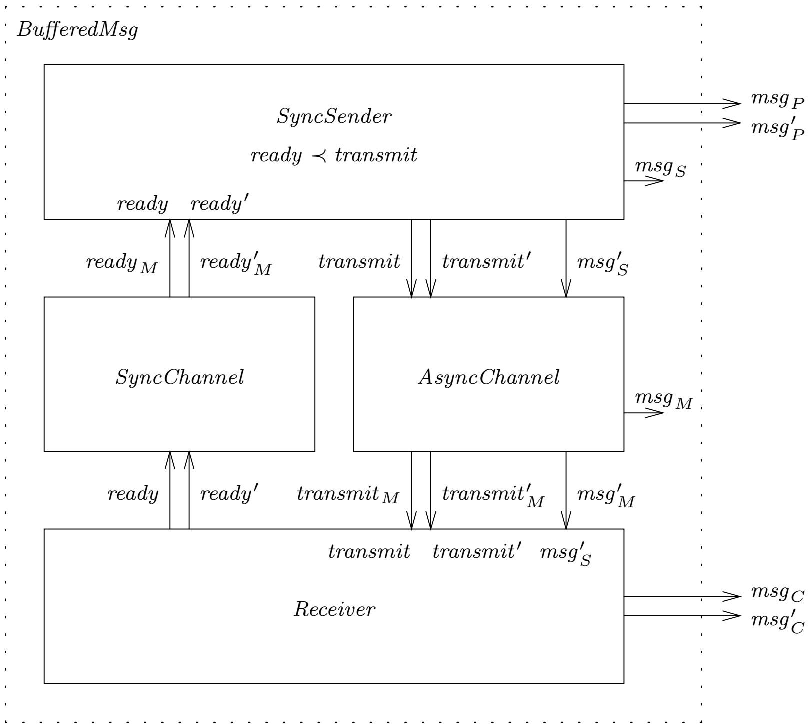


Figure 1.31: Message passing through channels

reordered, lost, and corrupted in the medium. We model the communication medium by two agents that are interjected between the sender and the receiver of the synchronous message-passing protocol *SyncMsg*. The signal channel *SyncChannel*, with the external variable *ready* and the interface variable $ready_M$, delivers signals from the receiver to the sender. The message channel *AsyncChannel*, with the external variables *transmit* and msg_S and the interface variables $transmit_M$ and msg_M, delivers messages from the sender to the receiver. The new message-passing protocol is implemented by the module

module *BufferedMsg* **is**
  —**interface** msg_P, msg_C
  hide $ready, ready_M, transmit, transmit_M, msg_S, msg_M$ **in**
    $\|$ $SyncSender[ready := ready_M]$
    $\|$ *SyncChannel*
    $\|$ *AsyncChannel*
    $\|$ $Receiver[transmit, msg_S := transmit_M, msg_M]$

whose abstract block diagram is shown in Figure 1.31. We wish to model both reliable and unreliable communication media with various latencies and capacities. For each of the following parts of the exercise, give at least one initialized trajectory of the module *BufferedMsg*. (a) Define a synchronous, passive module *SyncChannel* which delivers signals without delay (every signal is delivered in the round in which it is sent). Then define an asynchronous, passive module *AsyncChannel* which delays messages arbitrarily (by an unknown number of rounds), and delivers the messages in the order in which they are sent. (b) Modify the module *AsyncChannel* so that the messages are not necessarily delivered in the order in which they are sent. (b1) Assume that a message can be overtaken by at most 3 newer messages. (b2) Assume that a message can be overtaken by an arbitrary number of newer messages. (c) Modify the module *AsyncChannel* so that messages may be lost. (c1) Assume that any message may be lost. (c2) Assume that the message channel has a capacity of 5 messages; that is, in any given round, the channel can store at most 5 undelivered messages. If 5 messages are stored and a new message is received from the sender, then the new message is lost. (d) Modify the module *AsyncChannel* so that messages may be reordered, lost, and corrupted (a message is corrupted if its contents is changed arbitrarily). ■

Timed communication

Instead of waiting for the receiver to be ready to receive a message, the sender may choose to retransmit a message repeatedly until the receiver acknowledges the receipt of the message. The decision to retransmit can be based upon timing: a message is retransmitted if no acknowledgment is obtained within a certain amount of time. For measuring time, we let protocols refer to an external digital clock. We model the clock as an asynchronous, passive module:

```
module AsyncClock is
  interface tick : $\mathbb{E}$
  lazy atom controls tick reads tick
    update
      [] $true \to tick!$
```

The clock module *AsyncClock* issues *tick* events at undetermined times. The *tick* events represent clock ticks and can be observed by other modules.

Consider the receiver module *TimedReceiver* of Figure 1.32. Instead of signaling when it is ready to receive a message, the module *TimedReceiver* confirms the receipt of a message by issuing an interface *ack* event for acknowledgment. If for the duration of 4 clock ticks after receiving a message, the message has not been acknowledged, then a time-out occurs. If a time-out occurs, then an acknowledgment is issued right away, in the round that immediately follows the 4th clock tick after message reception. Consequently, the receipt of every message is confirmed within at most 4 time units, as measured by the clock module

```
module $TimedReceiver$ is
  private $pc\colon \{receive, confirm, consume\}$;  $msg_R\colon \mathbb{M}$;
    $done_C, timeout\colon \mathbb{E}$;  $timer\colon [0..3]$
  interface $ack\colon \mathbb{E}$;  $msg_C\colon \mathbb{M}_\perp$
  external $transmit, tick\colon \mathbb{E}$;  $msg_S\colon \mathbb{M}$

  passive atom $ReceiverProgramCounter$
    controls $pc, msg_R, ack$
    reads $pc, transmit, timeout, ack, done_C$
    awaits $transmit, msg_S, timeout, done_C$
    init
      $[\!]\ true \to pc' := receive;\ msg'_R := \mathbb{M}$
    update
      $[\!]\ pc = receive \wedge transmit?$     $\to msg'_R := msg'_S;\ pc' := confirm$
      $[\!]\ pc = confirm \wedge \neg timeout?$ $\to$
      $[\!]\ pc = confirm$                      $\to ack!;\ pc' := consume$
      $[\!]\ pc = consume \wedge done_C?$       $\to pc' := receive$

  passive atom $ReceiverTimer$
    controls $timer, timeout$
    reads $pc, transmit, tick, timer, timeout$
    awaits $transmit, tick$
    init
      $[\!]\ true \to timer := [0..3]$
    update
      $[\!]\ pc = receive \wedge transmit?$                      $\to timer' := 0$
      $[\!]\ pc = confirm \wedge tick? \wedge timer < 3$ $\to timer' := timer + 1$
      $[\!]\ pc = confirm \wedge tick? \wedge timer = 3$ $\to timeout!$

  lazy atom $Consumer$ controls $done_C, msg_C$ reads $pc, done_C, msg_R$
```

Figure 1.32: Timed message passing: receiver

```
module $TimedSender$ is
  private $pc\colon \{produce, send, wait\}$;  $done_P, timeout\colon \mathbb{E}$;  $timer\colon [0..4]$
  interface $transmit\colon \mathbb{E}$;  $msg_S\colon \mathbb{M}$;  $msg_P\colon \mathbb{M}_\perp$
  external $ack, tick\colon \mathbb{E}$

  passive atom $SenderProgramCounter$
    controls $pc$
    reads $pc, done_P, transmit, ack, timeout$
    awaits $done_P, transmit, ack, timeout$
    init
      $\| \ true \to pc' := produce$
    update
      $\| \ pc = produce \wedge done_P? \to pc' := send$
      $\| \ pc = send \wedge transmit? \ \to pc' := wait$
      $\| \ pc = wait \wedge ack? \qquad \to pc' := produce$
      $\| \ pc = wait \wedge timeout? \ \ \to pc' := send$

  passive atom $SenderTimer$
    controls $timer, timeout$
    reads $pc, transmit, tick, timer, timeout$
    awaits $transmit, tick$
    init
      $\| \ true \to timer := [0..4]$
    update
      $\| \ pc = send \wedge transmit? \qquad\quad \to timer' := 0$
      $\| \ pc = wait \wedge tick? \wedge timer < 4 \to timer' := timer + 1$
      $\| \ pc = wait \wedge tick? \wedge timer = 4 \to timeout!$

  lazy atom $Transmitter$
    controls $transmit, msg_S$
    reads $pc, transmit, msg_P$

  lazy atom $Producer$ controls $done_P, msg_P$ reads $pc, done_P$
```

Figure 1.33: Timed message passing: sender

AsyncClock. The corresponding sender does not know when the receiver is ready to receive a message. If a message is transmitted when the receiver is not ready to receive it, then the message is lost. Therefore the sender must retransmit every message that is not acknowledged. The sender module *TimedSender* of Figure 1.33 uses the atom *Transmitter* from Figure 1.30 for transmitting a message, and then waits for an acknowledgment for exactly 5 clock ticks. The waiting period of 5 clock ticks suffices, because the receiver acknowledges the receipt of every message within at most 4 clock ticks. If the sender does not obtain an acknowledgment for the duration of 5 clock ticks, then it decides to retransmit the message. The resulting timed message-passing protocol

```
module TimedMsg is
  —interface $msg_P, msg_C$
  hide $transmit, msg_S, ack, tick$ in
    || TimedSender
    || TimedReceiver
    || AsyncClock
```

has the same abstract type as the protocols *SyncMsg* and *AsyncMsg*.

Exercise 1.17 {P2} [Timed message passing] (a) Draw block diagrams for the module *TimedMsg* at several levels of abstraction. (b) Give a few initialized trajectories of the module *TimedMsg* and the corresponding traces. Then characterize, in precise words, the set of all traces of *TimedMsg*. (c) How do the traces of the module *TimedMsg* differ from the traces of the module *AsyncMsg* from Exercise 1.15? ■

Exercise 1.18 {P2} [More timed message passing] If the worst-case durations of message transmission and message consumption are known to the sender, then there is no need for the receiver to signal its readiness to receive a message, nor to acknowledge the receipt of a message. Design a timed message-passing protocol that consists of a sender, a message channel, a receiver, and the clock module *AsyncClock*, and reflects the following three timing assumptions: (1) the production of a message requires at least 3 clock ticks, and after transmitting a message, the sender waits for 4 clock ticks before producing another message; (2) the channel takes at least 2 and at most 5 clock ticks to deliver a message; (3) every message is consumed within a single clock tick. Since $5 + 1 < 3 + 4$, the receiver can be ready to receive every message that is transmitted by the sender. The resulting message-passing protocol should have the same abstract type and the same traces as the module *AsyncMsg*. ■

1.3.4 Asynchronous Circuits*

In an asynchronous circuit, unlike synchronous circuits, there is no single global clock, and a change in the value of an output due to changes in the values of the inputs may be delayed. We model each asynchronous logic gate by a nondeterministic passive module for which a change in the values of its inputs causes a corresponding change in the value of the output after an arbitrary number of update rounds. An asynchronous logic gate is *stable* when its output is the desired function of the inputs, and *unstable* otherwise. For example, an asynchronous AND gate is stable when its output is the conjunction of both inputs. The condition

$$\textsc{And}(in_1, in_2, out) \; = \; (out = in_1 \cdot in_2)$$

is called the *stability condition* of an AND gate with inputs in_1 and in_2 and output *out*. The output of an asynchronous gate can change only if the gate is unstable; when this happens the gate becomes stable. The gate takes an unknown number of rounds to become stable. If the gate is stable, and any of the inputs change in a way that violates the stability condition, then the gate turns unstable. If the gate is unstable, and any of the inputs change without rendering the stability condition true, the gate remains unstable. If, however, any of the inputs of an unstable gate change in a way that renders the stability condition true, a hazard is encountered, and the gate fails. If a gate has failed, its output may change arbitrarily. These modeling assumptions for an asynchronous AND gate are specified by the asynchronous, passive module *AsyncAnd* of Figure 1.34. The private variable *pc* indicates the status of the gate (*stable*, *unstable*, or *hazard*) at the end of each round. The interface and external variables of *AsyncAnd* are identical to the interface and external variables of the synchronous module *SyncAnd* from Figure 1.15. However, unlike *SyncAnd*, the asynchronous module *AsyncAnd* has no (derived) await dependencies.

Exercise 1.19 {P3} [Asynchronous circuits] (a) Define an asynchronous, passive module *AsyncNot* which specifies an asynchronous NOT gate (use the variable name *in* for input, and use *out* for output). Give a few initialized trajectories of the module *AsyncNot*. Then characterize, in precise words, the set of all traces of *AsyncNot*. (b) Give a few initialized trajectories of the module

```
module $AsyncNor$ is
  hide $z_1, z_2$ in
    $\| \; AsyncAnd[in_1, in_2, out := z_1, z_2, out]$
    $\| \; AsyncNot[in, out := in_1, z_1]$
    $\| \; AsyncNot[in, out := in_2, z_2]$
```

and characterize its traces. Given our modeling assumptions, is the module *AsyncNor* a correct implementation of an asynchronous NOR gate? (How do the traces of *AsyncNor* compare with the traces of the module that results from replacing each AND condition in the module *AsyncAnd* by a NOR condition?) (c) An asynchronous latch has the two external variables *set* and *reset* and the interface variable *out*. The state of the asynchronous latch is stable when *set* $= 1$ implies that the state is 1, and *reset* $= 1$ implies that the state is 0. The

```
module $AsyncAnd$ is
  private $pc\colon \{stable, unstable, hazard\}$
  interface $out\colon \mathbb{B}$
  external $in_1, in_2\colon \mathbb{B}$

  lazy atom $Output$ controls $out$ reads $pc, out$
    init
      $[\!]\ true \rightarrow out' := \mathbb{B}$
    update
      $[\!]\ pc = unstable \rightarrow out' := \neg out$
      $[\!]\ pc = hazard \quad \rightarrow out' := \neg out$

  passive atom $Status$ controls $pc$ reads $pc, out$ awaits $in_1, in_2, out$
    init
      $[\!]\ \textsc{And}(in_1', in_2', out') \quad \rightarrow pc' := stable$
      $[\!]\ \neg\textsc{And}(in_1', in_2', out') \rightarrow pc' := unstable$
    update
      $[\!]\ pc = stable \wedge \neg\textsc{And}(in_1', in_2', out') \qquad\qquad\qquad \rightarrow pc' := unstable$
      $[\!]\ pc = unstable \wedge \textsc{And}(in_1', in_2', out') \wedge out' \neq out \rightarrow pc' := stable$
      $[\!]\ pc = unstable \wedge \textsc{And}(in_1', in_2', out') \wedge out' = out \rightarrow pc' := hazard$
```

Figure 1.34: Asynchronous AND gate

output of the asynchronous latch is stable when *out* is equal to the state of the latch. The state and the output stabilize independently, each taking an unknown number of rounds to switch from unstable to stable. A hazard is encountered if either the state is unstable and a change of the inputs renders it stable, or the output is unstable and a change of the state renders it stable. Define an asynchronous, passive module *AsyncLatch* which specifies an asynchronous latch under these modeling assumptions. Give a few initialized trajectories of the module *AsyncLatch* and characterize its traces. (d) Give a few initialized trajectories of the module

```
module $AsyncLatch2$ is
  hide $z$ in
    $\| \; AsyncNor[in_1, in_2, out := set, z, out]$
    $\| \; AsyncNor[in_1, in_2, out := reset, out, z]$
```

and characterize its traces. Given our modeling assumptions, is the module *AsyncLatch2* a correct implementation of an asynchronous latch? (How do the traces of *AsyncLatch2* compare with the traces of *AsyncLatch*?) ■

Exercise 1.20 {P3} [Explicitly clocked circuits] (a) Modify the modules *AsyncAnd* and *AsyncNot* (from Exercise 1.19) so that each gate, when unstable, stabilizes within at most 3 rounds, provided no hazard is encountered in the meantime. Are the resulting modules *ClockedAnd* and *ClockedNot* synchronous or asynchronous? Active or passive? (b) Modify the module *AsyncLatch* (from Exercise 1.19) so that the state of the latch, when unstable, stabilizes within at most 3 rounds, provided no hazard is encountered in the meantime. Furthermore, the output of the latch, when unstable, stabilizes whenever an external *tick* event occurs. The resulting module *ClockedLatch* should be synchronous and passive. Unlike the synchronous, active latch *SyncLatch* of Figure 1.17, which is implicitly clocked (every update round corresponds to a clock cycle), the synchronous, passive latch *ClockedLatch* is explicitly clocked (every external *tick* event corresponds to a clock cycle). (c) Let *Clocked3BitCounter* be the module that results from replacing every component of the module *Sync3BitCounter* from Example 1.20 as follows: replace each occurrence of *SyncAnd* by *ClockedAnd*, each occurrence of *SyncNot* by *ClockedNot*, and each occurrence of *SyncLatch* by *ClockedLatch*. Define a module *Clock*, which issues interface *tick* events, so that the compound module

hide *tick* **in** *Clocked3BitCounter* ‖ *Clock*

implements an asynchronous three-bit counter whose only hazards can be caused by primary inputs (*start* and *inc*) changing too frequently. (You need to determine the minimal frequency of clock ticks which cannot cause hazards.) ■

1.4 Bibliographic Remarks

Reactive modules were introduced by [AlurHenzinger99]. Rml is at its core a synchronous modeling language based on read-shared variables, and thus is closely related to synchronous programming languages such as Esterel by [BerryGonthier88]. In Rml, asynchrony is modeled by nondeterministic progress, and communication events are modeled by changes in the values of variables. Paradigmatic modeling languages that are based on these alternative primitives include the asynchronous shared-variables language Unity by [ChandyMisra88], the asynchronous event-communication language I/O Automata by [Lynch96], and the synchronous event-communication languages Csp and Ccs by [Hoare85, Milner89].

Bibliography

[AlurHenzinger99] R. Alur and T.A. Henzinger, Reactive modules, *Formal Methods in System Design* 15, 1999, pp. 7–48.

[BerryGonthier88] G. Berry and G. Gonthier, *The Synchronous Programming Language* ESTEREL*: Design, Semantics, Implementation*, Technical Report 842, INRIA, 1988.

[ChandyMisra88] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[Hoare85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Lynch96] N.A. Lynch, *Distributed Computing*, Morgan-Kaufmann, 1996.

[Milner89] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

List of Exercises

Index

Contents

Chapter 2

Invariant Verification

In this chapter, we study the formulation and verification of the simplest but most important kind of system requirements, called *invariants.* An invariant classifies the states of a reactive module into *safe* and *unsafe*, and asserts that during the execution of the module, no unsafe state can be encountered.

2.1 Transition Graphs

The information about a module which is necessary for checking invariants is captured by the transition graph of the module. Consequently, invariant verification is performed on transition graphs.

2.1.1 Definition of Transition Graphs

At every point during the execution of a system, the information that is necessary to continue the execution is called the *state* of the system. The state of a discrete system changes in a sequence of update rounds. Every possible state change is called a *transition* of the system. The behaviors of a discrete system can thus be captured by a directed graph whose vertices represent the system states and whose edges represent the system transitions. Such a graph is called a *transition graph*.

Transition graph

A *transition graph* G consists of (1) a set Σ of vertices, (2) a subset $\sigma^I \subseteq \Sigma$ of the vertices, and (3) a binary edge relation $\rightarrow \subseteq \Sigma^2$ on the vertices. The vertices in Σ are called *states*, the vertices in σ^I are called *initial states*, and the edges in $\rightarrow$ are called *transitions*. We refer to the set Σ of states as the *state space* of G. Every subset of states from Σ is called a *region*; in particular, σ^I is the *initial region* of G. Every binary relation on Σ is called an *action*; in particular, $\rightarrow$ is the *transition action* of G.

Properties of transition graphs

The following properties of transition graphs are important. First, the transition action of every deadlock-free system is *serial*: for every state s, there is at least one successor state t with $s \rightarrow t$. Second, the mathematical analysis of a system is often simplified if the transition action is *finitely branching*: for every state s, there are at most finitely many successor states t with $s \rightarrow t$. Third, if the system may decide, in every update round, to leave the state unchanged, then the transition action is *reflexive*: every state s is its own successor state; that is, $s \rightarrow s$. Last, systems are amenable to analysis by graph algorithms if the state space is *finite*.

Serial, finitely branching, reflexive, finite transition graph

The transition graph $G = (\Sigma, \sigma^I, \rightarrow)$ is *serial* if (1) the initial region σ^I is nonempty and (2) the transition action $\rightarrow$ is serial. The transition graph G is *finitely branching* if (1) the initial region σ^I is finite and (2) the transition action $\rightarrow$ is finitely branching. The transition graph G is *reflexive* if the transition action $\rightarrow$ is reflexive. The transition graph G is *finite* if the state space Σ is finite.

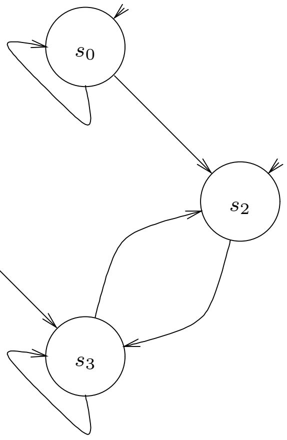


Figure 2.1: The transition graph $\hat{G}$

Remark 2.1 [Finite implies finite branching] Every finite transition graph is finitely branching. ■

Trajectories of transition graphs

The execution of a discrete system follows a path in the corresponding transition graph. Such a path starts in an initial state and proceeds through successive transitions. We are interested only in the states that are encountered within a finite number of transitions. The resulting finite paths are called *initialized trajectories*.

TRAJECTORY OF TRANSITION GRAPH

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. A *trajectory* of G is a nonempty word $\overline{s}_{1..m}$ over the alphabet Σ of states such that $s_i \rightarrow s_{i+1}$ for all $1 \leq i < m$. The first state s_1 is the *source*, the last state s_m is the *sink*, and the number m of states is the *length* of the trajectory $\overline{s}_{1..m}$. The trajectory $\overline{s}_{1..m}$ is an *initialized trajectory* of G if the source s_1 is an initial state of G. The set of initialized trajectories of G, denoted L_G, is called the *state language* of the transition graph G.

Remark 2.2 [Seriality implies trajectories of arbitrary length] Let G be a serial transition graph, and let s be a state of G. For every positive integer i, there is at least one trajectory of G with source s and length i. In particular, for every positive integer i, there is at least one initialized trajectory of G with length i. It follows that for serial transition graphs G, the state language L_G is infinite. ■

Example 2.1 [Transition graph] Figure 2.1 shows a finite transition graph with four states (s_0, s_1, s_2, and s_3). The two states s_0 and s_2 are initial, as is indicated by the short arrows without source state. The transition graph $\hat{G}$ has a total of six transitions. Since every state has at least one outgoing transition, $\hat{G}$ is serial. The infinite state language $L_{\hat{G}}$ includes the following four initialized trajectories:

$$s_0$$
$$s_0 s_0 s_0 s_0$$
$$s_0 s_0 s_2 s_3$$
$$s_2 s_3 s_2 s_3 s_3 s_3 s_2$$

The state language $L_{\hat{G}}$ is the regular set $s_0^+ \cup (s_0^* s_2 (s_3^+ s_2)^* s_3^*)$. ■

Remark 2.3 [Languages defined by transition graphs] For a transition graph G with state space Σ, the state language L_G is a language over the alphabet Σ. We say that G *defines* the language L_G. If G is finite, then L_G is a regular language. But not every subset of Σ^* is definable by a transition graph with state space Σ, and not every regular language is definable by a finite transition graph. This is shown in the following exercise. ■

Exercise 2.1 {`T2`} [Languages defined by transition graphs] Let A be an alphabet, and let $L \subseteq A^*$ be a language over A. (a) Prove that the language L is definable by a transition graph iff L is prefix-closed and fusion-closed. (Fusion closure captures the fact that the system state determines the possible future behaviors of the system.) (b) Prove that the language L is definable by a transition graph with the initial region A iff L is prefix-closed, fusion-closed, and suffix-closed. (c) Prove that the language L is definable by a reflexive transition graph iff L is prefix-closed, fusion-closed, and stutter-closed. ■

2.1.2 From Reactive Modules to Transition Graphs

We associate with every reactive module a serial transition graph that captures the behaviors of the module.

The states of a module

The *state* of a module between two rounds is determined by the values of all module variables. This is because the possible outcomes of the next and all future update rounds are determined solely by the current values of the module variables, and do not depend on any previous values.

State space of a module

Let P be a reactive module. A *state* of P is a valuation for the set X_P of module variables. We write Σ_P for the set of states of P.

Remark 2.4 [Existence of states] Every module has at least one state. The *empty module*, with the empty set of module variables, has exactly one state. ■

The state s of a module is *initial* if after the initial round, all module variables may have the values indicated by s. Consider a variable x. If x is external, then s can map x to any value of the appropriate type. If x is controlled by an atom U, then all variables in $\mathsf{await}X_U$ are initialized before x. In this case, the initial value $s(x)$ depends on the initial values of the awaited variables of U. The dependence is specified by the initial command init_U, which defines a relation between the valuations for the primed awaited variables $\mathsf{await}X'_U$ and the valuations for the primed controlled variables $\mathsf{ctr}X'_U$. In the following, if s is a valuation for a set X of unprimed variables, we write $\mathit{prime}(s)$ for the valuation for the set X' of corresponding primed variables such that $\mathit{prime}(s)(x') = s(x)$ for all variables $x \in X$.

Initial region of a module

Let P be a reactive module, let s be a state of P, and let $s' = \mathit{prime}(s)$. The state s is an *initial state* of P if for every atom U of P,

$$(s'[\mathsf{await}X'_U], s'[\mathsf{ctr}X'_U]) \in [\![\mathsf{init}_U]\!].$$

We write σ^I_P for the set of initial states of P.

Example 2.2 [Mutual exclusion] Recall Peterson's solution to the asynchronous mutual-exclusion problem from Figure 1.23. The module *Pete* has $3 \times 2 \times 3 \times 2 = 36$ states. Four of the states —s_1, s_2, s_3, and s_4— are initial:

$$\begin{array}{l}
s_1(pc_1) = outC,\ s_1(x_1) = true,\ s_1(pc_2) = outC,\ s_1(x_2) = true;\\
s_2(pc_1) = outC,\ s_2(x_1) = true,\ s_2(pc_2) = outC,\ s_2(x_2) = false;\\
s_3(pc_1) = outC,\ s_3(x_1) = false,\ s_3(pc_2) = outC,\ s_3(x_2) = true;\\
s_4(pc_1) = outC,\ s_4(x_1) = false,\ s_4(pc_2) = outC,\ s_4(x_2) = false.
\end{array}$$

It follows that the initial value of pc_1 and pc_2 is $outC$, and the initial values of x_1 and x_2 are unspecified. ■

Lemma 2.1 [Existence of initial states] *Every module has an initial state.*

Proof. Consider a module P. We prove the stronger claim that for every valuation s_e for the external variables of P, there is an initial state s of P such that $s[\mathsf{extl}X_P] = s_e$. Consider a valuation s_e for $\mathsf{extl}X_P$ and an execution order $U_1, \ldots, U_n$ for the atoms of P. We construct a sequence $s_0, s_1, \ldots, s_n$ of valuations for X_P as follows (let $s'_i = \mathit{prime}(s_i)$ for all $0 \le i \le n$): first, choose s_0 so that $s_0[\mathsf{extl}X_P] = s_e$; then, for all $1 \le i \le n$ and $Y_i = X_P \backslash \mathsf{ctr}X_{U_i}$, let $s_i[Y_i] = s_{i-1}[Y_i]$ and choose $s_i[\mathsf{ctr}X_{U_i}]$ so that $(s'_{i-1}[\mathsf{await}X'_{U_i}], s'_i[\mathsf{ctr}X'_{U_i}]) \in [\![\mathsf{init}_{U_i}]\!]$. At each step, at least one choice is possible because the binary relation

$[\![\mathsf{init}_{U_i}]\!]$ is serial. The construction ensures that for all module variables x of P, if x is an external variable, then $s_n(x) = s_e(x)$, and if x is a controlled variable of the atom U_i, then $s_n(x) = s_i(x)$. It follows that s_n is an initial state of P with $s_n[\mathsf{extl}X_P] = s_e$. ■

The transitions of a module

Consider two states s and t of a module. If the state s indicates the current values of the module variables at the beginning of an update round, and the state t indicates possible next values of the module variables at the end of the update round, then the state pair (s, t) is a *transition* of the module. For a formal definition of transitions, consider a variable x. If x is external, then t can map x to any value of the appropriate type. If x is controlled by an atom U, then all variables in $\mathsf{await}X_U$ are updated before x. In this case, the next value $t(x)$ depends on the current values of the read variables of U and on the next values of the awaited variables. The dependence is specified by the update command update_U, which defines a relation between the valuations for the unprimed read variables $\mathsf{read}X_U$ and the primed awaited variables $\mathsf{await}X'_U$ on one hand, and the valuations for the primed controlled variables $\mathsf{ctr}X'_U$ on the other hand.

Transition action of a module

Let P be a reactive module, let s and t be two states of P, and let $t' = \mathit{prime}(t)$. The state pair (s, t) is a *transition* of P if for every atom U of P,

$$(s[\mathsf{read}X_U] \cup t'[\mathsf{await}X'_U], t'[\mathsf{ctr}X'_U]) \in [\![\mathsf{update}_U]\!].$$

We write $\rightarrow_P$ for the set of transitions of P.

Example 2.3 [Mutual exclusion] Consider the state s_1 of the module *Pete* from Example 2.2. There are four transitions —(s_1, s_1), (s_1, s_5), (s_1, s_6), and (s_1, s_7)— whose first component is the initial state s_1:

$$\begin{aligned}&s_5(pc_1) = reqC,\ s_5(x_1) = true,\ s_5(pc_2) = outC,\ s_5(x_2) = true;\\ &s_6(pc_1) = outC,\ s_6(x_1) = true,\ s_6(pc_2) = reqC,\ s_6(x_2) = false;\\ &s_7(pc_1) = reqC,\ s_7(x_1) = true,\ s_7(pc_2) = reqC,\ s_7(x_2) = false.\end{aligned}$$

The transition (s_1, s_1) corresponds to an update round in which both processes sleep; the transition (s_1, s_5) corresponds to an update round in which the first process proceeds and the second process sleeps; the transition (s_1, s_6) corresponds to an update round in which the first process sleeps and the second process proceeds; and the transition (s_1, s_7) corresponds to an update round in which both processes proceed. ■

Lemma 2.2 [Existence of transitions] *Let P be a module. For every state s of P, there is a state t of P such that $s \rightarrow_P t$.*

Exercise 2.2 {T1} [Proof of Lemma 2.2] Let P be a module. Prove that for every state s of P, and every valuation t_e for the external variables of P, there is a state t of P such that (1) $s \rightarrow_P t$ and (2) $t[\mathsf{extl}X_P] = t_e$. Lemma 2.2 follows. ■

The transition graph of a module

We can now collect together the state space, initial region, and transition action of a module, thus obtaining a transition graph.

> TRANSITION GRAPH OF A MODULE
>
> Given a reactive module P, the *transition graph underlying* P is $G_P = (\Sigma_P, \sigma^I_P, \rightarrow_P)$.

Terminology. From now on, we freely attribute derivatives of the transition graph G_P to the module P. For example, each trajectory of G_P is called a trajectory of P; the state language L_{G_P} is called the state language of P, and denoted L_P. ■

Example 2.4 [Mutual exclusion] Figure 2.2 shows the transition graph G_{Pete} for Peterson's mutual-exclusion protocol. The label $o1i0$ denotes the state s with $s(pc_1) = outC$, $s(x_1) = true$, $s(pc_2) = inC$, and $s(x_2) = false$, etc. Each state s has a reflexive transition of the form $s \rightarrow s$, and these transitions are omitted from the figure. Note that some states at the left and right borders of the figure are identical, so as to avoid a large number of crossing edges in the figure. It can be checked that the state sequence shown in Figure 1.24 is indeed an initialized trajectory of G_{Pete}. ■

Proposition 2.1 [Seriality of transition graphs that underlie modules] *For every module P, the transition graph G_P is serial.*

Proof. Proposition 2.1 follows from Lemmas 2.1 and 2.2. ■

Remark 2.5 [Transition graphs for finite, closed, deterministic, and passive modules] If P is a finite module, then G_P is a finite transition graph. If all external variables of P have finite types, then G_P is a finitely branching transition graph. In particular, for every closed module, the underlying transition graph is finitely branching, and for every closed deterministic module P, the underlying transition graph has branching degree 1: there is exactly one initial state, and for each state s, there is exactly one successor state t with $s \rightarrow_P t$. If P is a passive module, then the transition action $\rightarrow_P$ is reflexive. ■

The transition graph G_P captures only the behaviors of the module P, and not its interface structure. First, transition graphs do not distinguish between controlled and external variables. Hence there is no composition operation on

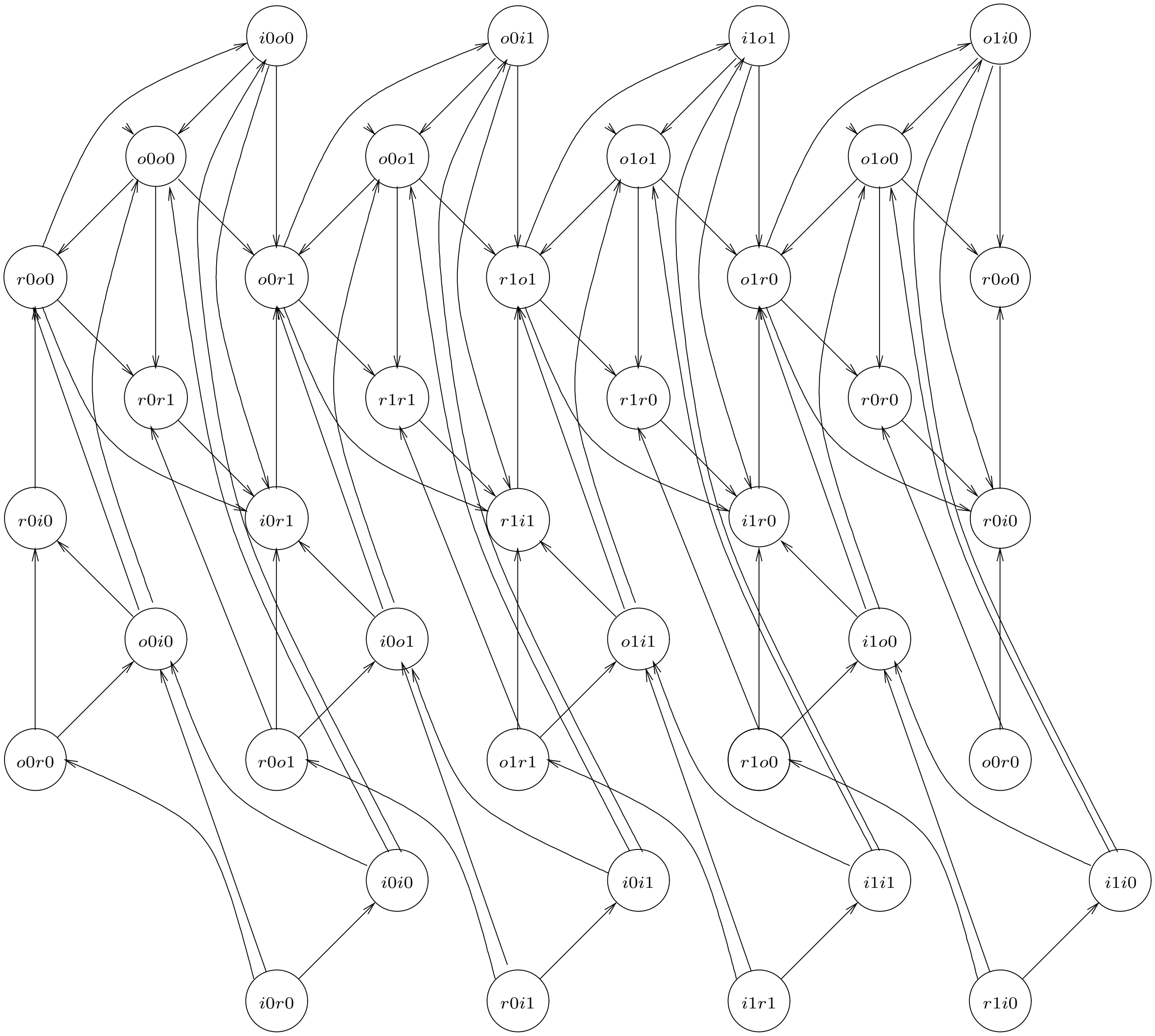


Figure 2.2: The transition graph G_{Pete}

transition graphs. Second, transition graphs do not distinguish between private and observable variables. Hence asynchronicity does not correspond to a property of transition graphs, and there is no hiding operation on transition graphs (the hiding of variables does not alter the transition graph of a module).

Exercise 2.3 {T2} [Transition graph of compound modules] Consider two compatible modules P and Q. (a) Assume that the two modules have no private variables ($\mathsf{priv}X_P = \mathsf{priv}X_Q = \emptyset$), and that the interface variables of one module are the external variables of the other module ($\mathsf{intf}X_P = \mathsf{extl}X_Q$ and $\mathsf{extl}X_P = \mathsf{intf}X_Q$). Then the two modules and the compound module have the same state space: $\Sigma_P = \Sigma_Q = \Sigma_{P\|Q}$. Prove that the transition action of the compound module $P\|Q$ is the intersection of the transition actions of the two component modules P and Q; that is, $s \rightarrow_{P\|Q} t$ iff $s \rightarrow_P t$ and $s \rightarrow_Q t$. What can you say about the initial states of the compound module $P\|Q$? (b) Assume that the two modules have no variables in common ($X_P \cap X_Q = \emptyset$). Then $\Sigma_{P\|Q} = \{s_1 \cup s_2 \mid s_1 \in \Sigma_P \text{ and } s_2 \in \Sigma_Q\}$. Prove that the transition action of the compound module is the cartesian product of the transition actions of the two component modules; that is, $(s_1 \cup s_2) \rightarrow_{P\|Q} (t_1 \cup t_2)$ iff $s_1 \rightarrow_P t_1$ and $s_2 \rightarrow_Q t_2$. What can you say about the initial states of $P\|Q$? (c) Now consider the general case. Consider two states s and t of the compound module $P\|Q$. Prove that (1) $s \in \sigma^I_{P\|Q} t$ iff $s[X_P] \in \sigma^I_P$ and $s[X_Q] \in \sigma^I_Q$, and (2) $s \rightarrow_{P\|Q} t$ iff $s[X_P] \rightarrow_P t[X_P]$ and $s[X_Q] \rightarrow_Q t[X_Q]$. ■

The following proposition asserts that the (initialized) trajectories of a compound module are determined by the (initialized) trajectories of the component modules. In particular, for two compatible modules P and Q, if the two modules have the same state space, then $L_{P\|Q} = L_P \cap L_Q$.

Proposition 2.2 [Trajectories of compound modules] *For every pair P and Q of compatible modules, a sequence $\overline{s}$ of states in $\Sigma_{P\|Q}$ is an (initialized) trajectory of the compound module $P\|Q$ iff $\overline{s}[X_P]$ is an (initialized) trajectory of P and $\overline{s}[X_Q]$ is an (initialized) trajectory of Q.*

Proof. Proposition 2.2 follows from part (c) of Exercise 2.3. ■

Exercise 2.4 {T1} [Least constraining environments] The module Q is a *least constraining environment* for the module P if (1) P and Q are compatible, (2) the compound module $P\|Q$ is closed, and (3) $G_{P\|Q} = G_P$. Prove that for every module P, if all external variables of P have finite types, then there exists a least constraining environment for P. Can a module have more than one least constraining environment? ■

An interpreter for reactive modules

Following the informal execution model of Chapter 1, we are now equipped to build an interpreter for reactive modules. The execution of a module for a finite number of rounds yields an initialized trajectory of the module. Therefore, if the interpreter receives as input the module P, it returns as output an initialized trajectory of P. Since P may have many initialized trajectories, the output of the interpreter is nondeterministic. Indeed, every initialized trajectory of P must be a possible output of the interpreter.

The interpreter, Algorithm 2.1, proceeds in three phases. The first phase computes an execution order for the atoms of P. The second phase simulates the initial round, by executing the initial commands of all atoms in the chosen execution order. The third phase simulates a finite number of update rounds, by iteratively executing the update commands of all atoms in the chosen execution order. Algorithm 2.1 uses the following notation. The function $\mathit{Execute}(\Gamma, s)$, shown below, computes the result of executing the guarded command Γ on the valuation s. If Γ is a guarded command from X to Y, then s must be a valuation for X, and the function *Execute* returns a valuation t for Y such that $(s, t) \in [\![\Gamma]\!]$:

```
**function** $\mathit{Execute}(\Gamma, s)$
  Assume $\Gamma$ is a guarded command from $X$ to $Y$;
  Choose a guarded assignment $\gamma$ of $\Gamma$ such that $s(p_\gamma) = \mathit{true}$;
  Let $t$ be the valuation for the empty set of variables;
  **foreach** $y$ **in** $Y$ **do** $t := t[y \mapsto s(e^x_\gamma)]$ **od**;
  **return** $t$.
```

If s is a valuation for a set X' of primed variables, we write $\mathit{unprime}(s)$ for the valuation for the set X of corresponding unprimed variables such that $\mathit{unprime}(s)(x) = s(x')$ for all variables $x \in X$.

2.1.3 The Reachability Problem

For a transition graph G that captures the behaviors of a system, we are interested only in the states of G that occur on initialized trajectories. These states are called *reachable*. By deleting the unreachable states from G, we obtain the reachable subgraph of G. The reachable subgraph can often be significantly smaller than the complete transition graph.

Algorithm 2.1 [Module Execution] (schema)

```
Input: a reactive module $P$.
Output: an initialized trajectory $\overline{s}_{1..m}$ of $P$.
Preparation.
  Topologically sort the atoms of $P$ with respect to the precedence
  relation $\prec\!\!\prec_P$, and store the result as $(U_1, \ldots, U_n)$;
Initial round.
  Choose an arbitrary valuation $s$ for $\mathsf{extl}X'_P$;
  for $j := 1$ to $n$ do $s := s \cup \mathit{Execute}(\mathsf{init}_{U_j}, s)$ od;
  $s_1 := \mathit{unprime}(s)$;
Update rounds.
  Choose an arbitrary positive integer $m$;
  for $i := 2$ to $m$ do
    Choose an arbitrary valuation $s$ for $\mathsf{extl}X'_P$;
    for $j := 1$ to $n$ do $s := s \cup \mathit{Execute}(\mathsf{update}_{U_j}, s_{i-1} \cup s)$ od;
    $s_i := \mathit{unprime}(s)$
    od.
```

REACHABILITY

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph, and let s be a state of G. The state s is *reachable in i transitions*, for a nonnegative integer i, if the transition graph G has an initialized trajectory with sink s and length $i + 1$. The state s is a *reachable state* of G if there is a nonnegative integer i such that s is reachable in i transitions. The transition graph G is *finitely reaching* if there is a nonnegative integer i such that every reachable state of G is reachable in at most i transitions. The *reachable region* of G is the set σ^R of reachable states of G. The *reachable subgraph* of G is the transition graph $G^R = (\sigma^R, \sigma^I, \rightarrow^R)$, where $\rightarrow^R = \rightarrow [\sigma^R]$ is the restriction of the transition action $\rightarrow$ to the reachable region σ^R. The transitions of G^R are called the *reachable transitions* of G.

Example 2.5 [Reachable subgraph] In the simple transition graph $\hat{G}$ from Example 2.1, the state s_1 is unreachable, and so is the transition from s_1 to s_3. The reachable subgraph of $\hat{G}$ is shown in Figure 2.3. ■

Example 2.6 [Mutual exclusion] Figure 2.4 shows the reachable subgraph of the transition graph G_{Pete} from Example 2.4 (reflexive transitions are suppressed). It has four initial states, 20 reachable states, and 64 reachable transitions. In other words, 16 of the states in Figure 2.2 are unreachable. ■

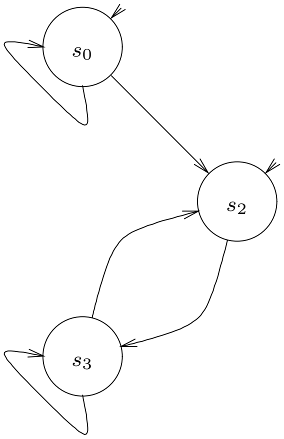


Figure 2.3: The reachable subgraph of $\hat{G}$

Remark 2.6 [Finite vs. finitely branching vs. finitely reaching] Every finite transition graph is finitely reaching. If a transition graph G is both finitely branching and finitely reaching, then the reachable subgraph G^R is finite. ■

The most important questions in computer-aided verification can be phrased as reachability questions. A *reachability question* asks if any state from a given region is reachable in a given transition graph. If the reachable subgraph is finite, then the reachability question can be solved using graph-traversal algorithms (see Section 2.3).

REACHABILITY PROBLEM

An instance (G, σ^T) of the *reachability problem* consists of (1) a transition graph G and (2) a region σ^T of G, which is called the *target region*. The answer to the reachability question (G, σ^T) is YES if a state in the target region σ^T is reachable, and otherwise NO. A *witness* for a YES-instance (G, σ^T) of the reachability problem is an initialized trajectory of G whose sink is in σ^T.

Remark 2.7 [Emptiness problem for finite automata] The reachability problem is equivalent to the *one-letter emptiness problem for finite automata*, which asks if a given finite automaton with a singleton input alphabet accepts any input word. To see this, view the target region as an accepting region. ■

2.2 Invariants

To an observer, only the values of the interface and external variables of a module are visible. We therefore specify properties of module states by constraints on

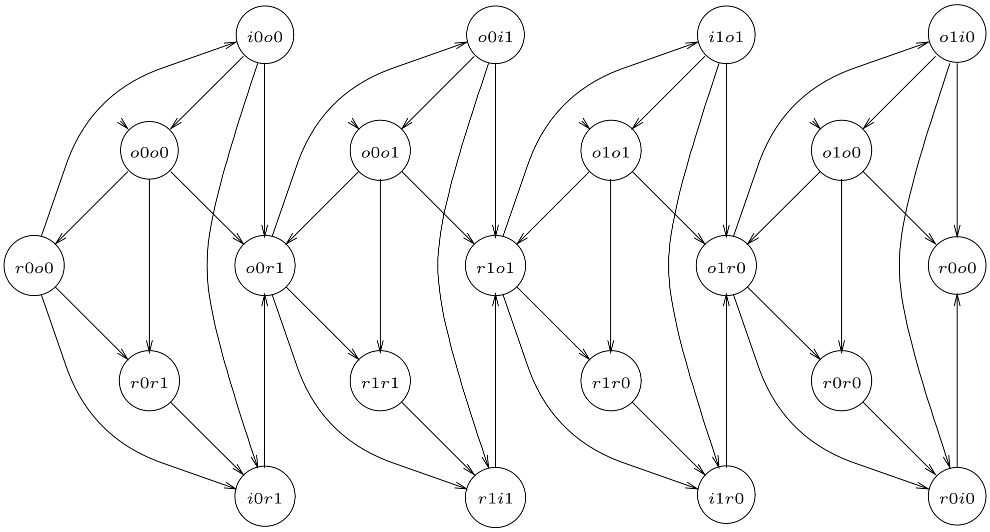


Figure 2.4: The reachable subgraph of G_{Pete}

the values of the observable variables. For example, if x is an observable integer variable, then the constraint $x > 5$ is satisfied by the states that map x to a value greater than 5, and the constraint is violated by the states that map x to a value less than or equal to 5. A constraint r on the observable variables of a module P is an *invariant* of P if all reachable states of P satisfy r. If r is an invariant of P, then it cannot happen that within a finite number of rounds, the module P moves into a state that violates r. Many important requirements on the behavior of reactive modules can be expressed as invariants.

Example 2.7 [Mutual exclusion] Peterson's protocol meets the mutual-exclusion requirement iff the constraint

$$r^{mutex}: \quad \neg(pc_1 = inC \wedge pc_2 = inC)$$

is an invariant of the module *Pete*. This constraint asserts that at most one of the two processes is inside its critical section. Note that the status of each process can be observed, because both pc_1 and pc_2 are interface variables. It is evident from inspecting the reachable subgraph of G_{Pete} (see Example 2.6) that every reachable state of *Pete* satisfies the constraint r^{mutex}. It follows that r^{mutex} is an invariant of the module *Pete*. ■

2.2.1 The Invariant-Verification Problem

If P is a reactive module, then a constraint on the values of module variables is called a *state predicate* for P. We do not allow the occurrence of event variables in state predicates, because the value of an event variable in any given state is immaterial. A state predicate that constrains only the values of observable variables is called an *observation predicate.* In particular, observation predicates cannot constrain the values of private variables.

State predicate

Let P be a reactive module. A *state predicate* for P is a boolean expression over the set $X_P \backslash \mathsf{event} X_P$ of module variables that are not event variables. The state predicate q is an *observation predicate* if all free variables of q are observable variables of P. The observation predicate q is an *interface predicate* if all free variables of q are interface variables, and q is an *external predicate* if all free variables of q are external variables. Given a state predicate q for P, we write $[\![q]\!]_P$ for the set of states of P that satisfy q.

Remark 2.8 [Regions defined by state predicates] Let P be a module, and let q and r be two state predicates for P. We say that the state predicate q *defines* the region $[\![q]\!]_P = \{s \in \Sigma_P \mid s \models q\}$ of P. The regions of P that are definable by state predicates form a boolean algebra:

$[\![\mathit{true}]\!]_P = \Sigma_P$ and $[\![\mathit{false}]\!]_P = \emptyset$;
$[\![q \wedge r]\!]_P = [\![q]\!]_P \cap [\![r]\!]_P$ and $[\![q \vee r]\!]_P = [\![q]\!]_P \cup [\![r]\!]_P$;
$[\![\neg q]\!]_P = \Sigma_P \backslash [\![q]\!]_P$.

■

Invariant

Let P be a reactive module, and let r be an observation predicate for P. The predicate r is an *invariant* of P if all reachable states of P satisfy r.

In other words, given a module P with the reachable region σ^R, the observation predicate r is an invariant of P iff $\sigma^R \subseteq [\![r]\!]_P$.

Remark 2.9 [Monotonicity of invariants] Let P be a module, and let q and r be two observation predicates for P. (1) The observation predicate *true* is an invariant of P. If q is an invariant of P, and $q \rightarrow r$ is valid, then r is also an invariant of P. It follows that every valid observation predicate for P is an invariant of P. (2) If both q and r are invariants of P, then $q \wedge r$ is also an invariant of P. ■

Invariant-verification problem

An instance (P, r) of the *invariant-verification problem* consists of (1) a reactive module P and (2) an observation predicate r for P. The answer to the invariant-verification question (P, r) is Yes if r is an invariant of P, and otherwise No. An *error trajectory* for a No-instance (P, r) of the invariant-verification problem is an initialized trajectory of P whose sink violates r.

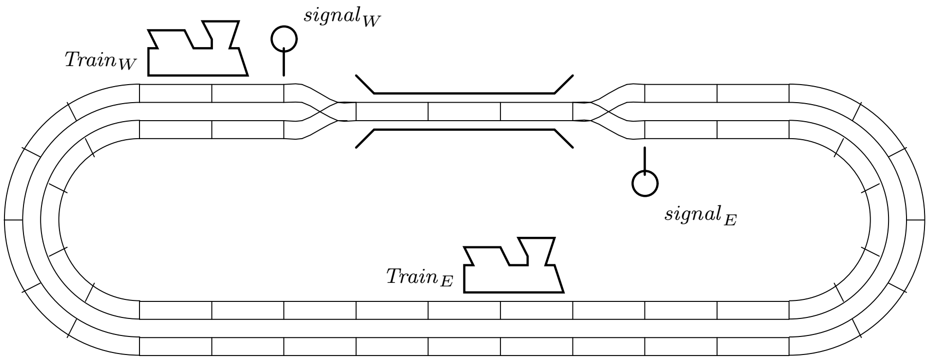


Figure 2.5: Railroad example

If the observation predicate r is not an invariant of the module P, then error trajectories present evidence to the designer of P as to how the module can end up in a state that violates r. Error trajectories thus provide valuable debugging information on top of the answer No to an invariant-verification question.

Example 2.8 [Railroad control] Figure 2.5 shows two circular railroad tracks, one for trains that travel clockwise, and the other for trains that travel counterclockwise. At one place in the circle, there is a bridge which is not wide enough to accommodate both tracks. The two tracks merge on the bridge, and for controlling the access to the bridge, there is a signal at either entrance. If the signal at the western entrance is green, then a train coming from the west may enter the bridge; if the signal is red, the train must wait. The signal at the eastern entrance to the bridge controls trains coming from the east in the same fashion.

A train is modeled by the asynchronous and passive module *Train* shown in Figure 2.6. When the train approaches the bridge, it sends an *arrive* event to the railroad controller and checks the signal at the entrance to the bridge ($pc = wait$). When the signal is red, the train stops and keeps checking the signal. When the signal is green, the train proceeds onto the bridge ($pc = bridge$). When the train exits from the bridge, it sends a *leave* event to the controller and travels around the circular track ($pc = away$). The traveling around the circular track, the checking of the signal, and the traveling time across the bridge each take an unknown number of rounds. There are two trains, one traveling clockwise and the other traveling counterclockwise. The first train, which arrives at the western entrance of the bridge, is represented by the module

module $Train_W$ **is**
  $Train[pc, arrive, signal, leave := pc_W, arrive_W, signal_W, leave_W]$,

```
module $Train$ is
  interface $pc\colon \{away, wait, bridge\};\; arrive, leave : \mathbb{E}$
  external $signal\colon \{green, red\}$
  lazy atom controls $arrive$ reads $pc$
    update
      $[\!]\; pc = away \to arrive!$
  lazy atom controls $leave$ reads $pc$
    update
      $[\!]\; pc = bridge \to leave!$
  lazy atom controls $pc$ reads $pc, arrive, leave, signal$ awaits $arrive, leave$
    init
      $[\!]\; true \to pc' := away$
    update
      $[\!]\; pc = away \wedge arrive?$           $\to pc' := wait$
      $[\!]\; pc = wait \wedge signal = green$  $\to pc' := bridge$
      $[\!]\; pc = bridge \wedge leave?$         $\to pc' := away$
```

Figure 2.6: Train

and the second train, which arrives at the eastern entrance, is represented by the module

```
module $Train_E$ is
  $Train[pc, arrive, signal, leave := pc_E, arrive_E, signal_E, leave_E]$.
```

We are asked to design a passive controller module *Controller* that prevents collisions between the two trains by ensuring the *train-safety* requirement that in all rounds, at most one train is on the bridge. The module *Controller* enforces the train-safety requirement iff the observation predicate

$$r^{safe}\colon \quad \neg(pc_W = bridge \wedge pc_E = bridge)$$

is an invariant of the compound module

```
module $RailroadSystem$ is
  hide $arrive_W, arrive_E, leave_W, leave_E$ in
    $\|\; Train_W$
    $\|\; Train_E$
    $\|\; Controller$.
```

The external variables of the module *Controller* should be $arrive_W$, $arrive_E$, $leave_W$, and $leave_E$.

```
module Controller1 is
  interface $signal_W, signal_E : \{green, red\}$
  external $arrive_W, arrive_E, leave_W, leave_E : \mathbb{E}$
  passive atom controls $signal_W, signal_E$
    reads $signal_W, signal_E, arrive_W, arrive_E, leave_W, leave_E$
    awaits $arrive_W, arrive_E, leave_W, leave_E$
    init
      $[\!] \; true \rightarrow signal'_W := green;\; signal'_E := green$
    update
      $[\!] \; arrive_W? \rightarrow signal'_E := red$
      $[\!] \; arrive_E? \rightarrow signal'_W := red$
      $[\!] \; leave_W? \;\; \rightarrow signal'_E := green$
      $[\!] \; leave_E? \;\;\; \rightarrow signal'_W := green$
```

Figure 2.7: First attempt at railroad control

pc_W	*away*	*wait*	*bridge*	*away*	*away*	*wait*	*bridge*
$arrive_W$		◇				◇	
$leave_W$				◇			
pc_E	*away*	*wait*	*wait*	*wait*	*bridge*	*bridge*	*bridge*
$arrive_W$		◇					
$leave_W$							
$signal_W$	*green*	*green*	*green*	*green*	*green*	*green*	*green*
$signal_E$	*green*	*red*	*red*	*green*	*green*	*red*	*red*

Figure 2.8: An error trajectory that violates train safety

```
module $Controller2$ is
  private $near_W, near_E : \mathbb{B}$
  interface $signal_W, signal_E : \{green, red\}$
  external $arrive_W, arrive_E, leave_W, leave_E : \mathbb{E}$
  passive atom controls $near_W$
    reads $near_W, arrive_W, leave_W$
    awaits $arrive_W, leave_W$
    init
      $[\!] \; true \to near'_W := false$
    update
      $[\!] \; arrive_W? \to near'_W := true$
      $[\!] \; leave_W? \;\; \to near'_W := false$
  passive atom controls $near_E$
    reads $near_E, arrive_E, leave_E$
    awaits $arrive_E, leave_E$
    init
      $[\!] \; true \to near'_E := false$
    update
      $[\!] \; arrive_E? \to near'_E := true$
      $[\!] \; leave_E? \;\; \to near'_E := false$
  lazy atom controls $signal_W, signal_E$
    reads $near_W, near_E, signal_W, signal_E$
    init
      $[\!] \; true \to signal'_W := red; \; signal'_E := red$
    update
      $[\!] \; near_W \wedge signal_E = red \to signal'_W := green$
      $[\!] \; near_E \wedge signal_W = red \to signal'_E := green$
      $[\!] \; \neg near_W$                    $\to signal'_W := red$
      $[\!] \; \neg near_E$                    $\to signal'_E := red$
```

Figure 2.9: Second attempt at railroad control

Figure 2.7 shows a first attempt at designing the railroad controller. Initially, both signals are green. A signal turns red whenever a train approaches the opposite entrance to the bridge, and it turns back to green whenever that train exits from the bridge. If both trains approach the bridge in the same round, then only one of the two signals turns red (the one that turns red is chosen nondeterministically), and the other train is admitted to the bridge. Unfortunately, the resulting railroad system does not have the invariant r^{safe}. This is evidenced by the error trajectory shown in Figure 2.8, which leads to a state with both trains on the bridge. If both trains approach the bridge simultaneously, then one is admitted to the bridge. When that train exits from the bridge, the other train is admitted to the bridge. At that point both signals are green. So when the first train returns while the second train is still on the bridge, the two trains will collide. It can be checked that the state sequence shown in Figure 2.8 is in fact the shortest initialized trajectory whose sink violates r^{safe}. ■

Exercise 2.5 {P2} [Railroad control] Figure 2.9 shows a second attempt at designing a railroad controller for Example 2.8. How many states does the module $\mathit{Train}_W \parallel \mathit{Train}_E \parallel \mathit{Controller2}$ have? How many of these states are reachable? Is there a reachable state with both trains on the bridge? To answer the latter two questions, draw the reachable subgraph of the transition graph. ■

2.2.2 From Invariant Verification to Reachability

Given a reactive module P, the execution of P generates a single initialized trajectory of P. By contrast, for solving an invariant-verification question about P, we must systematically explore all initialized trajectories of P. This can be done by solving a reachability question about the underlying transition graph G_P.

Proposition 2.3 [Reduction from invariant verification to reachability] *The answer to an instance (P, r) of the invariant-verification problem is* Yes *iff the answer to the instance $(G_P, [\![\neg r]\!]_P)$ of the reachability problem is* No. *Furthermore, if (P, r) is a* No*-instance of the invariant-verification problem, then every witness for the reachability question $(G_P, [\![\neg r]\!]_P)$ is an error trajectory for the invariant-verification question (P, r).*

It follows that we can answer the question if an observation predicate r is an invariant of a reactive module P if we can solve the reachability question $(G_P, [\![\neg r]\!]_P)$. Furthermore, if r is not an invariant of P, then we can provide an error trajectory by generating a witness for the Yes-instance $(G_P, [\![\neg r]\!]_P)$ of the reachability problem. We will discuss several algorithms for solving and generating witnesses for reachability questions. Yet it is important to clearly distinguish between the two problems: the input to the invariant-verification problem is a module and an observation predicate; the input to the reachability problem is a transition graph and a region. While the former can be reduced to the latter, this reduction typically requires exponential amount of work: indeed, as we shall see, invariant verification is inherently harder —for finite state spaces, by an exponential factor— than reachability.

Remark 2.10 [State predicates as invariants] Our formulation of the invariant-verification problem allows us to check whether r is an invariant of a module P when r refers only to the observable variables of P. The prohibition of requirements that refer to private variables is a good specification discipline, which can be exploited by reduction techniques such as minimization (see Chapter 5). However, it should be evident that one can check whether all reachable states of a module P satisfy a state predicate r by solving the reachability question $(G_P, [\![\neg r]\!]_P)$ even when r refers to private variables of P. ■

Exercise 2.6 {`T1`} [Transition invariants] Invariants cannot be used to directly specify module requirements that involve events, because observation predicates are interpreted over individual states. It is possible, however, to generalize invariants from observation predicates to transition predicates, which are interpreted over individual transitions and therefore can refer to the presence and absence of events. Let P be a reactive module. A *transition predicate* for P is a boolean expression over the set $X_P \cup X'_P$ of unprimed and primed module variables, with the restriction that event variables can occur only in subexpressions of the form $x?$ (which stands for $x' \neq x$). A pair (s, t) of states of P *satisfies* the transition predicate r' if $(s \cup t) \models r'$. It follows that every transition predicate r' *defines* an action $[\![r']\!]_P \subseteq \Sigma_P^2$, which contains all pairs of states of P that satisfy r'. The transition predicate r' is *observable* if no private variables (unprimed or primed) of P occur in r'. The observable transition predicate r' is a *transition invariant* of P if all reachable transitions of P satisfy r'; that is, if $\rightarrow_P^R \subseteq [\![r']\!]_P$. An instance (P, r') of the *transition-invariant verification problem* consists of (1) a reactive module P and (2) an observable transition predicate r' for P. The answer to the transition-invariant question (P, r') is Yes if r' is a transition invariant of P, and otherwise No.

Define a notion of error trajectories for the transition-invariant problem and reduce the problem, including the generation of error trajectories, to the following transition-reachability problem. An instance (G, α^T) of the *transition-reachability problem* consists of (1) a transition graph G and (2) an action α^T of G, which is called the *target action.* The answer to the transition-reachability question (G, α^T) is Yes if a transition in the target action α^T is reachable, and otherwise No. The answer Yes can be *witnessed* by an initialized trajectory of G of the form $\overline{s}_{1..m}$ with $m \geq 2$ and $(s_{m-1}, s_m) \in \alpha^T$. ■

```
module MonMonitor is
  interface $alert\colon \{0,1\}$
  external $x\colon \mathbb{N}$
  passive atom controls $alert$ reads $x$ awaits $x$
    init
      $\talloblong$ $true \rightarrow alert' := 0$
    update
      $\talloblong$ $x' \geq x \rightarrow alert' := 0$
      $\talloblong$ $x' < x \rightarrow alert' := 1$
```

Figure 2.10: Monitoring monotonicity

2.2.3 Monitors

Invariants can distinguish between two trajectories only if one of the trajectories contains a state that does not occur on the other trajectory. Hence there are requirements on the behavior of a reactive module P that cannot be expressed as invariants of P. However, many such requirements can be expressed as invariants of the compound module $P \| M$, for a monitor M of P. The module M is a *monitor* of P if (1) M is compatible with P and (2) $\mathsf{intf}X_M \cap \mathsf{extl}X_P = \emptyset$. If M is a monitor of P, then in every round, M may record the values of the observable variables of P, but M cannot control any external variables of P. Thus the monitor M can watch but not interfere with the behavior of P. In particular, the monitor M may check if P meets a requirement, and it may signal every violation of the requirement by sounding an observable alarm. The module P then meets the given requirement iff the compound module $P \| M$ has the invariant that no alarm is sounded by the monitor M.

Consider, for example, a module P with an interface variable x that ranges over the nonnegative integers. Assume that, during every update round, it is ok for P to increase the value of x, or to leave it unchanged, but it is not ok for P to decrease the value of x. This *monotonicity* requirement cannot be expressed as an invariant of P. However, we can design a monitor *MonMonitor* of P so that the monotonicity requirement can be expressed as an invariant of the compound module $P \,\|\, \mathit{MonMonitor}$. The monitor *MonMonitor*, shown in Figure 2.10, has but one variable, *alert*, which is an interface variable and ranges over the set $\{0,1\}$ of two alertness levels. The monitor *MonMonitor* watches for changes in the value of x. In every update round, if the value of x does not decrease, then the new value of *alert* is 0, which indicates that there is no reason for concern; if the value of x decreases, then the new value of *alert* is 1, which sounds an alarm. The module P then meets the monotonicity requirement iff the observation predicate $alert \neq 1$ is an invariant of the module $P \,\|\, \mathit{MonMonitor}$.

```
module $AltMonitor$ is
  interface $alert \colon \{0,1,2\}$
  external $x \colon \mathbb{N}$
  atom controls $alert$ reads $alert, x$ awaits $x$
    init
      $\| \; true \rightarrow alert' := 0$
    update
      $\| \; x' \geq x \qquad\qquad\quad \rightarrow alert' := 0$
      $\| \; alert = 0 \wedge x' < x \rightarrow alert' := 1$
      $\| \; alert = 1 \wedge x' < x \rightarrow alert' := 2$
```

Figure 2.11: Monitoring alternation

For a slightly more involved variation of the previous example, assume that it is ok for P to decrease the value of x occasionally, but it is not ok to decrease the value of x twice in a row, during two consecutive update rounds. Figure 2.11 shows a monitor of P that checks this *alternation* requirement. The interface variable *alert* of the monitor *AltMonitor* ranges over the set $\{0, 1, 2\}$ of three alertness levels. If the value of x does not decrease during an update round, then $alert = 0$, which indicates that there is no immediate danger of P violating the alternation requirement; if $alert = 0$ and the value of x decreases during an update round, then $alert = 1$, which indicates that there is an immediate danger of P violating the alternation requirement; if $alert = 1$ and the value of x decreases during an update round, then $alert = 2$, which indicates that P has violated the alternation requirement. The module P then meets the alternation requirement iff the observation predicate $alert \neq 2$ is an invariant of the module $P \parallel AltMonitor$.

Example 2.9 [Railroad control] This is a continuation of Example 2.8. Figure 2.9 presents an asynchronous railroad controller that enforces the train-safety requirement. Yet the module *Controller2* is not a satisfactory railroad controller, because it may keep a train waiting at a red signal while the other train is allowed to cross the bridge repeatedly. In particular, the resulting railroad system does not meet the *equal-opportunity* requirement that, while a train is waiting at a red signal, it is not possible that the signal at the opposite entrance to the bridge turns from green to red and back to green. Since the equal-opportunity requirement is violated by trajectories, and not by individual states, we need to employ monitors. The module

```
module $EqOppMonitor_W$ is
  $EqOppMonitor[alert, pc, signal_1, signal_2 := alert_W, pc_W, signal_W, signal_E]$
```

monitors the equal-opportunity requirement for the train that travels clockwise, where *EqOppMonitor* is shown in Figure 2.12. The monitor has four levels of

```
module $EqOppMonitor$ is
  interface $alert\colon \{0, 1, 2, 3\}$
  external $pc\colon \{away, wait, bridge\};\ signal_1, signal_2\colon \{green, red\}$
  passive atom controls $alert$ reads $alert, pc, signal_1, signal_2$
    init
      $[\!]\ true \to alert' := 0$
    update
      $[\!]\ alert = 0 \wedge pc = wait \wedge signal_1 = red \wedge signal_2 = green \to alert' := 1$
      $[\!]\ alert = 1 \wedge signal_1 = green \to alert' := 0$
      $[\!]\ alert = 1 \wedge signal_1 = red \wedge signal_2 = red \to alert' := 2$
      $[\!]\ alert = 2 \wedge signal_1 = green \to alert' := 0$
      $[\!]\ alert = 2 \wedge signal_1 = red \wedge signal_2 = green \to alert' := 3$
```

Figure 2.12: Monitoring equal opportunity

pc_W	*away*	*wait*	*wait*	*wait*	*wait*	*wait*	*wait*	*wait*	*wait*
$arrive_W$		◇							
$leave_W$									
pc_E	*away*	*away*	*wait*	*wait*	*bridge*	*away*	*wait*	*wait*	*bridge*
$arrive_W$			◇				◇		
$leave_W$						◇			
$signal_W$	*red*	*red*	*red*	*red*	*red*	*red*	*red*	*red*	*red*
$signal_E$	*red*	*red*	*red*	*green*	*green*	*red*	*red*	*green*	*green*
$alert_W$	0	0	0	0	1	1	2	2	3
$alert_E$	0	0	0	0	0	0	0	0	0

Figure 2.13: An error trajectory that violates equal opportunity

alertness. The alertness level is 0 as long as the train is not waiting at a red signal while the other signal is green, in which case the alertness level rises to 1. The alertness level rises to 2 when the other signal turns red, and to 3, when the other signal turns green again, while the train is still waiting at a red signal. An alertness level of 3 sounds an alarm that indicates a violation of the equal-opportunity requirement for the train that travels clockwise. The equal-opportunity requirement for the train that travels counterclockwise is monitored by the module

module $\mathit{EqOppMonitor}_E$ **is**
$\mathit{EqOppMonitor}[\mathit{alert}, \mathit{pc}, \mathit{signal}_1, \mathit{signal}_2 := \mathit{alert}_E, \mathit{pc}_E, \mathit{signal}_E, \mathit{signal}_W]$

in the same manner. The module *RailroadSystem* then meets the equal-opportunity requirement iff the observation predicate

$$\neg(\mathit{alert}_W = 3 \vee \mathit{alert}_E = 3)$$

is an invariant of the compound module

$$\mathit{RailroadSystem} \parallel \mathit{EqOppMonitor}_W \parallel \mathit{EqOppMonitor}_E.$$

The error trajectory of Figure 2.13 shows that this is not the case. ■

Exercise 2.7 {P3} [Mutual exclusion] The *first-request-first-in* requirement for mutual-exclusion protocols asserts that the first process to request admission to the critical section (meaning: $\mathit{pc} = \mathit{reqC}$) is the first process with an opportunity to enter the critical section (meaning: the guard is true for some guarded command that updates *pc* from *reqC* to *inC*). (If both processes request to enter simultaneously, no order is specified.) (a) Write a monitor that checks the first-request-first-in requirement for mutual-exclusion protocols, and reduce the question of whether a mutual-exclusion protocol meets the first-request-first-in requirement to a invariant-verification question. (b) Does Peterson's mutual-exclusion protocol (Figure 1.23) meet the first-request-first-in requirement? What about the synchronous mutual-exclusion protocol from Figure 1.22? (c) How does the first-request-first-in requirement relate to the accessibility requirement specified in Chapter 1? (Does one imply the other?)

The *equal-opportunity* requirement for mutual-exclusion protocols asserts that, while a process is requesting to enter its critical section, it is not possible that the other process enters its critical section more than once. Equal opportunity, then, is a weaker requirement than first-request-first-in. Repeat parts (a)–(c) for the equal-opportunity requirement. ■

2.3 Graph Traversal

The reachability problem, and therefore the invariant-verification problem, can be solved by classical graph-search algorithms.

2.3.1 Reachability Checking

Graph-search algorithms traverse a graph one edge at a time, moving from a given vertex to its successor (or predecessor) vertices. It is useful to view these algorithms in terms of the following notions.

PREDECESSOR AND SUCCESSOR REGIONS

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph, and let s be a state of G. The state t of G is a *predecessor* of s if $t \rightarrow s$, and t is a *successor* of s if $s \rightarrow t$. The *predecessor region* $\mathit{pre}_G(s)$ of s is the set of predecessors of s, and the *successor region* $\mathit{post}_G(s)$ of s is the set of successors of s. We write $\mathit{pre}^*_G(s)$ for the so-called *source region* $(\cup\, i \in \mathbb{N} \mid \mathit{pre}^i_G(s))$ of s, and $\mathit{post}^*_G(s)$ for the *sink region* $(\cup\, i \in \mathbb{N} \mid \mathit{post}^i_G(s))$.

In other words, given a transition graph G and a state s of G, the source region $\mathit{pre}^*_G(s)$ contains the sources of all trajectories of G with sink s, and the sink region $\mathit{post}^*_G(s)$ contains the sinks of all trajectories of G with source s.

Terminology. The functions pre_G, post_G, pre^*_G, and post^*_G are extended to regions in the natural way: for a region σ of the transition graph G, let $\mathit{pre}_G(\sigma) = (\cup\, s \in \sigma \mid \mathit{pre}_G(s))$ —i.e., the region $\mathit{pre}_G(\sigma)$ contains the predecessors of all states in σ— let $\mathit{post}_G(\sigma) = (\cup\, s \in \sigma \mid \mathit{post}_G(s))$ —i.e., the region $\mathit{post}_G(\sigma)$ contains the successors of all states in σ— etc. As usual, if the transition graph underlies a module P, we write pre_P instead of pre_{G_P}, etc. If the transition graph is understood, we suppress the subscript altogether. ■

Remark 2.11 [Reachability] Let G be a transition graph with the initial region σ^I, let s be a state of G, and let σ^T be a region of G. The state s is reachable in i transitions iff $s \in \mathit{post}^i(\sigma^I)$, and s is reachable iff $s \in \mathit{post}^*(\sigma^I)$; that is, $\mathit{post}^*(\sigma^I)$ is the reachable region of G. The transition graph G is finitely reaching iff there is a nonnegative integer i such that $\mathit{post}^*(\sigma^I) = (\cup\, j \leq i \mid \mathit{post}^j(\sigma^I))$. The answer to the reachability question (G, σ^T) is YES iff $\mathit{post}^*(\sigma^I) \cap \sigma^T \neq \emptyset$ or, equivalently, iff $\sigma^I \cap \mathit{pre}^*(\sigma^T) \neq \emptyset$. ■

Enumerative graph search

Algorithm 2.2 shows a generic schema for graph search. As the algorithm finds new reachable states, they are explored by traversing all transitions to successor states. Throughout the algorithm, the multiset τ, which is called the *frontier*, contains the states that have been found but not yet explored; the set σ^R always contains the states that have been both found and explored. Algorithm 2.2 is said to be *enumerative*, because the states in the frontier τ are processed one state at a time. Therefore the multiset τ is best implemented by an enumeration of its members. If τ is implemented as a queue (when choosing

Algorithm 2.2 [Enumerative Graph Search] (schema)

```
Input: a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$.
Output: the reachable region $\sigma^R$ of $G$.
Local: a multiset $\tau$ of states from $\Sigma$.

Initialize $\sigma^R$ to $\emptyset$;
Initialize $\tau$ to $\sigma^I$;
while $\tau \neq \emptyset$ do
  Choose a state $s$ in $\tau$, and remove $s$ from $\tau$;
  if $s \notin \sigma^R$ then
    Add $s$ to $\sigma^R$;
    Add all states in $post_G(s)$ to $\tau$
    fi
  od.
```

a state from τ, always choose the state that was inserted least recently), then we obtain *breadth-first search.* If τ is implemented as a stack (always choose the state that was inserted most recently), then we obtain *depth-first search.* Algorithm 2.2 terminates iff the reachable subgraph G^R of the input graph G is finite. Consider a state s of G with m_s^R reachable incoming transitions; that is, $m_s^R = |pre(s) \cap \sigma^R|$. If s is reachable but not initial, then s is added to the frontier τ exactly m_s^R times; if s is initial, then s is added to τ exactly $1 + m_s^R$ times; if s is not reachable, then s is never added to τ. Every iteration of the while loop removes one state from τ. It follows that the while loop is executed $n^I + m^R$ times, where n^I is the number of initial states of G, and m^R is the number of reachable transitions.

Lemma 2.3 [Enumerative graph search] *Let G be a transition graph with n^I initial states and m^R reachable transitions. Algorithm 2.2 computes the reachable region σ^R within $n^I + m^R$ iterations of the while loop.*

Remark 2.12 [Backward search] Algorithm 2.2 performs a forward search of the input graph, starting from the initial region. Symmetrically, the graph may be searched backward from the target region, using the predecessor operation *pre* instead of the successor operation *post*. While forward search explores only reachable states, this is not necessarily the case for backward search. Hence the running time of backward search cannot be bounded by the number of reachable transitions. ■

Algorithm 2.3 [Depth-first Reachability Checking]

Input: a finitely branching transition graph G, and a finite region σ^T of G.

Output: *Done*, if the instance (G, σ^T) of the reachability problem has the answer No; a witness for the reachability question (G, σ^T), otherwise.

```
input $G$: enumgraph;  $\sigma^T$: enumreg;
local $\sigma^R$: enumreg;  $\tau$: stack of state;  $t$: state;
begin
  $\sigma^R := \mathit{EmptySet}$;
  $\tau := \mathit{EmptyStack}$;
  foreach $t$ in $\mathit{InitQueue}(G)$ do
    if $\mathit{DepthFirstSearch}(t)$ then return $\mathit{Reverse}(\tau)$ fi
    od;
  return $\mathit{Done}$
  end.

function $\mathit{DepthFirstSearch}(s)$: $\mathbb{B}$
  local $t$: state;
  begin
    $\tau := \mathit{Push}(s, \tau)$;
    if not $\mathit{IsMember}(s, \sigma^R)$ then
      if $\mathit{IsMember}(s, \sigma^T)$ then return $\mathit{true}$ fi;
      $\sigma^R := \mathit{Insert}(s, \sigma^R)$;
      foreach $t$ in $\mathit{PostQueue}(s, G)$ do
        if $\mathit{DepthFirstSearch}(t)$ then return $\mathit{true}$ fi;
        od
      fi;
    $\tau := \mathit{Pop}(\tau)$;
    return $\mathit{false}$
    end.
```

Depth-first reachability checking

Algorithm 2.3 shows a recursive depth-first implementation of graph search for solving the reachability problem. The implementation differs from the schematic Algorithm 2.2 in three respects. First, for checking reachability, the graph search is aborted when a state in the target region is found. Second, the recursive implementation of depth-first search allows the construction of witnesses without bookkeeping. Third, the input graph is assumed to be finitely branching and the input region is assumed to be finite, so that the initial region, the successor region of each state, and the target region all can be represented as queues of states. More specifically, Algorithm 2.3 uses the following abstract types. Assuming a given type **state** for states, the type of a finitely branching transition graph is **enumgraph**, and the type of a finite region is **enumreg**. The abstract type **enumgraph** supports two operations:

InitQueue: **enumgraph** $\mapsto$ **queue of state.** The operation *InitQueue*(G) returns a queue that contains the initial states of G, in some order.

PostQueue: **state** $\times$ **enumgraph** $\mapsto$ **queue of state.** The operation *PostQueue*(s, G) returns a queue that contains the successors of s, in some order.

The abstract type **enumreg** supports three standard set operations:

EmptySet: **enumreg.** The operation *EmptySet* returns the empty region.

Insert: **state** $\times$ **enumreg** $\mapsto$ **enumreg.** The operation *Insert*(s, σ) returns the region that results from adding the state s to the region σ.

IsMember: **state** $\times$ **enumreg** $\mapsto \mathbb{B}$. The operation *IsMember*(s, σ) returns *true* if the region σ contains the state s, and otherwise returns *false*.

If all states in the target region are unreachable and the reachable subgraph of the input graph is finite, then the algorithm terminates once every reachable state is found; if some state in the target region is reachable in a finite number of transitions, then the algorithm may terminate even if the reachable subgraph is infinite. This is because as soon as a state in the target region σ^T is found, the search is aborted. At this point, the stack τ of unexplored states contains a witness for the given reachability question, in reverse order. To see this, observe that τ always contains an initialized trajectory of the input graph G, in reverse order.

Lemma 2.4 [Partial correctness of depth-first reachability checking] *If Algorithm 2.3 terminates, then it solves the reachability question* (G, σ^T) *and returns a witness, if one exists.*

2.3.2 Enumerative Graph and Region Representations

For the analysis of the time and space requirements of Algorithm 2.3, we need to agree on the representation of the abstract types **enumgraph** and **enumreg**. For this purpose, we restrict ourselves to finite input graphs. We distinguish between two cases, depending on whether or not the type **state** is atomic.

- In the *state-level model*, every variable of type **state** is stored in constant space, and constant time is required for every read or write access to a state. This is the standard model used in the analysis of graph algorithms. It is appropriate if the number of states is bounded. For example, for computers with 64-bit words, the state-level model is realistic if the number of states does not exceed 2^{64}; otherwise, the storage of a state requires more than a single word.
- The *bit-level model* is more detailed and makes no assumptions about the number of states. If the total number of states is n, then in the bit-level model, every variable of type **state** is stored in $\Theta(\log n)$ space, and $\Theta(\log n)$ time is required for every read or write access to a state. The bit-level model is of particular interest in computer-aided verification, where we regularly encounter very large state spaces.

In the following analysis, we consider a transition graph G with n states, n^I initial states, and m transitions, and we consider a region σ of G. We first discuss state-level data structures for representing G and σ, and then we move on to bit-level data structures.

State-level data structures

The finite transition graph G can be represented using adjacency lists, by a record $\{G\}_{se}$ with two components:

$$\textbf{enumgraph} = (\textbf{queue of state}) \times (\textbf{array}[\textbf{state}]\ \textbf{of queue of state})$$

The first component of $\{G\}_{se}$ is a queue that contains the initial states of G. The second component of $\{G\}_{se}$ is an array, indexed by the states of G, which points, for each state s, to a queue that contains the successors of s. The record $\{G\}_{se}$ is called the *state-enumerative representation of the transition graph G*, because it is built from atomic components of the type **state** to facilitate the enumerative graph operations *InitQueue* and *PostQueue*. The state-level data structure $\{G\}_{se}$ requires $\Theta(n+m)$ space and supports the operations *InitQueue* and *PostQueue* in constant time. The *state-enumerative representation of the region* σ is a boolean array, denoted $\{\sigma\}_{se}$, which is indexed by the states of G, so that a state s is contained in σ iff $\{\sigma\}_{se}[s] = \mathit{true}$:

$$\textbf{enumreg} = \textbf{array}[\textbf{state}]\ \textbf{of}\ \mathbb{B}$$

The state-level data structure $\{\sigma\}_{se}$ requires $\Theta(n)$ space and supports the enumerative region operations *EmptySet*, *Insert*, and *IsMember*: the first in $O(n)$ time, the second and third in constant time.

Remark 2.13 [Space-efficient state-level data structures] The state-enumerative graph and region representations $\{G\}_{se}$ and $\{\sigma\}_{se}$ optimize the running time, in the state model, of Algorithm 2.3. If we want to optimize, instead, the space requirements of the data structures, different choices are necessary. We can define an alternative state-enumerative graph representation $\{G\}_{se}^T$ which uses $\Theta(n^I + m)$ space, and an alternative state-enumerative region representation $\{\sigma\}_{se}^T$ which uses $\Theta(|\sigma|)$ space, both of which are optimal. In both cases, we replace the array indexed by states with a balanced binary search tree over states: the second component of the record $\{G\}_{se}^T$ is a tree whose nodes represent the states that have nonempty queues of successors; the nodes of the tree $\{\sigma\}_{se}^T$ represent the states that are members of the region σ. The search-tree implementations of the abstract data types **enumgraph** and **enumreg** support the operations *InitQueue* and *EmptySet* in constant time, the operation *PostQueue* in $O(\log n)$ time, and the operations *Insert* and *IsMember* in $O(\log |\sigma|)$ time. ■

Bit-level data structures

In the bit-level model, we cannot have arrays indexed by states, as the index elements are no longer representable by a fixed number of bits. Without loss of generality, we assume that each state of the transition graph G is identified by a unique sequence of $\lceil \log n \rceil$ bits. For example, the transition graph G_P of a module with k boolean variables has 2^k states, and each state can be represented by a sequence of k bits denoting the values of the module variables. The *bit-enumerative representation* $\{\sigma\}_{be}$ *of the region* σ is a binary tree whose paths represent the states of G that are members of σ. The height of the tree is $\lceil \log n \rceil$, the number of leaves is $|\sigma|$, and the total number of nodes is $\Theta(|\sigma| \cdot (1 + \log n - \log |\sigma|))$ or, less precisely, $\Theta(\min(|\sigma| \cdot \log n, n))$. In particular, if n is a power of 2 and σ contains all n states, then $\{\sigma\}_{be}$ is the complete binary tree with $2n$ nodes. The *bit-enumerative representation* $\{G\}_{be}$ *of the transition graph* G is, like the state-enumerative representation, a record whose first component is a queue of the initial states, and whose second component is an index structure over states which points to queues of successor states. The index structure is implemented as a binary tree whose paths represent the states of G that have nonempty queues of successors. A queue of pairwise distinct states can be implemented in a space-efficient way by sharing common suffixes of the bitvector representations of the states in the queue. This is achieved by a binary tree with child-to-parent pointers, whose leaf-to-root paths represent the states in the queue, and whose leaves are connected by pointers that represent the order of the states in the queue. If the queue contains ℓ states, then the bit-level queue representation requires $\Theta(\min(\ell \cdot \log n, n))$ bits. The following lemma completes the space and time analysis of the bit-enumerative data structures.

Lemma 2.5 [Bit-enumerative graph and region representations] *Let G be a transition graph with n states, n^I initial states, and m transitions, and let s be a sequence of $\lceil \log n \rceil$ bits. The bit-enumerative graph representation $\{G\}_{be}$ uses $\Theta(\min(n^I \cdot \log n, n) + \min(m \cdot \log n, n^2))$ space. The operations InitQueue$(\{G\}_{be})$ and PostQueue$(s, \{G\}_{be})$ require $O(1)$ and $O(\log n)$ time, respectively. Let σ be a region of G. The bit-enumerative region representation $\{\sigma\}_{be}$ uses $\Theta(\min(|\sigma| \cdot \log n, n))$ space. The operation EmptySet requires $O(1)$ time; the operations Insert$(s, \{\sigma\}_{be})$ and IsMember$(s, \{\sigma\}_{be})$ each require $O(\log n)$ time.*

Exercise 2.8 {T2} [Proof of Lemma 2.5] Let G be a transition graph whose states are the bitvectors of length k, and let σ be a region of G. Give formal definitions of the bit-enumerative graph and region representations $\{G\}_{be}$ and $\{\sigma\}_{be}$, and prove Lemma 2.5. ■

Time and space requirements of depth-first reachability checking

To determine the time complexity of Algorithm 2.3, let n be the total number of states of the input graph, let n^I be the number of initial states, and let m^R be the number of reachable transitions. Recall the analysis of the schematic Algorithm 2.2. In particular, if s is an initial state with m_s^R reachable incoming transitions, then the function *DepthFirstSearch* is invoked with input state s at most $1 + m_s^R$ times; if s is reachable but not initial, then *DepthFirstSearch* is invoked with input s at most m_s^R times; if s is unreachable, then *DepthFirstSearch* is never invoked with input s. The first time that *DepthFirstSearch* is invoked with input s, the function call performs $O(|post(s)|)$ state-level work, and $O(\log n + |post(s)|)$ bit-level work, in addition to invoking *DepthFirstSearch* for every successor of s. Each subsequent call of *DepthFirstSearch* with input state s terminates, after a single membership test *IsMember*(s, σ^R), within constant state-level time and $O(\log n)$ bit-level time. It follows that the total time required by all invocations of *DepthFirstSearch* is, in the worst case, $O(n^I + m^R)$ time in the state-level model, and $O((n^I + m^R) \cdot \log n)$ time in the bit-level model. The worst case is obtained when no state in the target region is reachable. The initialization of the region σ^R requires $O(n)$ state-level time vs. constant bit-level time. The space complexity of Algorithm 2.3 is dominated by the space requirements of the input representations. The complete analysis is summarized in the following theorem.

Theorem 2.1 [Depth-first reachability checking] *Let G be a finite transition graph with n states, of which n^I are initial, and m transitions, of which m^R are reachable. Let σ^T be a region of G. In the state-level model, given the input $\{G\}_{se}$ and $\{\sigma^T\}_{se}$, Algorithm 2.3 solves the reachability question (G, σ^T) and computes a witness, if one exists, in $O(n + m^R)$ time and $\Theta(n + m)$ space. In the bit-level model, given the input $\{G\}_{be}$ and $\{\sigma^T\}_{be}$, Algorithm 2.3 requires*

$O((n^I + m^R) \cdot \log n)$ time and $\Theta(\min(n^I \cdot \log n, n) + \min(m \cdot \log n, n^2) + \min(|\sigma^T| \cdot \log n, n))$ space.

Remark 2.14 [Time complexity of state-level reachability checking] In the state-level model, the running time of Algorithm 2.3 is proportional to the size n of the state space, no matter how quickly a state in the target region is found. This is caused by the initialization of the array $\{\sigma^R\}_{se}$ for representing the region of explored states. In practice, this behavior is undesirable, and alternative representations of the region σ^R are preferred. One such representation, based on search trees, is studied in Exercise 2.9; another one, based on hashing, and by far the most popular in practice, will be discussed in Section 2.3.3. ■

Exercise 2.9 {T2} [Space-efficient state-level data structures] Suppose that the input to Algorithm 2.3 is given by the alternative state-enumerative graph and region representations $\{G\}^T_{se}$ and $\{\sigma^T\}^T_{se}$ introduced in Remark 2.13, and that the region σ^R of explored states is stored in the same manner. What is the resulting time and space complexity of Algorithm 2.3 in the state-level model? Under which conditions on the input G and σ^T are the alternative data structures preferable in order to optimize running time? Under which conditions are they preferable in order to optimize memory space? ■

Exercise 2.10 {P2} [Nonrecursive depth-first reachability checking] Write a nonrecursive, state-level version of Algorithm 2.3 with the same time and space complexity, assuming that the input (G, σ^T) is given by the state-enumerative graph and region representations $\{G\}_{se}$ and $\{\sigma^T\}_{se}$. Be careful with the witness construction. ■

Exercise 2.11 {P3} [Breadth-first reachability checking] Algorithm 2.3 traverses the input graph in depth-first fashion. Write a breadth-first algorithm for reachability checking, including witness construction, assuming that the input (G, σ^T) is given, first, by the state-enumerative graph and region representations $\{G\}_{se}$ and $\{\sigma^T\}_{se}$, and second, by the bit-enumerative representations $\{G\}_{be}$ and $\{\sigma^T\}_{be}$. (Maintain the frontier τ of unexplored states as a queue.) Determine the time and space requirements of your algorithm in both the state-level and bit-level models. ■

Exercise 2.12 {P2} [Transition invariants] Modify Algorithm 2.3, without changing its state-level and bit-level time and space requirements, to solve the transition-reachability problem from Exercise 2.6. For this purpose, you must define state-enumerative and bit-enumerative representations for transition predicates. ■

Algorithm 2.4 [Enumerative Invariant Verification] (schema)

```
Input: a finite module $P$, and an observation predicate $r$ for $P$.
Output: Done, if the instance $(P, r)$ of the invariant-verification
    problem has the answer YES; an error trajectory for the
    invariant-verification question $(P, r)$, otherwise.

Construct the enumerative graph representation $\{G_P\}_e$;
Construct the enumerative region representation $\{[\![\neg r]\!]_P\}_e$;
Return the result of Algorithm 2.3 on the input $\{G_P\}_e$ and $\{[\![\neg r]\!]_P\}_e$.
```

2.3.3 Invariant Verification

For finite modules P, invariant-verification questions of the form (P, r) can be reduced to reachability checking, as is shown in Algorithm 2.4. In terms of the size of the input (P, r), the asymptotic amount of work for constructing and the asymptotic amount of space required for storing the enumerative representations of the transition graph G_P and the target region $[\![\neg r]\!]_P$ are independent of whether the state-level or bit-level model is used. Hence, as in Algorithm 2.4, if the argument G is a transition graph that underlies a module, we write $\{G\}_e$ as a place-holder for either the state-enumerative representation $\{G\}_{se}$ or the bit-enumerative representation $\{G\}_{be}$; similarly, we write $\{\sigma\}_e$ if the region σ is defined by a state predicate. The translation from the module P to the enumerative graph representation $\{G_P\}_e$, as well as the translation from the observation predicate r to the enumerative region representation $\{[\![\neg r]\!]_P\}_e$, may involve an exponential amount of work. To make these claims precise, we need to agree on a syntax for the legal expressions in the initial and update commands of reactive modules.

Propositional modules

The most basic type is the boolean type. In propositional modeling, we restrict ourselves to variables of this type, which are called *propositions*. If all variables of a module are propositions, then the module is said to be a *propositional module*. Every propositional module is finite, and dually, every finite module can be viewed propositionally —by replacing each variable of a finite type with k values by $\lceil k \rceil$ boolean variables. For the propositional modules, we now agree on a specific syntax. In particular, all expressions that occur in the textual description of a propositional module result from combining propositions using a standard set of logical connectives.

PROPOSITIONAL MODULE

A *proposition* is a variable of type boolean. The *propositional formulas* are the boolean expressions generated by the grammar

$$p \ ::= \ x \mid \mathit{true} \mid \mathit{false} \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p_1 \mid p_1 \rightarrow p_2 \mid p_1 \leftrightarrow p_2,$$

where x is a proposition, and p_1 and p_2 are propositional formulas. A *propositional module* is a reactive module P such that (1) all module variables of P are propositions, and (2) every expression that appears in the initial and update commands of P is a propositional formula.

Remark 2.15 [Transition graphs for propositional modules] If P is a propositional module with k variables, then the transition graph G_P has 2^k states and at most 4^k transitions. ■

From propositional modules to enumerative graph representations

Let P be a propositional module with k variables, and let $|P|$ be the number of symbols in the textual description of P. Note that $|P| > k$. In the first step of Algorithm 2.4, we need to construct the enumerative representation $\{G_P\}_e$ of the underlying transition graph. In the following analysis, it is immaterial whether or not each of the 2^k states can be stored and accessed atomically: in terms of the parameters $|P|$ and k, an asymptotically equal amount of work is required to construct, for sufficiently small k (say, $k \leq 64$), the state-level representation $\{G_P\}_{se}$ or, for arbitrary k, the bit-level representation $\{G_P\}_{be}$; we therefore use the notation $\{G_P\}_e$. To construct the queue of initial states of P, we generate each state s of P and check if s is an initial state of P. Similarly, for every state s, to construct the queue of successors of s, we generate each state t of P and check if (s,t) is a transition of P. Since each state of P is a bitvector of length k, we can generate all states in $O(2^k)$ time. The next lemma shows that each of the 2^k initiality checks and each of the 4^k transition checks can be performed in linear time. It follows that the construction of the enumerative graph representation $\{G_P\}_e$ can be completed in $O(4^k \cdot |P|)$ time.

Lemma 2.6 [Initial states and transitions for propositional modules] *Given a propositional module P, and two states s and t of P, it can be checked in $O(|P|)$ time if s is an initial state of P and if (s,t) is a transition of P.*

Exercise 2.13 {T2} [Proof of Lemma 2.6] Given a propositional module P, construct two propositional formulas q_P^I and q_P^T, whose lengths are linear in $|P|$: the *initial predicate* q_P^I is a boolean expression over the set X_P of module variables so that for every state s of P, the expression q_P^I evaluates to true in s iff s is an initial state of P; the *transition predicate* q_P^T is a boolean expression over the set $X_P \cup X_P'$ of unprimed and primed module variables so that for every

```
module $Nondet$ is
  interface $x_1, \ldots, x_k : \mathbb{B}$
  atom controls $x_1$
    initupdate
      [] $true \rightarrow x'_1 := true$
      [] $true \rightarrow x'_1 := false$
  ⋮
  atom controls $x_k$
    initupdate
      [] $true \rightarrow x'_k := true$
      [] $true \rightarrow x'_k := false$
```

Figure 2.14: Unconstrained propositional module

pair (s, t) of states of P, the expression q_P^T evaluates to true in $s \cup t'$ iff (s, t) is a transition of P. Since each state of P is a bitvector of length k, where k is the number of propositions of P, and since $|P| > k$, Lemma 2.6 follows. ■

The construction time of the enumerative graph representation $\{G_P\}_e$ is exponential in the number k of variables. This exponential amount of work cannot be avoided, as the record $\{G_P\}_e$ may require exponentially more space than the textual description of the module P. Consider the propositional module *Nondet* with k boolean interface variables such that all initial values are arbitrary, and in every update round, the values of all variables can change arbitrarily. If every variable is controlled by a separate atom, then P can be specified using $\Theta(k)$ symbols, as shown in Figure 2.14. The transition graph G_{Nondet} is the complete graph with 2^k states, all of which are initial, and 4^k transitions. It follows that the enumerative graph representation $\{G_{Nondet}\}_e$ requires $\Theta(4^k)$ space, independent of whether we can use the state-level model or must resort to the bit-level model.

Propositional invariant verification

If we restrict our attention to propositional modules, then we obtain a special case of the invariant-verification problem.

Propositional invariant-verification problem

An instance (P, r) of the invariant-verification problem is *propositional* if P is a propositional module and r is a propositional formula. The instances of the *propositional invariant-verification problem* are the propositional instances of the invariant-verification problem. The propositional instance (P, r) *has k variables* if the module P has k module variables.

Let (P, r) be a propositional instance of the invariant-verification problem with k variables. The enumerative representation $\{[\![\neg r]\!]_P\}_e$ of the target region can be constructed in $O(2^k \cdot |r|)$ time, by generating each state s of P and checking if s satisfies the predicate r. The constructed data structure $\{[\![\neg r]\!]_P\}_e$ occupies $\Theta(2^k)$ space. Both construction time and memory space are independent of whether we use the state-enumerative region representation $\{[\![\neg r]\!]_P\}_{se}$ or the bit-enumerative region representation $\{[\![\neg r]\!]_P\}_{be}$. Together with Exercise 2.13 and Theorem 2.1, this gives exponential time and space bounds for solving the propositional invariant-verification problem which are independent of the state-level vs. bit-level issue.

Theorem 2.2 [Propositional invariant verification] *Let* (P, r) *be a propositional instance of the invariant-verification problem with* k *variables. Algorithm 2.4 solves the invariant-verification question* (P, r) *and computes an error trajectory, if one exists, in* $O(4^k \cdot (|P| + |r|))$ *time and* $\Theta(4^k)$ *space.*

Exercise 2.14 {T4} [Equational modules with interpreted constants] An *ic-equational term* is either a variable or an interpreted constant. Like variables, constants are typed. Each interpreted constant denotes a fixed value of its type. Examples of interpreted constants are the boolean constant *true* and the integer constant 19. In particular, for two interpreted constants, it is known if they denote equal or different values. The *ic-equational formulas* are the boolean expressions that are generated by the grammar

$$p \ ::= \ f_1 = f_2 \mid p_1 \wedge p_2 \mid \neg p_1,$$

where f_1 and f_2 are ic-equational terms of the same type, and p_1 and p_2 are ic-equational formulas. An *ic-equational module* is a reactive module P such that (1) every guard that appears in the initial and update commands of P is an ic-equational formula, and (2) every assignment that appears in the initial and update commands of P is an ic-equational term. An instance (P, r) of the invariant-verification problem is *ic-equational* if P is an ic-equational module and r is an ic-equational formula. The instance (P, r) is *finite* if the ic-equational module P is finite. Suppose we wish to use Algorithm 2.4 for solving the ic-equational instances of the invariant-verification problem. (a) Given an ic-equational instance (P, r) of the invariant-verification problem, find a finite propositional instance (P', r') that has the same answer as (P, r). (b) Write a preprocessor that, given the ic-equational module P and the ic-equational formula r, constructs the enumerative graph representation $\{G_{P'}\}_e$ and the enumerative region representation $\{[\![\neg r']\!]_{P'}\}_e$. What are the space requirements of your representations as a function of the size of the input (P, r)? What are the running times of your preprocessor and of Algorithm 2.4 in terms of the size of (P, r)? Does it make a difference if the state-level model or the bit-level model is used? ■

Exercise 2.15 {P3} [Backward search] Write a backward-search algorithm for reachability checking, and an invariant-verification algorithm that invokes your backward-search algorithm. In your algorithms, assume that the abstract type **enumgraph** supports the operation

$$\mathit{PreQueue}\colon \textbf{state} \times \textbf{enumgraph} \mapsto \textbf{queue of state}$$

which, given a transition graph G and a state s of G, returns a queue that contains the predecessors of s, in some order. Assuming that the state-level model is adequate, suggest a representation for transition graphs that supports the three operations *InitQueue*, *PostQueue*, and *PreQueue* in constant time, and write an algorithm that constructs your representation for the transition graphs of propositional modules. Give the running time required to construct and the memory space required to store your representation. ■

2.3.4 Three Space Optimizations

The exponential space requirements of Algorithm 2.4 (Theorem 2.2) are a serious obstacle to practical applications. The problem is caused by the enumerative graph representation $\{G_P\}_e$ for the input module P, and by the enumerative region representations for the sets σ^T of target states and σ^R of explored states, all of which require space at least proportional to the number of states of P. For many invariant-verification questions, the number of states is too large for the transition graph and its regions to be stored explicitly. The following three observations allow us to reduce the space requirements of invariant verification. First, on-the-fly methods avoid the enumerative representations of the input graph and target region. Second, state-hashing methods avoid the enumerative storage of the explored states. Third, latch-reduction methods reduce both the size and number of frontier states and explored states that need to be stored.

On-the-fly graph and region representations

Consider an instance (P, r) of the invariant-verification problem. Instead of constructing, at once, the space-intensive enumerative graph representation $\{G_P\}_e$ from the input module P, we can construct portions of the graph only as needed, whenever the operations *InitQueue* and *PostQueue* are invoked in Algorithm 2.3. Similarly, instead of constructing, at once, the space-intensive region representation $\{[\![\neg r]\!]_P\}_e$ from the input predicate r, we can answer each query in Algorithm 2.3 of the form $\mathit{IsMember}(s, \sigma^T)$ by evaluating the predicate $\neg r$ in the state s. Such an "on-the-fly" implementation of Algorithm 2.3 relies on the following data structures for representing the input graph and the target region.

- On-the-fly representations for graphs are restricted to transition graphs that underly modules. Given a module P, the *on-the-fly representation* $\{G_P\}_{of}$ *for the transition graph* G_P is a queue that contains the atoms of the module P in some execution order.

- On-the-fly representations for regions are restricted to regions that are defined by state predicates. Given a module P and a state predicate q for P, the *on-the-fly representation* $\{[\![q]\!]_P\}_{of}$ *for the region* $[\![q]\!]_P$ is the predicate q.

In an on-the-fly implementation of Algorithm 2.3, the input graph G is given by the queue $\{G_P\}_{of}$ and the input region σ^T is given by the predicate $\{[\![\neg r]\!]_P\}_{of}$ (the region σ^R of explored states, which is not defined by a state predicate, has no on-the-fly representation). Clearly, the space requirements of the on-the-fly graph and region representations $\{G_P\}_{of}$ and $\{[\![\neg r]\!]_P\}_{of}$ are linear in the input (P, r) to the invariant-verification problem. The overall time and space requirements of on-the-fly invariant verification for propositional modules are analyzed in the following exercise.

Exercise 2.16 {P3} [On-the-fly invariant verification for propositional modules] Consider a propositional module P with k variables, and an observation predicate r for P. (a) Write an algorithm that computes the on-the-fly representation $\{G_P\}_{of}$ of the underlying transition graph. What is the running time of your algorithm? (b) Write algorithms for computing the operations $\mathit{InitQueue}(\{G_P\}_{of})$, $\mathit{PostQueue}(s, \{G_P\}_{of})$, and $\mathit{IsMember}(s, \{[\![\neg r]\!]_P\}_{of})$, where s is a state of P. What are the running times of your algorithms? (c) Using the state-level array representation $\{\sigma^R\}_{se}$ for the region of explored states, solve the propositional instance (P, r) of the invariant-verification problem in $O((2^k + m^R) \cdot (|P| + |r|))$ time and $\Theta(2^k + |P| + |r|)$ space, where m^R is the number of reachable transitions of the transition graph G_P. (d) Using the state-level search-tree representation $\{\sigma^R\}_{se}^T$ or the bit-level representation $\{\sigma^R\}_{be}$ for the region of explored states, solve the instance (P, r) of the invariant-verification problem in $O((n^I + m^R) \cdot (|P| + |r|))$ time and $\Theta(n^R + |P| + |r|)$ space, where n^I is the number of initial states and n^R is the number of reachable states of the transition graph G_P. ■

Remark 2.16 [On-the-fly invariant verification for finitely branching modules] The on-the-fly representation $\{G_P\}_{of}$ of the transition graph does not require that the input module P is finite. Neither does the on-the-fly representation

$\{[\![\neg r]\!]_P\}_{of}$ of the target region require that the input predicate r evaluates to true in all but finitely many states of P. Rather, the on-the-fly implementation of Algorithm 2.3 can be applied to input modules with finitely branching transition graphs and infinite target regions. As observed earlier, Algorithm 2.3 is guaranteed to terminate if the reachable subgraph G_P^R of the input module P is finite. The algorithm may terminate even when G_P^R is infinite, if it visits a state that belongs to the target region. ■

Exercise 2.17 {P3} [Integer modules with addition] The *integer terms with addition* are the nonnegative integer expressions generated by the grammar

$$f \;::=\; x \mid m \mid f_1 + f_2 \mid f_1 \dot{-} f_2,$$

where x is a variable of type $\mathbb{N}$, where m is a nonnegative integer (i.e., an interpreted constant of type $\mathbb{N}$), and f_1 and f_2 are integer terms with addition. The *integer formulas with addition* are the boolean expressions generated by the grammar

$$p \;::=\; f_1 \leq f_2 \mid p \wedge p \mid \neg p,$$

where f_1 and f_2 are integer terms with addition. An *integer module with addition* is a reactive module P such that (1) all module variables of P are of type $\mathbb{N}$, (2) every guard that appears in the initial and update commands of P is an integer formula with addition, and (3) every assignment that appears in the initial and update commands of P is an integer term with addition. The instance (P, r) of the invariant-verification problem is an *integer instance with addition* if P is an integer module with addition and r is an integer formula with addition. (a) Suppose we wish to apply an on-the-fly implementation of Algorithm 2.3 to the integer instances with addition of the invariant-verification problem. In order to obtain a finitely branching transition graph, we need to restrict ourselves to integer modules with addition which are closed. Write algorithms that, given a closed integer module P with addition, computes the operations *InitQueue* and *PostQueue* for the transition graph G_P on the fly, and write an algorithm that, given an integer formula r with addition, computes the operation *IsMember* for the region $[\![\neg r]\!]_P$ on the fly. What are the running times of your algorithms? (b) Give an example of a closed, deterministic integer module P with addition and a state s of P so that the set $pre_P(s)$ is infinite. What are the ramifications for on-the-fly backward search for integer modules with addition? ■

Hashing of explored states

On-the-fly implementations of Algorithm 2.3 reduce the space required by the transition graph G and the target region σ^T, but they do not address the space required by the region σ^R of explored states. In particular, in the state-level model, neither the array representation $\{\sigma^R\}_{se}$ nor the search-tree representation $\{\sigma^R\}_{se}^T$ perform satisfactorily in practice: the array representation $\{\sigma^R\}_e$ is exponential in the number of input variables, even if the region σ^R, which is initially empty, remains small compared to the size of the state space; the search-tree representation $\{\sigma^R\}_{se}^T$ is space-optimal, but the rebalancing overhead involved in the frequent insertions adversely affects the verification time in practice. *State hashing* is a compromise which often offers the best practical performance. In state hashing, the region σ^R is represented by a hash table $\{\sigma^R\}_{se}^H$ that consists of (1) a hash function that maps each state s to an integer between 0 and N, for a suitably chosen nonnegative integer N, and (2) an array of length N whose i-th entry, for $1 \leq i \leq N$, points to a queue of states that are mapped to i by the hash function:

$$\textbf{enumreg} \;=\; (\textbf{state} \mapsto \{0..N\}) \times (\textbf{array}[0..N] \textbf{ of queue of state})$$

The choice of N is determined by the expected number of reachable states and by the word size of the computer on which the hash table is implemented; for example, $N = 2^{64} - 1$. The hash table $\{\sigma^R\}_{se}^H$ is a state-level data structure which uses $\Theta(N + |\sigma^R|)$ space. The running time of Algorithm 2.3 depends crucially on the complexity of the membership test for the hash table $\{\sigma^R\}_{se}^H$, which in turn depends on the choice of hash function and on the ratio of N to the number of explored states. A detailed analysis of hashing can be found in [Knuth:Vol.1].

Remark 2.17 [Bit-state hashing] While hashing is an effective technique to represent the set of explored states, often the number of reachable states is too large to be stored in memory. In such cases, an approximate strategy, known as *bit-state hashing*, can be used. This approach uses a hash table of size N whose i-th entry, for $1 \leq i \leq N$, is a single bit. The insertion of a state, which is mapped to an integer i between 0 and N by the hash function, is implemented by setting the i-th bit of the hash table to 1. All hash collisions are ignored. Suppose that two states s and t are mapped to the same integer i, and s is inserted in the hash table first. When the state t is encountered, as the i-th bit of the hash table is already set, the membership test returns a positive answer. Consequently, Algorithm 2.3 does not explore the successors of t. Hence, only a fraction of the reachable region is explored. The algorithm may return *false negatives* (the false answer No for Yes-instances of the reachability problem), but no *false positives* (the false answer Yes for No-instances of the reachability problem). In particular, every error trajectory that is found indeed signals a violation of the invariant. More general approximation schemes will be discussed in detail in Chapter 5.

What fraction of the reachable region is visited by bit-state hashing depends on the choice of table size and hash function. The table size can be increased iteratively until either an error trajectory is found or all available memory space is used. The performance of bit-state hashing can be improved dramatically by using two bit-state hash tables that employ independent hash functions. Each explored state is stored in preferably both, but at least in one hash table, so that a collision occurs only if both table entries are already occupied. If N is the size of the hash tables, this strategy typically ensures that, if necessary, close to N reachable states are explored. ■

Latch reduction for event variables and history-free variables

If, for every state s of a module, the value $s(x)$ of the module variable x is not needed for determining the successors of s, then in Algorithm 2.2 the value of x does not have to be stored as part of the frontier τ of unexplored states nor as part of the region σ^R of explored states. This is the case for event variables, whose values in a given state are immaterial, and for variables whose values are never read, only awaited. The variables whose values are not read are called *history-free*; their values in a given state depend (possibly nondeterministically) on the values of other variables in the same state. The space savings that arise from not storing the values of event variables and history-free variables during graph search can be substantial. For example, in synchronous circuits, all variables that represent input and output wires of gates and latches are history-free, and only the variables that represent the internal states of the latches need to be stored. Motivated by this example, we refer to the variables that are neither event variables nor history-free as *latched*. The set of latched variables of a module can be computed easily from the module and atom declarations. The projection of the transition graph of a module to the latched variables is called the *latch-reduced transition graph* of the module.

LATCH-REDUCED TRANSITION GRAPH OF A MODULE

A variable x of the module P is *latched* if (1) x does not have the type $\mathbb{E}$, and (2) x is read by some atom of P. We write $latchX_P$ for the set of latched module variables of P. The *latch-reduced transition graph* of P is the transition graph $G_P^L = (\Sigma_P[latchX_P], \sigma_P^I[latchX_P], \rightarrow_P^L)$, where $s^L \rightarrow_P^L t^L$ iff there is a transition $s \rightarrow_P t$ such that $s^L = s[latchX_P]$ and $t^L = t[latchX_P]$.

Example 2.10 [Latch-reduced transition graph for three-bit counter] Recall the circuit from Figure 1.20 which realizes a three-bit binary counter. For the module *Sync3BitCounter*, all variables except the three output bits out_0, out_1, and out_2, are history-free. Consequently, the latch-reduced transition graph of *Sync3BitCounter* has only 8 states, which correspond to the possible values of the three output bits. Each state of the latch-reduced transition graph encodes a counter value, and there is a transition from state s to state t iff the value encoded by t is one greater (modulo 8) than the value encoded by s. ■

Exercise 2.18 {P1} [Latch-reduced transition graph for railroad control] Consider the module $\mathit{Train}_W \parallel \mathit{Train}_E \parallel \mathit{Controller1}$ of Example 2.8. Which variables are latched? Draw the latch-reduced transition graph of the module. ■

Remark 2.18 [Latch-reduced transition graphs] The latch-reduced transition graph G_P^L of a module P may be finite even if the transition graph G_P is not, and G_P^L may be finitely branching even if G_P is not. ■

The latched-reduced transition graph of a module can be used for invariant verification. Let (P, r) be an instance of the invariant verification problem. If the observation predicate r contains only latched variables —that is, $\mathit{free}(r) \subseteq \mathit{latchX}_P$— then the invariant-verification question (P, r) reduces to the reachability question $(G_P^L, [\![\neg r]\!]_P)$. If the observation predicate r refers to some history-free variables, then a transition-reachability question on the latch-reduced transition graph G_P^L needs to be answered. To see this, we make use of the following definitions.

Latch-satisfaction of a state predicate

Let P be a module, and let q be a state predicate for P. The initial state s^L of the latch-reduced transition graph G_P^L *latch-satisfies* q if there is an initial state s of P such that $s^L = s[\mathit{latchX}_P]$ and $s \models q$. The transition (s^L, t^L) of G_P^L *latch-satisfies* q if there is a transition (s, t) of P such that $s^L = s[\mathit{latchX}_P]$ and $t^L = t[\mathit{latchX}_P]$ and $t \models q$.

Exercise 2.19 {P2} [Latch-satisfaction] Consider a propositional module P and a propositional formula q which is a state predicate for P. Implement the following four functions. The function $\mathit{LatchReducedInit}(\{G_P\}_{of})$ returns a queue containing the initial states of the latch-reduced transition graph G_P^L. Given a state s^L of G_P^L, the function $\mathit{LatchReducedPost}(s^L, \{G_P\}_{of})$ returns a queue containing the successors of s^L in the latch-reduced transition graph G_P^L. Given an initial state s^L of G_P^L, the boolean function $\mathit{InitLatchSat}(s^L, \{G_P\}_{of}, q)$ checks if s^L latch-satisfies q. Given a transition (s^L, t^L) of G_P^L, the boolean function $\mathit{TransLatchSat}(s^L, t^L, \{G_P\}_{of}, q)$ checks if (s^L, t^L) latch-satisfies q. What are the running times of your algorithms in terms of the size of the input (P, q)? ■

Proposition 2.4 [Latch-reduced invariant verification] *The answer to the invariant-verification question (P, r) is* No *iff there is an initialized trajectory $\overline{s}_{1..m}^L$ of the latch-reduced transition graph G_P^L such that either $m = 1$ and the initial state s_1^L latch-satisfies r, or $m > 1$ and the transition (s_{m-1}^L, s_m^L) latch-satisfies r.*

Exercise 2.20 {T2} [Proof of Proposition 2.4] Consider a module P and three states s^L, t^L, and u^L of the latch-reduced transition graph G_P^L. Prove that if $s^L \rightarrow_P^L t^L \rightarrow_P^L u^L$, then there are three states s, t, and u of P such that $s^L = s[\mathit{latchX}_P]$ and $t^L = t[\mathit{latchX}_P]$ and $u^L = u[\mathit{latchX}_P]$ and $s \rightarrow_P t \rightarrow_P u$. Proposition 2.4 follows. ■

Proposition 2.4 gives a recipe for invariant verification using the latch-reduced transition graph instead of the full transition graph of a module: the invariant verifier Algorithm 2.4 can call the reachability checker Algorithm 2.3 on the latch-reduced transition graph G_P^L of the input module P, provided that the membership test *IsMember*(s, σ^T) for the target region is replaced by appropriate applications of the boolean functions *InitLatchSat* and *TransLatchSat*. The space savings may be substantial, as the type **state** needs to store only the values for the latched variables of P.

Exercise 2.21 {P3} [On-the-fly, latch-reduced transition-invariant verification] Since the latch-satisfaction of a state predicate is based on transitions, rather than states, latch reduction lends itself naturally to checking transition invariants (cf. Exercise 2.6). Give a detailed algorithm for solving the propositional transition-invariant verification problem, using both on-the-fly and latch-reduction techniques. Use the functions *LatchReducedInit* and *LatchReducedPost* from Exercise 2.20, and modify *TransLatchSat* for transition invariants. Choose either the state-level or the bit-level model. In either case, for every propositional instance (P, r') of the transition-invariant problem, you should aim for the time complexity $O(4^k \cdot (|P| + |r'|))$ and the space complexity $O(2^k + |P| + |r'|)$, where k is the number of latched variables of the input module P. ■

2.4 State Explosion*

The exponential difference between reachability checking for transition graphs (Theorem 2.1) and invariant verification for reactive modules (Theorem 2.2) is intrinsic and, in general, cannot be avoided: in this section, we formally prove that the complexity class of the propositional invariant-verification problem is PSPACE, and therefore, in absence of any major breakthroughs in complexity theory, the invariant-verification problem cannot be solved efficiently. The stark contrast to the complexity class of the reachability problem, NLOGSPACE, is caused by the fact that a module provides an exponentially more succinct description of a transition graph than an enumerative graph representation. This phenomenon is called *state explosion.* The source of state explosion is the number of module variables: for a module P, the number of states of the transition graph G_P grows exponentially with the number of variables of P. State explosion is the single most important obstacle to verification practice, for two reasons. First, state explosion does not arise from any peculiarities of our modeling framework —any discrete system with k boolean variables gives rise to 2^k states— and therefore is present in all modeling frameworks. Second, state explosion arises in invariant verification, which asks the simplest kind of global questions about the dynamics of a discrete system, and therefore is present for all verification questions. This prominence has thrust state explosion into the center of verification research. At the same time, the results of this section show that all approaches to alleviate the state-explosion problem are ultimately doomed to be heuristics that work well in certain limited cases. Several of the next chapters will present such heuristics which have proved useful in practice.

2.4.1 Hardness of Invariant Verification

We first prove that in the propositional case —where all variables are boolean— the invariant-verification problem is hard for PSPACE, and then, that in the general case —specifically, in the presence of integer variables— the invariant-verification problem is undecidable. Both proofs are similar, in that reactive modules are used to simulate Turing machines: polynomial-space Turing machines in the boolean case; arbitrary Turing machines in the integer case.

PSPACE-hardness of propositional invariant verification

The PSPACE-hardness of propositional invariant verification follows from the fact that with a polynomial number of boolean variables, one can simulate the behavior of a Turing machine that visits a polynomial number of tape cells.

Theorem 2.3 [Hardness of propositional invariant verification] *The propositional invariant-verification problem is* PSPACE*-hard.*

Proof. We polynomial-time reduce the acceptance problem for polynomial-space Turing machines to the propositional invariant-verification problem. We are given a deterministic Turing machine M that accepts or rejects every input in polynomial space, and we are given an input word $\overline{a}$ for M. We need to construct, in time polynomial in the specification of M and the length of $\overline{a}$, a propositional module $P_{M,\overline{a}}$ and an observation predicate $r_{M,\overline{a}}$ so that the invariant-verification problem $(P_{M,\overline{a}}, r_{M,\overline{a}})$ has the answer YES iff the Turing machine M accepts the input $\overline{a}$. Since determining whether or not a polynomial-space Turing machine accepts an input is, by definition, PSPACE-hard, it follows that the propositional invariant-verification problem is also PSPACE-hard.

As the given Turing machine M uses only polynomial space, there is a polynomial function $p(\cdot)$ so that M accepts or rejects every input of length i by visiting at most $p(i)$ tape cells. Let A be the tape alphabet of M, containing the blank letter, and let Q be the set of control modes of M, containing the initial mode q_I, the accepting mode q_A, and the rejecting mode q_R. Let n be the length of the given input word $\overline{a}$. The Turing machine starts in the control mode q_I, its read head at the first tape cell, with the first n tape cells containing the input $\overline{a}$, and the remaining $p(n) - n$ tape cells containing blanks. The Turing machine accepts the input by entering the control mode q_A, and it rejects the input by entering the control mode q_R. Let t be the number of computation steps that M needs for accepting or rejecting the input $\overline{a}$.

We construct a finite, closed, deterministic module $P_{M,\overline{a}}$, which, for simplicity, is not necessarily propositional; the task of turning $P_{M,\overline{a}}$ into an appropriate propositional module is left to the reader (Exercise 2.22). The module $P_{M,\overline{a}}$ has $p(n)$ variables, $x_1, \ldots, x_{p(n)}$, each of the finite type A, and $p(n)$ variables, $y_1, \ldots, y_{p(n)}$, each of the finite type $Q \cup \{\perp\}$. The value of x_i indicates the contents of the i-th tape cell. If the read head of M is located at the i-th tape cell, then the value of y_i indicates the control mode of M; otherwise y_i has the value $\perp$. In this way, every state s of the module $P_{M,\overline{a}}$, such that $s(y_i)$ belongs to Q for precisely one i between 1 and $p(n)$, encodes a configuration of the Turing machine M. All variables are interface variables and are controlled by a single atom, $U_{M,\overline{a}}$. The initial and update commands of $U_{M,\overline{a}}$ ensure that the unique trajectory of $P_{M,\overline{a}}$ of length t encodes the computation of M on input $\overline{a}$. The initial command of $U_{M,\overline{a}}$ contains a single guard assignment, which assigns the input letter a_i to x_i for all $1 \leq i \leq n$, assigns the blank letter to x_i for all $n < i \leq p(n)$, assigns the initial mode q_I to y_1, and assigns $\perp$ to y_i for all $1 < i \leq p(n)$. Then, the unique initial state of $P_{M,\overline{a}}$ encodes the initial configuration of M. The Turing machine M is specified by a set of transition rules, which are tuples in $(Q \times A) \times (Q \times A \times \{\mathit{left}, \mathit{right}\})$. For example, the transition rule $((q,a),(q',a',\mathit{right}))$ specifies that "if the control mode is q and the tape letter at the read head is a, then switch the control mode to q', write letter a' onto the tape, and move the read head one tape cell to the right." For each transition rule of M, the update command of $U_{M,\overline{a}}$ contains $p(n)-1$ guarded assignments, which simulate the effect of the rule. For example, for the transition rule $((q,a),(q',a',\mathit{right}))$, for each $1 \leq i < p(n)$, the update command contains the guarded assignment

$$x_i = a \wedge y_i = q \;\rightarrow\; x_i' := a';\; y_i' := \perp;\; y_{i+1}' := q'.$$

Then, for all $j \leq t$, the unique initialized trajectory of $P_{M,\overline{a}}$ of length j encodes the first j computation steps of M on input $\overline{a}$. Consequently, the observation predicate

$$r_{M,\overline{a}}\colon \quad (\wedge\ 1 \leq i \leq p(n) \mid y_i \neq q_R)$$

is an invariant of the module $P_{M,\overline{a}}$ iff the Turing machine M accepts the input $\overline{a}$. If the specification of M contains $|M|$ symbols, then the textual description of $P_{M,\overline{a}}$ has $O(|M| \cdot p(n))$ symbols, and the predicate $r_{M,\overline{a}}$ consists of $O(p(n))$ symbols; so both $P_{M,\overline{a}}$ and $r_{M,\overline{a}}$ can be constructed in time polynomial in the size of $(M, \overline{a})$. ■

Remark 2.19 [Hardness of propositional invariant verification] The module $P_{M,\overline{a}}$ constructed in the proof of Theorem 2.3 contains a single atom that controls a polynomial number of variables. Instead, we can construct a module that has polynomial number of atoms, each of which controls a single variable, reads a constant number of variables, awaits none, and has initial and update commands of constant size. Thus, the state explosion is independent of the number or complexity of atoms; it occurs when all module variables are controlled by a single atom, and when each atom controls a single variable. Also note that, in the proof of Theorem 2.3, nondeterminism plays no role in establishing PSPACE-hardness. ■

Exercise 2.22 {T3} [Hardness of propositional invariant verification] (a) Complete the proof of Theorem 2.3 by turning the module $P_{M,\overline{a}}$ and the predicate $r_{M,\overline{a}}$, in polynomial time, into a propositional module and a propositional formula with the appropriate properties. (b) The module $P_{M,\overline{a}}$ used in the proof of Theorem 2.3 is synchronous. Prove that the invariant-verification problem is PSPACE-hard even if the problem instances are restricted to propositional modules with (1) a single atom which is a speed-independent process, and (2) multiple atoms each of which is a speed-independent process controlling a single variable. ■

Undecidability of invariant verification with counters

For infinite modules, the invariant-verification question can be algorithmically undecidable. To see this, we define a class of reactive modules which make use of nonnegative integer variables in a very restricted way. A *counter* is a variable of type $\mathbb{N}$ which can be initialized to 0 or 1, tested for 0, incremented, and decremented. If all variables of a module are counters, then the module is said to be a *counter module*. Thus, the counter modules are a proper subset of the integer modules with addition from Exercise 2.17.

COUNTER MODULE

A *counter module* is a reactive module P such that (1) all module variables of P are of type $\mathbb{N}$, (2) every guard that appears in the initial and update commands of P is a finite (possibly empty) conjunction of predicates of the form $x = 0$ and $x > 0$, (3) every assignment that appears in the initial commands of P is either 0 or 1, and (4) every assignment that appears in the update commands of P has the form $x + 1$ or $x \dot{-} 1$. The variables of a counter module are called *counters*. An instance (P, r) of the invariant-verification problem is a *counter instance* if P is a counter module and the predicate r has the form $x = 0$. The instances of the *counter invariant-verification problem* are the counter instances of the invariant-verification problem.

A classical *counter machine* is a discrete, deterministic system with a finite number of control modes and a finite number of counters. Since each control mode q can be replaced by a counter, whose value is 1 iff the control is in the mode q, and otherwise 0, every counter machine can be simulated by a closed, deterministic counter module. Since every Turing machine can, in turn, be simulated,

Algorithm 2.5 [Pspace algorithm for invariant verification] (schema)

```
Input: a propositional module $P$, and a propositional formula $r$.
Output: the answer to the instance $(P, r)$ of the invariant-verification
    problem.

Let $k$ be the number of module variables of $P$;
foreach $s, t \in \Sigma_P$ do
  if $s \in \sigma_P^I$ and not $t \models r$ then
    if $PSpaceSearch(s, t, 2^k)$ then return Yes fi
    fi
  od;
return No.

function $PSpaceSearch(s, t, i)$: $\mathbb{B}$
  if $s = t$ then return true fi;
  if $i > 1$ and $s \rightarrow_P t$ then return true fi;
  if $i > 2$ then
    foreach $u \in \Sigma_P$ do
      if $PSpaceSearch(s, u, \lceil i/2 \rceil)$ and $PSpaceSearch(u, t, \lceil i/2 \rceil)$ then
        return true
        fi
      od
    fi;
  return false.
```

by a counter machine —in fact, two counters suffice to encode the contents of an unbounded number of tape cells— it follows that the reachability problem for counter machines, which asks if a given counter machine ever enters a given control mode, and therefore also the counter invariant-verification problem, is undecidable.

Theorem 2.4 [Hardness of counter invariant verification] *The counter invariant-verification problem is undecidable.*

2.4.2 Complexity of Invariant Verification

The Pspace lower bound for propositional invariant verification (Theorem 2.3) can be tightly matched by an upper bound. However, Algorithm 2.3 uses $\Omega(2^k)$ space for solving an invariant-verification question with k variables, even if on-the-fly methods are employed, and independent of state-level vs. bit-level analysis, because the region σ^R of explored states may contain up to 2^k states. A different approach is needed if we wish to use only space polynomial in k. We now present such an algorithm and give a detailed, bit-level space analysis. (Bit-level analysis is, strictly speaking, not necessary, as the state-level and bit-level space requirements of any algorithm can differ at most by a polynomial factor of k.)

Let P be a propositional module with k variables, and let r be a propositional formula that is an observation predicate for P. Since P has 2^k states, a state s of P is reachable iff there is an initialized trajectory with sink s and length at most 2^k. This suggests Algorithm 2.5, which performs a binary search to check reachability on the transition graph G_P. Given two states s and t and a positive integer i, the boolean function $\mathit{PSpaceSearch}(s, t, i)$ returns *true* iff there is a trajectory with source s and sink t of length at most i. The function is computed recursively by attempting to find a state u at the midpoint of the trajectory. Since every state of P is a bitvector of length k, all pairs of states can be enumerated one after the other in $O(k)$ space. By Lemma 2.6, given two states s and t, it can be determined in $O(|P| + |r|)$ time, and therefore linear space, if s is initial, if t satisfies r, and if t is a successor of s. In Algorithm 2.5, the space used by the first recursive call of the function *PSpaceSearch* can be reused by the second recursive call. Hence, the total space used by Algorithm 2.5 is $O(k \cdot d + |P| + |r|)$, where d is the depth of the recursion. Since each recursive call searches for a trajectory of half the length, starting from length 2^k, the depth of the recursion is bounded by k. Thus, Algorithm 2.5 can be implemented in $O(k^2 + |P| + |r|)$ space, which is quadratic in the size of the input. This establishes that the propositional invariant-verification problem can be solved in PSPACE.

Theorem 2.5 [Complexity of invariant verification] *The propositional invariant-verification problem is* PSPACE*-complete.*

Remark 2.20 [Depth-first search versus PSPACE search] Algorithm 2.5 has an $\Omega(8^k)$ time complexity, even in the state-level model, and thus pays a price in running time for achieving polynomial space. In practice, one always prefers search algorithms whose running time is at worst proportional to the number of transitions (say, $O(4^k \cdot (|P| + |r|))$ for propositional instances of the invariant-verification problem with k variables), or better yet, proportional to the number of states and reachable transitions (as is the case for on-the-fly depth-first-search). ■

Remark 2.21 [Nondeterministic complexity of invariant verification and reachability] An alternative, conceptually simpler proof that the propositional invariant-verification problem belongs to PSPACE can be based on the knowledge that deterministic and nondeterministic polynomial space coincide (i.e., every nondeterministic polynomial-space Turing machine can be simulated, using only polynomial space, by a deterministic Turing machine). Hence it suffices to give a non-

Algorithm 2.6 [Npspace schema for invariant verification]

```
Input: a propositional module $P$, and a propositional formula $r$.
Output: one of the nondeterministic runs returns Yes iff the instance $(P, r)$ of the invariant-verification problem has the answer Yes.

Let $k$ be the number of module variables of $P$;
Choose an arbitrary state $s \in \Sigma_P$;
if not $s \in \sigma_P^I$ then return No fi;
Choose an arbitrary nonnegative integer $m$ between 0 and $2^k$;
for $i := 1$ to $m$ do
  Choose an arbitrary state $t \in \Sigma_P$;
  if not $s \rightarrow_P t$ then return No fi;
  $s := t$
  od;
if not $s \models r$ then return No fi;
return Yes.
```

deterministic approach for solving the propositional invariant-verification problem in polynomial space. Such an approach is outlined in Algorithm 2.6. The nondeterministic algorithm solves propositional invariant-verification questions with k variables using only the local variables s, m, i, and t, in $\Theta(k + |P| + |r|)$ space. Essentially the same nondeterministic algorithm, applied to inputs of the form (G, σ^T), where G is a transition graph and σ^T is a region of G, shows that the reachability problem belongs to Nlogspace (i.e., the algorithm uses only logarithmic space in addition to the space occupied by the input). ■

2.5 Compositional Reasoning

Complex systems are often built from parts of small or moderate complexity. For example, circuits are built from individual gates and memory cells. In such a setting, state explosion is caused by the parallel composition of many modules, each with a small number of variables. When possible, we want to make use of the structure inherent in such designs for verification purposes. In particular, the state-explosion problem may be avoided if an invariant of a compound module can be derived from invariants of the component modules.

2.5.1 Composing Invariants

A divide-and-conquer approach to verification attempts to reduce a verification task about a complex system to subtasks about subsystems of manageable complexity. If the reduction follows the operators that are used in the construction of the complex system, then the divide-and-conquer approach is called *compositional reasoning*. Complex reactive modules are built from simple modules using the three operations of parallel composition, variable renaming, and variable hiding. Hence, for compositional invariant verification, we need to know how invariants distribute over the three module operations.

Proposition 2.5 [Compositionality of invariants] *If the observation predicate r is an invariant of the module P, then the following three statements hold.*

Parallel composition *For every module Q that is compatible with P, the observation predicate r is an invariant of the compound module $P \| Q$.*

Variable renaming *For every variable renaming ρ for the module variables of P, the renamed observation predicate $r[\rho]$ is an invariant of the renamed module $P[\rho]$.*

Variable hiding *For every interface variable x of P, the observation predicate $(\exists x \mid r)$ is an invariant of the module* **hide** x **in** P. *In particular, if x does not occur freely in r, then r is an invariant of* **hide** x **in** P.

Proof. The first part of Proposition 2.5 follows from Proposition 2.2. The second and third parts are immediate. ■

The compositionality and monotonicity of invariants (the first part of Proposition 2.5 and Remark 2.9) suggest the following verification strategy, called *compositional invariant verification*:

> Let P and Q be two compatible modules, and let r be an observation predicate for the compound module $P \| Q$. In order to show that r is an invariant of $P \| Q$, it suffices to find an observation predicate p for P, and an observation predicate q for Q, such that (1) p is an invariant of P, (2) q is an invariant of Q, and (3) the implication $p \wedge q \rightarrow r$ is valid.

The local invariants p and q represent the guarantees that the components P and Q make as to jointly maintain the global invariant r. Compositional invariant verification is usually beneficial if the state spaces of P and Q are smaller than the state space of the compound module $P \| Q$, which is the typical scenario. It should be noted, however, that the compound module may have fewer reachable states than either component module, in which case decomposition does not achieve the desired effect. This happens when the two components are tighly coupled, strongly restraining each others behaviors, and have few private variables.

Example 2.11 [Compositional verification of railroad control] Let us revisit Example 2.8 and prove that the railroad controller *Controller2* from Figure 2.9 enforces the train-safety requirement that in all rounds, at most one train is on the bridge. More precisely, we wish to establish that the observation predicate

$$r^{safe}: \quad \neg(pc_W = bridge \wedge pc_E = bridge)$$

is an invariant of the module

module *RailroadSystem2* **is**
  hide $arrive_W, arrive_E, leave_W, leave_E$ **in**
    $\|$ $Train_W$
    $\|$ $Train_E$
    $\|$ *Controller2*.

To decompose the verification problem, we observe that the controller ensures that (1) the train traveling clockwise is allowed to proceed onto the bridge only when the western signal is green and the eastern signal is red, and symmetrically, (2) the train traveling counterclockwise is allowed to proceed onto the bridge only when the eastern signal is green and the western signal is red. The conjunction of these two assertions entails the train-safety requirement. The formal argument proceeds in six steps:

1. We establish that

   $$r_W^{safe}: \quad pc_W = bridge \rightarrow (signal_W = green \wedge signal_E = red)$$

   is an invariant of the module $Train_W \| Controller2$.

2. By the second part of Proposition 2.5 (variable renaming in invariants), we deduce that the renamed predicate

   $$r_E^{safe}: \quad pc_E = bridge \rightarrow (signal_E = green \wedge signal_W = red)$$

   is an invariant of the renamed module $Train_E \| Controller2$.

3. By the first part of Proposition 2.5 (compositionality of invariants), we deduce that both r_W^{safe} and r_E^{safe} are invariants of the compound module $Train_W \| Train_E \| Controller2$.

4. By the second part of Remark 2.9 (monotonicity of invariants), we deduce that the conjunction $r_W^{safe} \wedge r_E^{safe}$ is an invariant of the module $Train_W \| Train_E \| Controller2$.

5. Since the implication $r_W^{safe} \wedge r_E^{safe} \rightarrow r^{safe}$ is valid, by the first part of Remark 2.9 (monotonicity of invariants), we deduce that r^{safe} is also an invariant of $Train_W \| Train_E \| Controller2$.

```
module $BinArbiter$ is
  private $turn\colon \{1,2\}$
  interface $akin_1, akin_2, out\colon \mathbb{B}$
  external $in_1, in_2, akout\colon \mathbb{B}$

  atom controls $turn$ reads $turn$ awaits $akout$
    init
      $[\!]\ true \to turn' := 1$
      $[\!]\ true \to turn' := 2$
    update
      $[\!]\ akout' \wedge turn = 1 \to turn' := 2$
      $[\!]\ akout' \wedge turn = 2 \to turn' := 1$

  atom controls $akin_1, akin_2, out$ awaits $turn, in_1, in_2, akout$
    initupdate
      $[\!]\ turn' = 1 \to akin_1' := akout';\ akin_2' := 0;\ out' := in_1'$
      $[\!]\ turn' = 2 \to akin_1' := 0;\ akin_2' := akout';\ out' := in_2'$
```

Figure 2.15: Two-bit round-robin arbiter

6. Finally, by the third part of Proposition 2.5 (variable hiding in invariants), we conclude r^{safe} is an invariant of the module *RailroadSystem2*.

Only the first step requires state-space exploration, namely, the solution of a reachability problem on the latch-reduced transition graph of the module $Train_W \parallel Controller2$. The savings are easy to compute: the latch-reduced transition graph of $Train_W \parallel Controller2$ has 48 states; the latch-reduced transition graph of entire system *RailroadSystem2* has 144 states. ■

Exercise 2.23 {P2} [Composing invariants] Consider the two-bit arbiter module *BinArbiter* from Figure 2.15. When the control input *akout* is high, then one of the two data inputs in_1 and in_2 is relayed to the data output *out*. If in_i is relayed, $i = 1, 2$, then the corresponding control output $akin_i$ is set to 1, and the other control output, $akin_{3-i}$, is set to 0. When the control input *akout* is low, then both control outputs $akin_1$ and $akin_2$ are set to 0, as to indicate that none of the data inputs is relayed. The variable *turn* controls which data input is relayed and alternates the two choices. (a) Using three two-bit arbiters, we can build the four-bit round-robin arbiter

```
module $QuadArbiter$ is
  hide $in_{12}, in_{34}, akin_{12}, akin_{34}$ in
    $\parallel BinArbiter[in_1, in_2, akin_1, akin_2 := in_{12}, in_{34}, akin_{12}, akin_{34}]$
    $\parallel BinArbiter[out, akout := in_{12}, akin_{12}]$
    $\parallel BinArbiter[in_1, in_2, akin_1, akin_2, out, akout := in_3, in_4, akin_3, akin_4, in_{34}, akin_{34}]$
```

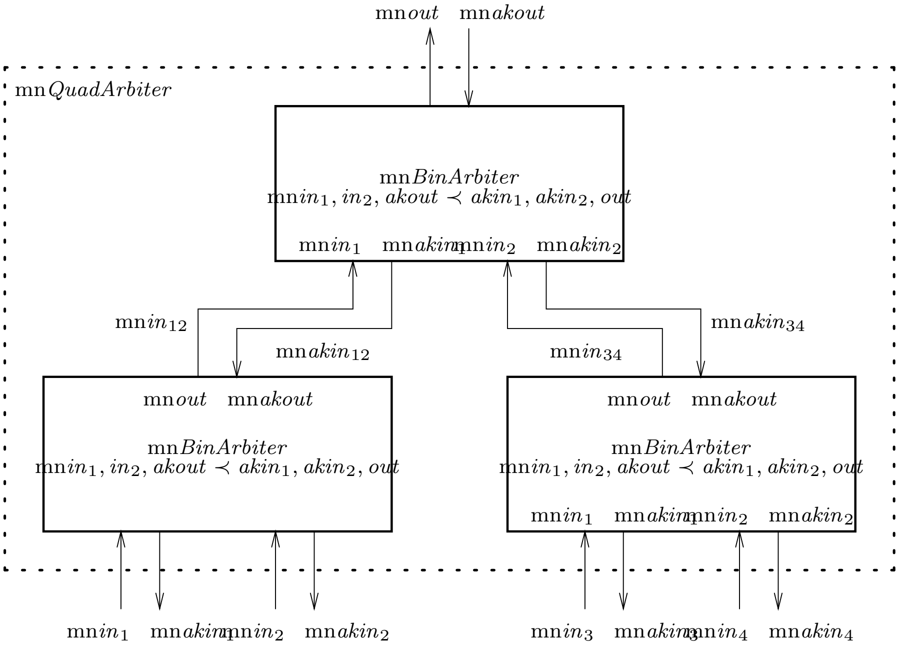

mn*out* mn*akout*
mn*QuadArbiter*
mn*BinArbiter*
mn*in*1, *in*2, *akout* ≺ *akin*1, *akin*2, *out*
mn*in*1 mn*akin*1 mn*in*2 mn*akin*2
mn*in*12
mn*akin*12
mn*in*34
mn*akin*34
mn*out* mn*akout*
mn*BinArbiter*
mn*in*1, *in*2, *akout* ≺ *akin*1, *akin*2, *out*
mn*out* mn*akout*
mn*BinArbiter*
mn*in*1, *in*2, *akout* ≺ *akin*1, *akin*2, *out*
mn*in*1 mn*akin*1 mn*in*2 mn*akin*2
mn*in*1 mn*akin*1 mn*in*2 mn*akin*2
mn*in*3 mn*akin*3 mn*in*4 mn*akin*4

Figure 2.16: Abstract block diagram for four-bit round-robin arbiter

whose abstract block diagram is shown in Figure 2.16. Why is *QuadArbiter* called a round-robin arbiter? Prove compositionally that the two observation predicates

$$p_2\colon\ (\forall\, 1 \le i \le 4 \mid akin_i \ \rightarrow\ out = in_i)$$
$$q_2\colon\ akout \ \leftrightarrow\ (\exists\, 1 \le i \le 4 \mid akin_i)$$

are invariants of the module *QuadArbiter*. First, find suitable invariants p_1 and q_1 for the two-bit arbiter *BinArbiter* and prove them by inspecting the latch-reduced transition graph (Proposition 2.4). Then, use compositional reasoning (Proposition 2.5 and Remark 2.9) to establish the invariants p_2 and q_2 of the four-bit arbiter *QuadArbiter*. (b) Compositional reasoning permits us to prove invariants for entire module classes, not only individual modules. An example of this is the class of all 2^k-bit round-robin arbiters, for positive integers k, which are built by connecting $2^k - 1$ two-bit arbiters to form a binary tree of height k. The construction of the resulting module schema *TreeArbiter* is shown in Figure 2.17. Use compositional reasoning to derive, for all $k \ge 1$, the invariants

$$p_k\colon\ (\forall\, 1 \le i \le 2^k \mid akin_i \ \rightarrow\ out = in_i)$$
$$q_k\colon\ akout \ \leftrightarrow\ (\exists\, 1 \le i \le 2^k \mid akin_i)$$

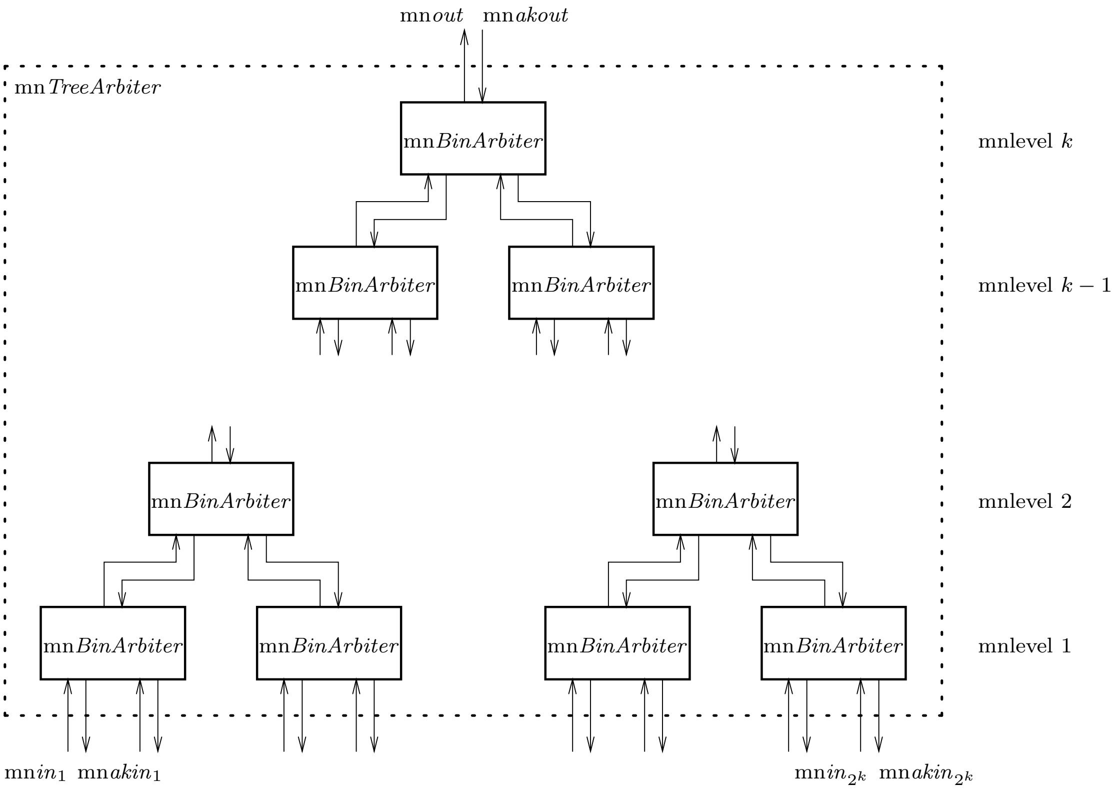


Figure 2.17: Schematic construction of 2^k-bit round-robin arbiter

of the 2^k-bit arbiter *TreeArbiter* from the invariants p_1 and q_1 of the two-bit arbiter *BinArbiter*. The integer k is a *parameter* that occurs in the module definition, the invariant definition, and the derivation. ■

2.5.2 Assuming Invariants

The compositional approach that was advocated in the previous section has limited applicability. Suppose that the predicate r is an invariant of the compound module $P \| Q$. A decomposition of the global invariant r into a local invariant p of P and a local invariant q of Q, which together imply r, may not be possible. Rather, it is often necessary to make certain assumptions about the environment of the component P in order for P to do its share in ensuring the global invariant r by maintaining the local invariant p. The assumptions on the environment of P need then to be discharged against Q. Symmetrically, Q may contribute to r by maintaining q only if its environment meets assumptions

```
module $P_1$ is
  interface $x : \mathbb{B}$
  external $y : \mathbb{B}$
  atom controls $x$ reads $x$
    init
      $[\!] \; true \; \rightarrow \; x' := 0$
    update
      $[\!] \; true \; \rightarrow \; x' := x$
```

```
module $Q_1$ is
  interface $y : \mathbb{B}$
  external $x : \mathbb{B}$
  atom controls $y$ awaits $x$
    initupdate
      $[\!] \; true \; \rightarrow \; y' := x'$
```

```
module $P_2$ is
  interface $x : \mathbb{B}$
  external $y : \mathbb{B}$
  atom controls $x$ reads $x$
    init
      $[\!] \; true \; \rightarrow \; x' := 0$
    update
      $[\!] \; true \; \rightarrow \; x' := x$
```

```
module $Q_2$ is
  interface $y : \mathbb{B}$
  external $x : \mathbb{B}$
  atom controls $y$ reads $x$
    init
      $[\!] \; true \; \rightarrow \; y' := 0$
    update
      $[\!] \; true \; \rightarrow \; y' := x$
```

```
module $P_3$ is
  interface $x : \mathbb{B}$
  external $y : \mathbb{B}$
  atom controls $x$ reads $y$
    init
      $[\!] \; true \; \rightarrow \; x' := 0$
    update
      $[\!] \; true \; \rightarrow \; x' := y$
```

```
module $Q_3$ is
  interface $y : \mathbb{B}$
  external $x : \mathbb{B}$
  atom controls $y$ reads $x$
    init
      $[\!] \; true \; \rightarrow \; y' := 0$
    update
      $[\!] \; true \; \rightarrow \; y' := x$
```

Figure 2.18: Three forms of collaboration to maintain the invariant $y = 0$

that can be discharged against P. The situation becomes apparently cyclic if the environment invariant q is the very assumption necessary for establishing that p is invariant with respect to P, and the environment invariant p is the assumption needed to establish the invariance of q with respect to Q.

For a concrete illustration, Figure 2.18 presents a series of small examples. First, consider the two modules P_1, controlling x, and Q_1, controlling y. We want to prove compositionally the invariant $y = 0$ of the compound module $P_1 \| Q_1$. This follows from the invariant $x = 0$ of P_1 and the invariant $y = x$ of Q_1. Second, consider the two modules P_2 and Q_2. In this case, the invariant $y = 0$ of the module $P_2 \| Q_2$ cannot be established compositionally, because the truth of $y = 0$ in one round depends on the truth of $x = 0$ in the previous round. Still, the problem can be solved with the help of transition invariants (Exercise 2.6). The transition invariant $y' = 0$ of $P_2 \| Q_2$ follows from the invariant $x = 0$ of P_1 and the transition invariant $y' = x$ of Q_1. Together with the fact that initially $y = 0$, this establishes the invariant $y = 0$ of $P_2 \| Q_2$. Finally, consider the two modules P_3 and Q_3. In this case, the invariant $y' = 0$ of the module $P_3 \| Q_3$ cannot be established from the transition invariants $y' = x$ of P_3 and $x' = y$ of Q_3. In order for Q_3 to guarantee the desired invariant $y = 0$, we need to assume that the environment of Q_3 maintains the invariant $x = 0$. Symmetrically, P_3 guarantees the invariant $x = 0$ only under the assumption that, in turn, the environment of P_3 keeps $y = 0$ invariant. Then, induction on the length of the initialized trajectories of the compound module $P_3 \| Q_3$ resolves the cyclic interdependence between assumptions and guarantees and establishes the global invariant $x = 0 \wedge y = 0$. This kind of compositional proof strategy is called *assume-guarantee reasoning.* In this chapter, we restrict ourselves to both assumptions and guarantees which are invariants.

Let P be a module, and let r be an external predicate for P. The assumption that the environment of P maintains the invariant r can be represented by composing P with a simple module whose only purpose is to keep r invariant and, while doing so, permitting as many initialized traces as possible. The most permissive module that ensures the invariance of r is called the *r-assertion module.*

ASSERTION MODULE

Let r be a satisfiable boolean expression whose free variables have finite types. We define the *r-assertion module* $\mathit{Assert}(r)$ as follows. The sets of private and external variables of $\mathit{Assert}(r)$ are empty; the set of interface variables of $\mathit{Assert}(r)$ is the set $\mathit{free}(r)$ of variables which occur freely in r. The module $\mathit{Assert}(r)$ has a single atom, which controls all variables in $\mathit{free}(r)$, neither reads nor awaits any variables, and has identical initial and update commands: for every valuation s of the variables in $\mathit{free}(r)$ which satisfies r, the initial and update commands contain a guarded assignment with the guard *true* and for each variable $x \in \mathit{free}(r)$, the assignment $s(x)$.

Remark 2.22 [Assertion module] The boolean expression r is an interface predicate and an invariant of the r-assertion module $Assert(r)$. In fact, r is the strongest invariant of $Assert(r)$; that is, for every invariant q of $Assert(r)$, the implication $r \rightarrow q$ is valid. All variables of the r-assertion module $Assert(r)$ are history-free, and therefore, the latch-reduced transition graph of $Assert(r)$ has a single state. ■

The following theorem formalizes our contention that if (1) the module P guarantees the invariant p assuming the environment maintains the external predicate q invariant, and (2) the module Q guarantees the invariant q assuming the environment maintains p invariant, then the compound module $P\|Q$ has both p and q as invariants. As in the third example from Figure 2.18, the proof will proceed by induction on the length of the initialized trajectories of $P\|Q$.

Theorem 2.6 [Assume-guarantee reasoning for invariants] *Let P and Q be two compatible modules. Let p be an external predicate for Q, and let q be an external predicate for P, such that all free variables of p and q have finite types. If p is an invariant of $P \,\|\, Assert(q)$, and q is an invariant of $Assert(p) \,\|\, Q$, then $p \wedge q$ is an invariant of $P\|Q$.*

Proof. Consider two compatible modules P and Q, an external predicate p for Q, and an external predicate q for P. The free variables of p and q have finite types, so that the assertion modules $Assert(p)$ and $Assert(q)$ are well-defined. Assume that p is an invariant of $P \,\|\, Assert(q)$, and q is an invariant of $Assert(p) \,\|\, Q$; that is, for every initialized trajectory $\overline{s}$ of $P \,\|\, Assert(q)$, the projection $\overline{s}[\mathsf{extl}X_P]$ is an initialized trajectory of $Assert(p)$, and for every initialized trajectory $\overline{s}$ of $Assert(p) \,\|\, Q$, the projection $\overline{s}[\mathsf{extl}X_Q]$ is an initialized trajectory of $Assert(q)$. We show that for every initialized trajectory $\overline{s}$ of $P\|Q$, the projection $\overline{s}[X_P]$ is an initialized trajectory of $P \,\|\, Assert(q)$, and the projection $\overline{s}[X_Q]$ is an initialized trajectory of $Assert(p) \,\|\, Q$. It follows that $p \wedge q$ is an invariant of $P\|Q$.

We need to define some additional concepts. Given a module R, a set $X \subseteq X_R$ of module variables is *await-closed* for R if for all variables x and y of R, if $y \prec_R x$ and $y \in X$, then $x \in X$. For an await-closed set X, the pair $(\overline{s}, t)$ consisting of an initialized trajectory $\overline{s}$ of R and a valuation t for X is an X-*partial trajectory* of R if there exists a state u of R such that (1) $u[X] = t$, and (2) $\overline{s}u$ is an initialized trajectory of R. Thus, partial trajectories are obtained by executing several complete rounds followed by a partial round, in which only some of the atoms are executed. The following two crucial facts about partial trajectories follow from the definitions.

(A) The partial trajectories of a compound module are determined by the partial trajectories of the component modules: for every pair R_1 and R_2 of compatible modules, every await-closed set X for $R_1\|R_2$, every sequence

$\overline{s}$ of states of $R_1 \| R_2$, and every valuation t for X, the pair $(\overline{s}, t)$ is an X-partial trajectory of $R_1 \| R_2$ iff $\overline{s}[X_{R_1}]$ is an $(X \cap X_{R_1})$-partial trajectory of R_1 and $\overline{s}[X_{R_2}]$ is an $(X \cap X_{R_2})$-partial trajectory of R_2. This property generalizes Proposition 2.2.

(B) If $(\overline{s}, t)$ is an X-partial trajectory of R, and u is a valuation for a set $Y \subseteq \mathsf{extl} X_R$ of external variables of R which is disjoint from X, then $(\overline{s}, t \cup u)$ is an $(X \cup Y)$-partial trajectory of R. This property is due the nonblocking nature of modules; it generalizes Lemmas 2.1 and 2.2.

Let $X_1, \ldots, X_m$ be a partition of $X_{P\|Q}$ into disjoint subsets such that (1) each X_i either contains only external variables of $P\|Q$, or contains only interface variables of P, or contains only interface variables of Q, and (2) if $y \prec_{P\|Q} x$ and $y \in X_i$, then $x \in X_j$ for some $j < i$. Define $Y_0 = \emptyset$, and for all $0 \leq i < m$, define $Y_{i+1} = Y_i \cup X_i$. Each set Y_i is await-closed for $P\|Q$. For all $0 \leq i \leq m$, let L_i be the set of Y_i-partial trajectories of $P\|Q$, and let $L = (\cup\, 0 \leq i \leq m \mid L_i)$. We define the following order $<$ on the partial trajectories in L: for $i < m$, if $(\overline{s}, t) \in L_i$ and $(\overline{s}, u) \in L_{i+1}$ and $u[Y_i] = t$, then $(\overline{s}, t) < (\overline{s}, u)$; for $i = m$, if $(\overline{s}, t) \in L_i$, then $(\overline{s}, t) < (\overline{s}t, \emptyset)$. Clearly, the order $<$ is well-founded. We prove by well-founded induction with respect to $<$ that for all $0 \leq i \leq m$, if $(\overline{s}, t)$ is a partial trajectory in L_i, then $(\overline{s}[X_P], t[X_P])$ is a $(X_i \cap X_P)$-partial trajectory of $P \,\|\, \mathit{Assert}(q)$, and $(\overline{s}[X_Q], t[X_Q])$ is a $(X_i \cap X_Q)$-partial trajectory of $\mathit{Assert}(p) \,\|\, Q$. In the following, for simplicity, we suppress projections.

Consider $(\overline{s}, \emptyset)$ in L_0. If $\overline{s}$ is the empty trajectory, then $(\overline{s}, \emptyset)$ is a trajectory of all modules. Otherwise, $\overline{s} = \overline{t}u$ for some state sequence $\overline{t}$ and state u of $P\|Q$. Then $(\overline{t}, u)$ is a Y_m-partial trajectory of $P\|Q$, and $(\overline{t}, u) < (\overline{s}, \emptyset)$. By induction hypothesis, $(\overline{t}, u)$ is a Y_m-partial trajectory of both $P \,\|\, \mathit{Assert}(q)$ and $\mathit{Assert}(p) \,\|\, Q$, and hence, $(\overline{s}, \emptyset)$ is a Y_0-partial trajectory of both $P \,\|\, \mathit{Assert}(q)$ and $\mathit{Assert}(p) \,\|\, Q$.

Consider $(\overline{s}, t)$ in L_{i+1} for some $0 \leq i < m$. Let $u = t[Y_i]$. Then $(\overline{s}, u)$ is a Y_i-partial trajectory of $P\|Q$, and $(\overline{s}, u) < (\overline{s}, t)$. By induction hypothesis, $(\overline{s}, u)$ is a Y_i-partial trajectory of both $P \,\|\, \mathit{Assert}(q)$ and $\mathit{Assert}(p) \,\|\, Q$. By fact (A) about partial trajectories, $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of P and Q, and $(\overline{s}, u)$ is a Y_i-partial trajectory of $\mathit{Assert}(p)$ and $\mathit{Assert}(q)$. It suffices to show that $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of both $\mathit{Assert}(p)$ and $\mathit{Assert}(q)$. Consider $Y_{i+1} = Y_i \cup X_i$. Without loss of generality, assume that X_i contains only interface variables of P. Then clearly, $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of $\mathit{Assert}(q)$. By fact (A), $(\overline{s}, t)$ is also a Y_{i+1}-partial trajectory of $P \,\|\, \mathit{Assert}(q)$. By fact (B), there is an initialized trajectory $\overline{s}v$ of $P \,\|\, \mathit{Assert}(q)$ such that $v[Y_{i+1}] = t$. By assumption, $\overline{s}v$ is an initialized trajectory of $\mathit{Assert}(p)$, which implies that $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of $\mathit{Assert}(p)$. ■

Remark 2.23 [Assume-guarantee reasoning]* For those interested in exactly which of our modeling choices make assume-guarantee reasoning possible, let us review the conditions of Theorem 2.6 by inspecting the proof. The condition that all variables that occur freely in the predicates p and q must have finite types is necessary for the assertion modules $Assert(p)$ and $Assert(p)$ to be well-defined, because the initial and update commands of reactive modules contain only finitely many choices in the form of guarded assignments. This requirement of reactive modules, which is useful for other purposes, is not needed for the soundness of assume-guarantee reasoning. The condition that all variables that occur freely in the predicate q must not be interface variables of the module P is necessary for the assertion module $Assert(q)$ to be compatible with P. In this strong form the condition is not needed for the soundness of assume-guarantee reasoning, as long as it is ensured that the system $P \| Assert(q)$ which represents the component P together with the invariance assumption q is nonblocking — that is, as long as the assumed invariant q does not prevent the module P from having in every state at least one successor state. A symmetric comment holds for the predicate p and the module Q. ■

Theorem 2.6, in conjunction with the monotonicity of invariants, suggests the following verification strategy, called *assume-guarantee invariant verification*:

> Let P and Q be two compatible modules, and let r be an observation predicate for the compound module $P\|Q$. In order to show that r is an invariant of $P\|Q$, it suffices to find an external predicate p for Q, and an external predicate q for P, such that (1) p is an invariant of $P \| Assert(q)$, (2) q is an invariant of $Assert(p) \| Q$, and (3) the implication $p \wedge q \rightarrow r$ is valid.

While condition (1) holds whenever p is an invariant of P, and condition (2) holds whenever q is an invariant of Q, either converse may fail. Therefore, provided one decomposes the desired invariant r of $P\|Q$ into two parts such that the first part, p, contains no interface variables of Q, and the second part, q, contains no interface variables of P, assume-guarantee invariant verification is more often successful than compositional invariant verification as presented in Section 2.5.1. Furthermore, since the latch-reduced transition graphs of the assertion modules $Assert(p)$ and $Assert(q)$ each contain only a single state, the invariant-verification problem (1) depends on the state space of P, and the invariant-verification problem (2) depends on the state space of Q, but neither involves the state space of the compound module $P\|Q$. In fact, after latch reduction, the reachable states of $P \| Assert(q)$ are a subset of the reachable states of P, so that the performance of assume-guarantee invariant verification can be no worse, only better, than the performance of compositional invariant verification.

Exercise 2.24 {`P2`} [Conditional invariant verification] An instance (P, q, r) of the *conditional invariant-verification problem* consists of a module P, an external predicate q for P, and an observation predicate r for P. The answer to the conditional invariant-verification question (P, q, r) is YES iff r is an invariant of $P \,\|\, \mathit{Assert}(q)$. Note that in the special case that the condition q is the boolean constant *true*, we obtain the standard invariant-verification problem. (a) Define the *conditional reachability problem* so that conditional invariant-verification questions of the form (P, q, r) can be reduced to conditional reachability questions of the form $(G_P, [\![q]\!]_P, [\![\neg r]\!]_P)$, which do not involve the transition graph of the compound module $P \,\|\, \mathit{Assert}(q)$. (b) Give a depth-first algorithm for conditional reachability checking and analyze its time and space requirements. Your algorithm should perform no worse, and in some cases better, than standard reachability checking. ■

Example 2.12 [] ■

Exercise 2.25 {T3} [Compositional reasoning with transition invariants] Generalize Proposition 2.5 and Theorem 2.6 to transition invariants (cf. Exercise 2.6). ■

List of Exercises

Index

Contents

Chapter 3

Symbolic Graph Representation

3.1 Symbolic Invariant Verification

As the invariant-verification problem is PSPACE-hard, we cannot hope to find a polynomial-time solution. There are, however, heuristic that perform well on many instances of the invariant-verification problem that occur in practice. One such heuristic is based on a symbolic reachability analysis of the underlying transition graph. Symbolic graph algorithms operate on implicit (or symbolic) —rather than explicit (or enumerative)— representations of regions. While an enumerative representation of the region σ is a list of the states in σ, a symbolic representation of σ is a set of constraints that identifies the states in σ. For example, for an integer variable x, the constraint $20 \leq x \leq 99$ identifies the set $\{20, 21, \ldots, 99\}$ of 80 states. A symbolic region representation may be much more succinct than the corresponding enumerative representation.

3.1.1 Symbolic Search

Consider a transition graph G with the initial region σ^I. Since the reachable region σ^R equals $(\cup i \in \ .\, post^i(\sigma^I))$, it can be computed from σ^I by iterating the function *post* on regions. This observation leads to a symbolic algorithm for solving the reachability problem. Unlike enumerative algorithms, the symbolic algorithm does not need to test membership of a state in a region, nor does it require to enumerate all states in a region.

We use the abstract type **symreg**, called *symbolic region*, to represent regions. The abstract type **symreg** supports five operations.

$\cup$: **symreg** $\times$ **symreg** $\mapsto$ **symreg.** The operation $\sigma \cup \tau$ returns the union of the regions σ and τ.

$\cap$: **symreg** $\times$ **symreg** $\mapsto$ **symreg.** The operation $\sigma \cap \tau$ returns the intersection of the regions σ and τ.

$=$: **symreg** $\times$ **symreg** $\mapsto$ $\mathbb{B}$**.** The operation $\sigma = \tau$ returns *true* iff the regions σ and τ contain the same states.

$\subseteq$: **symreg** $\times$ **symreg** $\mapsto$ $\mathbb{B}$**.** The region $\sigma \subseteq \tau$ returns *true* iff every state in σ is contained in τ.

EmptySet: **symreg.** The empty set of states.

Since $\sigma \subseteq \tau$ iff $\sigma \cup \tau = \tau$, the inclusion test can be implemented using the union and equality test. Alternatively, the equality test can be implemented using two inclusion tests: $\sigma = \tau$ iff both $\sigma \subseteq \tau$ and $\tau \subseteq \sigma$.

The abstract type of transition graphs is changed to **symgraph**, called *symbolic graph*, which supports two operations.

InitReg: **symgraph** $\mapsto$ **symreg.** The operation *InitReg*(G) returns the initial region of G.

PostReg: **symreg** $\times$ **symgraph** $\mapsto$ **symreg.** The operation *PostReg*(σ, G) returns the successor region $post_G(\sigma)$.

Algorithm 3.1 searches the input graph in a breadth-first fashion using symbolic types for the input graph and for regions. After j iterations of the repeat loop, the set σ^R equals $post^{\leq j}(\sigma^I)$. This is depicted pictorially in Figure 3.2.

Theorem 3.1 [Symbolic graph search] *Let G be a transition graph, and let σ^T be a region of G. Algorithm 3.1, if it terminates, correctly solves the reachability problem (G, σ^T). Furthermore, if there exists $j \in$ such that (1) every reachable state is the sink of some initialized trajectory of length at most j (i.e. $\sigma^R = (\cup i \leq j.\, post^i(\sigma^I))$), or (2) some state in the target region σ^T is the sink of some initialized trajectory of length j (i.e. $\sigma^T \cap post^j(\sigma^I)$ is nonempty), then the algorithm terminates within j iterations of the repeat loop.*

Algorithm 3.1 [Symbolic Search]

```
Input: a transition graph $G$, and a region $\sigma^T$ of $G$.
Output: the answer to the reachability problem $(G, \sigma^T)$.
input $G$: symgraph; $\sigma^T$: symreg;
local $\sigma^R$: symreg;
begin
  $\sigma^R := \mathit{InitReg}(G)$;
  repeat
    if $\sigma^R \cap \sigma^T \neq \mathit{EmptySet}$ then return YES fi;
    if $\mathit{PostReg}(\sigma^R, G) \subseteq \sigma^R$ then return NO fi;
    $\sigma^R := \sigma^R \cup \mathit{PostReg}(\sigma^R, G)$
    forever
  end.
```

Figure 3.1: Symbolic search

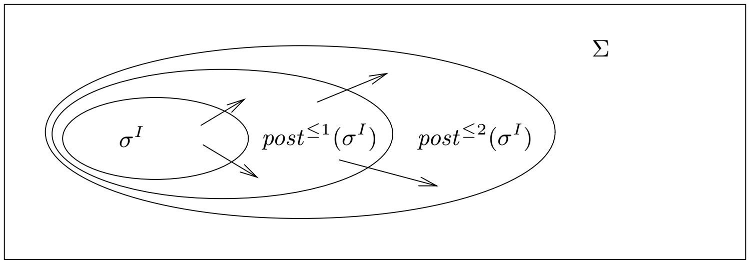


Figure 3.2: Symbolic computation of the reachable region

In particular, if the input graph G is finite with n states, then Algorithm 3.1 terminates within n iterations of the repeat loop.

Exercise 3.1 {T3} [Fixpoint view of breadth-first search] Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. The subset relation $\subseteq$ is a complete lattice on the set $\mathbf{2}^\Sigma$ of regions of G. Let f be a function from $\mathbf{2}^\Sigma$ to $\mathbf{2}^\Sigma$ such that for each region $\sigma \subseteq \Sigma$,

$$f(\sigma) \ = \ \sigma^I \cup post_G(\sigma).$$

(1) Prove that the function f is monotonic, $\bigcup$-continuous, and $\bigcap$-continuous. (2) What is the least fixpoint μf, and what is the greatest fixpoint νf? Conclude that Algorithm 3.1 can be viewed as a computation of the least fixpoint μf by successive approximation. ■

Exercise 3.2 {T2} [Enumerative region operations] Suppose we implement the abstract type **symreg** as a queue of states. Write algorithms that implement all boolean operations, emptiness test, equality test, and inclusion test. What is the cost of each operation, and what is the total cost of Algorithm 3.1? Repeat the exercise assuming that the type **symreg** is implemented as a boolean array indexed by states. ■

Exercise 3.3 {P2} [Witness reporting in symbolic search] Write an algorithm for symbolic search that, given an input transition graph G and a region σ^T of G, outputs *Done*, if the reachability problem (G, σ^T) has the answer NO; and a witness for the reachability problem (G, σ^T), otherwise. Assume that the following two additional operations are supported by our abstract types.

Element: **symreg** ↦ **state.** The operation *Element*(σ) returns a state belonging to σ.

PreReg: **symreg** × **symgraph** ↦ **symreg.** The operation *PreReg*(σ, G) returns the predecessor region $pre_G(\sigma)$.

■

3.1.2 Symbolic Implementation

Consider the transition graph over the state space Σ_X for a finite set X of typed variables. Let the type of a variable x be denoted by ${}_{x}$. The type ${}_{X}$ denotes the product type $\Pi_{x \in X}\ {}_{x}$. Then, the type **state** is the product type ${}_{X}$. The type **symreg** is parametrized by the state type **state**, and we write **symreg**[**state**]. A transition is a pair of states, and thus, has type ${}_{X} \times\ {}_{X}$. Equivalently, a transition can be viewed as a valuation for the set $X \cup X'$, where the values of the unprimed variables specify the source state of the transition and the values of the primed variables specify the sink state of the transition. Consequently, the type of a transition is ${}_{X \cup X'}$. Then, the symbolic representation of the transition graph G with the state space Σ_X is a record $\{G\}_s$ with two components, (1) the initial region $\{\sigma^I\}_s$ of type **symreg**$[\ {}_{X}]$ and (2) the transition relation $\{\rightarrow\}_s$ of type **symreg**$[\ {}_{X \cup X'}]$.

We consider the operations renaming and existential-quantifier elimination on the abstract type **symreg**$[\ {}_{X}]$.

Rename: **variable** × **variable** × **symreg**$[\ {}_{X}]$ ↦ **symreg**$[\ {}_{X[x:=y]}]$**.** For variables x and y of the same type, the operation *Rename*(x, y, σ) returns the renamed region $\sigma[x := y]$.

Exists: **variable** × **symreg**$[\ {}_{X}]$ ↦ **symreg**$[\ {}_{X \setminus \{x\}}]$**.** The operation *Exists*(x, σ) returns the region $\{s \in \Sigma_{X \setminus \{x\}} \mid (\exists m.\, s[x := m] \in \sigma)\}$.

The operations renaming and existential-quantifier elimination naturally extend to variable sets. For variable sets $X = \{x_1, \ldots x_n\}$ and $Y = \{y_1, \ldots y_n\}$ such that, for all $1 \leq i \leq n$, the variables x_i and y_i are of the same type, we write $\mathit{Rename}(X, Y, \sigma)$ for $\mathit{Rename}(x_n, y_n, \mathit{Rename}(\ldots \mathit{Rename}(x_1, y_1, \sigma)))$. Similarly, for a variable set $X = \{x_1, \ldots x_n\}$, we write $\mathit{Exists}(X, \sigma)$ for $\mathit{Exists}(x_n, \mathit{Exists}(\ldots \mathit{Exists}(x_1, \sigma)))$. We implicitly use simple forms of type inheritence and type polymorphism. For instance, if the set X of variables is a subset of Y then a region of type ${}_X$ is also a region of type ${}_Y$; if the region σ is of type ${}_X$ and the region τ is of type ${}_Y$ then the intersection $\sigma \cap \tau$ has type ${}_{X \cup Y}$.

Consider the symbolic representation $(\{\sigma^I\}_s, \{\rightarrow\}_s)$ of a transition graph G, and a region σ. Then, to compute a representation of the region $\mathit{post}_G(\sigma)$, we can proceed as follows. First, we conjunct σ with $\{\rightarrow\}_s$ to obtain the set of transitions originating in σ. Second, we project the result onto the set X' of variables by eliminating the variables in X. This yields a representation of the successor region $\mathit{post}_G(\sigma)$ in terms of the primed variables. Renaming each primed variable x' to x, then, leads to the desired result. In summary, the operation *PostReg* can be implemented using existential-quantifier elimination and renaming:

$$\mathit{PostReg}(\sigma, \{G\}_s) \;=\; \mathit{Rename}(X', X, \mathit{Exists}(X, \sigma \cap \{\rightarrow\}_s))$$

A natural choice for a symbolic representation of regions is boolean expressions. An expression is usually represented by its parse tree, or by a directed acyclic graph that allows sharing of common subexpressions to avoid duplication of syntactically identical subexpressions. If X contains only propositions, then we can represent a region as a propositional formula. The operation $\cup$ corresponds to disjunction of formulas, and the operation $\cap$ corresponds to conjunction of formulas. Both operations can be performed in constant time. The constant *EmptySet* corresponds to the formula *false*. Renaming corresponds to textual substitution, and can be performed in constant time. Existential-quantifier elimination can be performed in linear time:

$$\mathit{Exists}(x, p) \;=\; (p[x := \mathit{true}] \vee p[x := \mathit{false}]).$$

The satisfiability problem for propositional formulas is NP-complete and, therefore, the validity problem and the equivalence problem for propositional formulas are coNP-complete. The equality test corresponds to checking equivalence, and the inclusion test corresponds to checking validity of the implication. Thus, both these operations are coNP-complete.

The representation of regions as propositional formulas is possible for the propositional invariant-verification problem. Given a propositional module P with the set X of variables, the symbolic representation of the transition graph G_P consists of (1) [the *initial predicate*] a propositional formula $\{\sigma^I\}_s = q^I$ over X, and

(2) [the *transition predicate*] a propositional formula $\{\rightarrow\}_s = q^T$ over $X \cup X'$. The lengths of both formulas are linear in the size of the module description. The initial predicate is obtained by taking conjunction of the initial commands of all the atoms, and the transition predicate is obtained by taking conjunction of the update commands of all the atoms.

Remark 3.1 [Module operations for formula representation] The parallel composition of modules corresponds to the conjunction of initial and transition predicates, and the renaming of modules corresponds to the renaming of initial and transition predicates. The hiding of module variables does not affect the initial and transition predicates. ■

Analogously, enumerated formulas can be used as a symbolic representation of enumerated modules. Such a symbolic representation is linear in the size of the enumerated module. The complexities of implementing various operations on regions represented as enumerated formulas are analogous to the corresponding complexities for propositional formulas. In particular, union, intersection, renaming, and existential-quantifier elimination are easy, but equality and inclusion tests are hard, namely, coNP-complete.

Example 3.1 [Mutual exclusion] Recall Peterson's mutual-exclusion protocol from Chapter 1. A symbolic representation of the transition graph G_{P_1} has the set $\{pc_1, pc_2, x_1, x_2\}$ of variables, the initial predicate q_1^I:

$$pc_1 = \mathit{outC}, \qquad (q_1^I)$$

and the transition predicate q_1^T:

$$\begin{array}{l} \vee\, (pc_1 = \mathit{outC} \ \wedge\ pc_1' = \mathit{reqC} \ \wedge\ x_1' = x_2) \\ \vee\, (pc_1 = \mathit{reqC} \ \wedge\ (pc_2 = \mathit{outC} \ \vee\ x_1 \neq x_2) \ \wedge\ pc_1' = \mathit{inC} \ \wedge\ x_1' = x_1) \\ \vee\, (pc_1 = \mathit{inC} \ \wedge\ pc_1' = \mathit{outC} \ \wedge\ x_1' = x_1) \\ \vee\, (pc_1' = pc_1 \ \wedge\ x_1' = x_1). \end{array}$$

Given a propositional formula p, $\mathit{PostReg}(p)$ corresponds to

$$\mathit{Rename}(\{pc_1', pc_2', x_1', x_2'\}, \{pc_1, pc_2, x_1, x_2\}, \mathit{Exists}(\{pc_1, pc_2, x_1, x_2\}, p \cap q_1^T)).$$

Consider the computation of $\mathit{PostReg}(q_1^I)$. First, we take the conjunction of q_1^I and q_1^T. The resulting formula can be rewritten after simplification as

$$\begin{array}{l} \vee\, (pc_1 = \mathit{outC} \ \wedge\ pc_1' = \mathit{reqC} \ \wedge\ x_1' = x_2) \\ \vee\, (pc_1 = \mathit{outC} \ \wedge\ pc_1' = \mathit{outC} \ \wedge\ x_1' = x_1). \end{array}$$

After eliminating the variables pc_1, x_1, and x_2, we obtain

$$pc_1' = \mathit{outC} \ \vee\ pc_1' = \mathit{reqC}.$$

Finally, renaming the primed variables to unprimed ones, yields the expression

$$pc_1 = \mathit{outC} \ \vee\ pc_1 = \mathit{reqC}$$

which captures the set of states reachable from the initial states in one round. ■

Exercise 3.4 {`P2`} [Mutual exclusion] Give the initial predicate and the transition predicate for Peterson's protocol $P_1 \| P_2$. Simulate Algorithm 3.1 for checking that Peterson's protocol satisfies the mutual-exclusion requirement. For each iteration of the repeat loop, give the state predicate $\{\sigma^R\}_s$ after quantifier elimination and simplification. ■

Recall the definition of a latch-reduced transition graph of a module from Chapter 2. The initial and transition predicates of the reduced graph can be obtained from the corresponding predicates of the original graph using existential-quantifier elimination. Let P be a module with latched variables latchX_P, initial predicate q^I, and transition predicate q^T. Then, the initial region of the reduced transition graph G_P^L equals $\mathit{Exists}(X_P \backslash \mathit{latchX}_P, q^I)$, and the transition predicate of G_P^L equals $\mathit{Exists}((X_P \cup X'_P) \backslash (\mathit{latchX}_P \cup \mathit{latchX}'_P), q^T)$.

Exercise 3.5 {`P2`} [Message passing] Give the initial predicate and the transition predicate for the reduced transition graph of the send-receive protocol *SyncMsg* from Chapter 1. ■

Exercise 3.6 {`P3`} [Propositional invariant verification] (1) Write algorithms that implement the type **symreg** as propositional formulas supporting the operations $\cup$, $\cap$, $=$, *EmptySet*, $\subseteq$, *Rename*, and *Exists*. (2) Write an algorithm that, given a propositional module P, constructs the symbolic representation of the transition graph G_P. The size of the symbolic graph representation should be within a constant factor of the size of the module description. ■

Exercise 3.7 {`P3`} [Enumerated invariant verification] Write a symbolic algorithm for solving the enumerated invariant-verification problem. The size of the symbolic graph representation should be within a constant factor of the size of the module description. ■

Exercise 3.8 {`T3`} [Backward search] (1) Develop a symmetric version of Algorithm 3.1 that iterates the operator *pre* starting with the target region σ^T. Which region operations are used by your algorithm? (2) Given a symbolic representation $\{\sigma\}_s$ of the region σ, define a symbolic representation of the region $\mathit{pre}(\sigma)$ (use only positive boolean operations and quantifier elimination). ■

3.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) provide a compact and canonical representation for propositional formulas (or, equivalently, for boolean functions). The BDD-representation of propositional formulas is best understood by first considering a related structure called an *ordered binary decision graph* (BDG).

3.2.1 Ordered Binary Decision Graphs

Let X be a set containing k boolean variables. A boolean expression p over X represents a function from $\mathbb{B}^k$ to $\mathbb{B}$. For a variable x in X, the following equivalence, called the *Shannon expansion* of p around the variable x, holds:

$$p \;\equiv\; (\neg x \,\wedge\, p[x := \mathit{false}]) \,\vee\, (x \,\wedge\, p[x := \mathit{true}]).$$

Since the boolean expressions $p[x := \mathit{true}]$ and $p[x := \mathit{false}]$ are boolean functions with domain $\mathbb{B}^{k-1}$, the Shannon expansion can be used to recursively simplify a boolean function. This suggests representing boolean functions as decision graphs.

A decision graph is a directed acyclic graph with two types of vertices, *terminal* vertices and *internal* vertices. The terminal vertices have no outgoing edges, and are labeled with one of the boolean constants. Each internal vertex is labeled with a variable in X, and has two outgoing edges, a *left* edge and a *right* edge. Every path from an internal vertex to a terminal vertex contains, for each variable x, at most one vertex labeled with x. Each vertex v represents a boolean function $r(v)$. Given an assignment s of boolean values to all the variables in X, the value of the boolean function $r(v)$ is obtained by traversing a path starting from v as follows. Consider an internal vertex w labeled with x. If $s(x)$ is 0, we choose the left-successor; if $s(x)$ is 1, we choose the right-successor. If the path terminates in a terminal vertex labeled with 0, the value $s(r(v))$ is 0; if the path terminates in a terminal vertex labeled with 1, the value $s(r(v))$ is 1.

Ordered decision graphs are decision graphs in which we choose a linear order $\prec$ over X, and require that the labels of internal vertices appear in an order that is consistent with $\prec$.

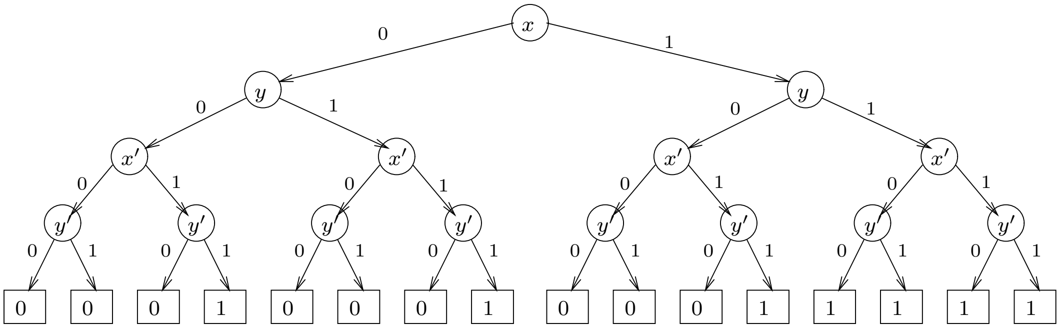


Figure 3.3: Ordered Binary Tree for $(x \wedge y) \vee (x' \wedge y')$

Ordered Binary Decision Graph

Let X be a finite set of propositions, and $\prec$ be a total order over X. An *ordered binary decision graph* B over $(X, \prec)$ consists of (1) [Vertices] a finite set V of vertices that is partitioned into two sets; *internal* vertices V^N and *terminal* vertices V^T, (2) [Root] a root vertex v^I in V, (3) [Labeling] a labeling function *label* : $V \mapsto X \cup \mathbb{B}$ that labels each internal vertex with a variable in X, and each terminal vertex with a constant in $\mathbb{B}$, (4) [Left edges] a left-child function *left* : $V^N \mapsto V$ that maps each internal vertex v to a vertex *left*(v) such that if *left*(v) is an internal vertex then *label*$(v) \prec$ *label*$($*left*$(v))$, and (5) [Right edges] a right-child function *right* : $V^N \mapsto V$ that maps each internal vertex v to a vertex *right*(v) such that if *right*(v) is an internal vertex then *label*$(v) \prec$ *label*$($*right*$(v))$.

The requirement that the labels of the children are greater than the label of a vertex ensures that every BDG is a finite and acyclic. Note that there is no requirement that every variable should appear as a vertex label along a path from the root to a terminal vertex, but simply that the sequence of vertex labels along a path from the root to a terminal vertex is monotonically increasing according to $\prec$. The semantics of BDGs is defined by associating boolean expressions with the vertices.

Boolean Function of a BDG

Given a BDG B over $(X, \prec)$, let r be a function that associates each element of V with a boolean function over X such that $r(v)$ equals *label*(v) if v is a terminal vertex, and equals

$$(\neg \mathit{label}(v) \wedge r(\mathit{left}(v))) \vee (\mathit{label}(v) \wedge r(\mathit{right}(v)))$$

otherwise. Define $r(B) = r(v^I)$ for the root v^I.

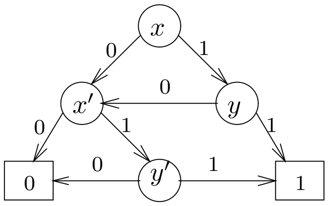


Figure 3.4: Ordered Binary Decision Diagram for $(x \wedge y) \vee (x' \wedge y')$

Example 3.2 [Binary decision graphs] A boolean constant is represented by a BDG that contains a single terminal vertex labeled with that constant. Figure 3.3 shows one possible BDG for the expression $(x \wedge y) \vee (x' \wedge y')$ with the ordering $x \prec y \prec x' \prec y'$. The left-edges are labeled with 0, and the right-edges are labeled with 1. The BDG of Figure 3.3 is, in fact, a tree. Figure 3.4 shows a more compact BDG for the same expression with the same ordering of variables. ■

Exercise 3.9 {`T3`} [Satisfying assignments] Write an algorithm that, given a BDG B over $(X, \prec)$, outputs an assignment s to X such that s satisfies $r(B)$. Write an algorithm that, given a BDG B over $(X, \prec)$, outputs the number of distinct assignments s to X such that s satisfies $r(B)$. What are the time complexities of your algorithms? ■

Two BDGs B and C are *isomorphic* if the corresponding labeled graphs are isomorphic. Two BDGs B and C are *equivalent* if the boolean expressions $r(B)$ and $r(C)$ are equivalent. If B is a BDG over $(X, \prec)$, and v is a vertex of B, then the subgraph rooted at v is also a BDG over $(X, \prec)$. Two vertices v and w of the BDG B are isomorphic, if the subgraphs rooted at v and w are isomorphic. Similarly, two vertices v and w are equivalent, if the subgraphs rooted at v and w are equivalent.

Example 3.3 [Isomorphic and equivalent BDGs] The binary decision graphs of Figures 3.3 and 3.4 are not isomorphic, but are equivalent. In Figure 3.3, the subgraph rooted at vertex v_3 is a BDG that represents the boolean expression $x' \wedge y'$. The subgraphs rooted at vertices v_3, v_4, and v_5, are isomorphic. On the other hand, the vertices v_5 and v_6 are not isomorphic to each other. ■

Remark 3.2 [Isomorphism and Equivalence of BDGs] Let B and C be two BDGs over a totally ordered set $(X, \prec)$. Checking whether B and C are isomorphic can be performed in time linear in the number of vertices in B. Isomorphic BDGs are equivalent. However, isomorphism is not necessary for equivalence, as evidenced by the two nonisomorphic, but equivalent, BDGs of Figures 3.3 and 3.4. ■

3.2.2 Ordered Binary Decision Diagrams

An ordered binary decision diagram (BDD) is obtained from a BDG by applying the following two steps:

1. Identify isomorphic subgraphs.

2. Eliminate internal vertices with identical left and right successors.

Each step reduces the number of vertices while preserving equivalence. For instance, consider the BDG of Figure 3.3. Since vertices v_3 and v_4 are isomorphic, we can delete one of them, say v_4, and redirect the right-edge of the vertex v_1 to v_3. Now, since both edges of the vertex v_1 point to v_3, we can delete the vertex v_1 redirecting the left-edge of the root v_0 to v_3. Continuing in this manner, we obtain the BDD of Figure 3.4. It turns out that the above transformations are sufficient to obtain a canonical form.

Ordered Binary Decision Diagram

An *ordered binary decision diagram* over a totally ordered set $(X, \prec)$ is an ordered binary decision graph B over $(X, \prec)$ with vertices V and root v^I such that (1) [No isomorphic subgraphs] if v and w are two distinct vertices in V, then v is not isomorphic to w, and (2) [No redundancy] for every internal vertex v, the two successors *left*(v) and *right*(v) are distinct.

The next two proposition assert the basic facts about representing boolean expressions using BDDs: every boolean function has a unique, upto isomorphism, representation as a BDD.

Proposition 3.1 [Existence of BDDs] *If p is a boolean expression over the set X of propositions and $\prec$ is a total order over X then there is a BDD B over $(X, \prec)$ such that $r(B)$ and p are equivalent.*

Proposition 3.2 [Canonicity of BDDs] *Let B and C be two BDDs over an ordered set $(X, \prec)$. Then, B and C are equivalent iff they are isomorphic.*

Exercise 3.10 {`T5`} [Existence and canonicity] Prove Proposition 3.1 and Proposition 3.2. ■

For a boolean function p and ordering $\prec$ of variables, let $B_{p,\prec}$ be the unique BDD B over $(X, \prec)$ such that $r(B)$ and p are equivalent.

Remark 3.3 [Checking Equivalence, Satisfiability, and Validity] Checking equivalence of two BDDs, with the same variable ordering, corresponds to checking isomorphism, and hence, can be performed in time linear in the number of vertices. The boolean constant 0 is represented by a BDD with a single terminal vertex labeled with 0, and the boolean constant 1 is represented by a BDD with a single terminal vertex labeled with 1. A boolean expression represented by the BDD B is satisfiable iff the root of B is not a terminal vertex labeled with 0. A boolean expression represented by the BDD B is valid iff the root of B is a terminal vertex labeled with 1. Thus, checking satisfiability or validity of boolean expressions is particularly easy, if we use BDD representation. Contrast this with representation as propositional formulas, where satisfiability is NP-complete and validity is coNP-complete. ■

The BDD of a boolean expression has the least number of vertices among all BDGs for the same expression using the same ordering.

Proposition 3.3 [Minimality of BDDs] *Let B be an BDD over an ordered set $(X, \prec)$. If C is a BDG over $(X, \prec)$ and is equivalent to B, then C contains at least as many vertices as B.*

Exercise 3.11 {T2} [Support sets] Let p be a boolean function over variables X. The *support-set* of p contains those variables x in X for which the boolean functions $p[x := \mathit{true}]$ and $p[x := \mathit{false}]$ are not equivalent. Show that a variable x belongs to the support-set of p iff some vertex in the BDD $B_{p,\prec}$ is labeled with x. ■

The size of the BDD may be exponential in the number of variables. Furthermore, one ordering may result in a BDD whose size is linear in the number of variables, while another ordering may result in a BDD whose size is exponential in the number of variables.

Example 3.4 [Variable ordering and BDD size] The size of the BDD representing a given predicate depends on the choice of the ordering of variables. Consider the predicate $(x \leftrightarrow y) \wedge (x' \leftrightarrow y')$. Figure 3.5 shows two BDDs for two different orderings. ■

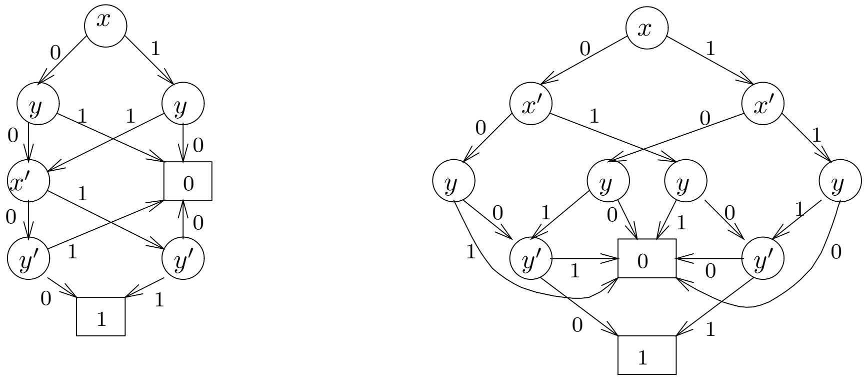
Exercise 3.12 {T2} [Representation using BDDs] Consider the boolean expression

$$(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_2 \wedge x_4) \vee (\neg x_3 \wedge x_4)$$

Choose a variable ordering for the variables $\{x_1, x_2, x_3, x_4\}$, and draw the resulting BDD. Can you reduce the size of the BDD by reordering the variables? ■

Exercise 3.13 {T3} [Exponential dependence on variable-ordering] Consider the set $X = \{x_1, x_2, \ldots x_{2k}\}$ with $2k$ variables. Consider the boolean expression

$$p\colon (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2k-1} \wedge x_{2k}).$$



Ordering: $x \prec y \prec x' \prec y'$ *Ordering:* $x \prec x' \prec y \prec y'$

Figure 3.5: Two BDDs for $(x \leftrightarrow y) \wedge (x' \leftrightarrow y')$.

Show that (1) for the ordering $x_1 \prec x_2 \prec \cdots \prec x_{2k}$ the resulting BDD has $2k+2$ vertices, and (2) for the ordering $x_1 \prec x_{k+1} \prec x_2 \prec x_{k+2} \prec \cdots \prec x_k \prec x_{2k}$, the resulting BDD has 2^{k+1} vertices. ■

Given a boolean expression p over X, the linear order $\prec$ over X is optimal for p if, for every linear order $\prec'$ over X, $B_{p,\prec'}$ has at least as many vertices as $B_{p,\prec}$. Choosing an optimal ordering can lead to exponential saving, however, computing the optimal oredring itself is computationally hard.

Proposition 3.4 [Complexity of optimal ordering] *The problem of checking, given a BDD B over $(X, \prec)$, whether the ordering $\prec$ is optimal for $r(B)$, is coNP-complete.*

There are boolean functions whose BDD representation does not depend on the chosen ordering, and the BDD representation of some functions is exponential in the number of variables, irrespective of the ordering.

Example 3.5 [BDD for parity] Let X be a set of propositions. Consider the parity function *Parity*: for an assignment s, $s(\mathit{Parity}) = 1$ if the number of variables x with $s(x) = 1$ is even, and $s(\mathit{Parity}) = 0$ if the number of variables x with $s(x) = 1$ is odd. If X contains k variables, then irrespective of the chosen ordering $\prec$, $B_{\mathit{Parity},\prec}$ contains $2k+1$ vertices. ■

Exercise 3.14 {`T3`} [BDD for addition] Let X be the set $\{x_0, x_1, y_0, y_1, \mathit{out}_0, \mathit{out}_1, \mathit{carry}\}$. Choose an appropriate ordering of the variables, and construct the BDD for the requirement that the output $\mathit{out}_1\mathit{out}_0$, together with the carry bit carry, is the sum of the inputs x_1x_0 and y_1y_0. Is your choice of ordering optimal? ■

Exercise 3.15 {T5} [BDD for multiplication] Let X contain $2k$ variables $\{x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}\}$. For $0 \leq i < 2k$, let $Mult_i$ denote the boolean function that denotes the i-th bit of the product of the two k-bit inputs, one encoded by the bits x_j and another encoded by the bits y_j. Prove that, for every ordering $\prec$ of the variables in X, there exists an index $0 \leq i < 2k$ such that the BDD $B_{Mult_i,\prec}$ has at least $2^{k/8}$ vertices. This shows that BDDs do not encode multiplication compactly irrespective of the variable ordering. ■

Exercise 3.16 {T3} [Deterministic finite automata and BDDs] Given a variable ordering, a boolean formula can be defined as a regular language over $\mathbb{B}$. A boolean expression p defines the region $[\![p]\!]$ that contains all states s that satisfy p. Let $x_1 \prec \ldots \prec x_k$ be the enumeration of the variables according to $\prec$. Each state s is an assignment to the variables in X, and can be represented by the vector $s(x_1) \ldots s(x_k)$ over $\mathbb{B}$. Thus, $[\![p]\!]$ is a language over $\mathbb{B}$ that contains words of length k. Since $[\![p]\!]$ is a finite language, it is regular, and can be defined by a deterministic finite automaton (DFA). DFAs also have canonical forms: every regular language is accepted by a unique minimal DFA. This suggests that we can use DFAs as a representation of boolean functions. (1) Give an example of a boolean expression whose DFA representation is smaller than its BDD representation. (2) Give an example of a boolean expression whose BDD representation is smaller than its DFA representation. ■

3.2.3 Operations on BDDs

Let us turn our attention to implementing regions as BDDs. Every vertex of a BDD is itself a BDD rooted at that vertex. This suggests that a BDD can be represented by an index to a global data structure that stores vertices of all the BDDs such that no two vertices are isomorphic. There are two significant advantages to this scheme, as opposed to maintaining each BDD as an individual data structure. First, checking isomorphism, or equivalence, corresponds to comparing indices, and does not require traversal of the BDDs. Second, two non-isomorphic BDDS may have isomorphic subgraphs, and hence, can share vertices.

Let X be an ordered set of k propositions. The type of states is then $\mathbb{B}^k$. The type of BDDs is **bdd**, which is a pointer or an index to the global data structure *BddPool*. The type of *BddPool* is **set of bddnode**, and it stores the vertices of BDDs. The vertices of BDDs have type **bddnode** which equals $([1..k] \times \textbf{bdd} \times \textbf{bdd}) \cup \mathbb{B}$. The type **bddnode** supports the following operations:

Label: **bddnode** $\mapsto [1..k]$. The operation *Label*(v), for an internal vertex v, returns the index of the variable labeling v.

Left: **bddnode** $\mapsto$ **bdd**. The operation *Left*(v), for an internal vertex v, returns a pointer to the global data structure *BddPool* that points to the left-successor of v.

```
function $MakeVertex$
Input: $i\colon [1..k]$, $B_0, B_1\colon$ bdd.
Output: $B$ : bdd such that $r(B)$ is equivalent to $(\neg x_i \wedge r(B_0)) \vee$
    $(x_i \wedge r(B_1))$.
begin
  if $B_0 = B_1$ then return $B_0$ fi;
  if $\neg IsMember((i, B_0, B_1), BddPool)$ then
     $Insert((i, B_0, B_1), BddPool)$ fi;
  return $Index((i, B_0, B_1))$
  end.
```

Figure 3.6: Creating BDD vertices

Right: **bddnode** ↦ **bdd**. The operation $Right(v)$, for an internal vertex v, returns a pointer to the global data structure *BddPool* that points to the right-successor of v.

The type **set of bddnode**, apart from usual operations such as *Insert* and *IsMember*, also supports

Index: **bddnode ↦ bdd.** For a vertex v in *BddPool*, $Index(v)$ returns a pointer to v.

$[\cdot]$: **set of bddnode** × **bdd** ↦ **bddnode**. The operation $BddPool[B]$ returns the root vertex of the BDD B.

For such a representation, given a pointer B of type **bdd**, we write $r(B)$ to denote the propositional formula associated with the BDD that B points to. To avoid duplication of isomorphic nodes while manipulating BDDs, it is necessary that new vertices are created using the function *MakeVertex* of Figure 3.6. If no two vertices in the global set *BddPool* were isomorphic before an invocation of the function *MakeVertex*, then even after the invocation, no two vertices in *BddPool* are isomorphic. The global set *BddPool* initially contains only two terminal vertices, and internal vertices are added only using *MakeVertex*.

Exercise 3.17 {T3} [BDD with complement edges] A binary decision graph with *complement edges* (CBDG) is a binary decision graph B with an additional component that classifies each right-edge as positive $+$ or negative $-$. The predicate $r(v)$, for an internal vertex v, is redefined so that $r(v)$ equals $(\neg label(v) \wedge r(left(v))) \vee (label(v) \wedge r(right(v)))$ if the right-edge of v is positive, and $(\neg label(v) \wedge r(left(v))) \vee (label(v) \wedge \neg r(right(v)))$ otherwise. Thus, when the right-edge is negative, we negate the function associated with the right-child. For instance, in Figure 3.7, the vertex labeled with y represents the function $y \wedge z$, while the root represents the function $(x \wedge \neg(y \wedge z))$.

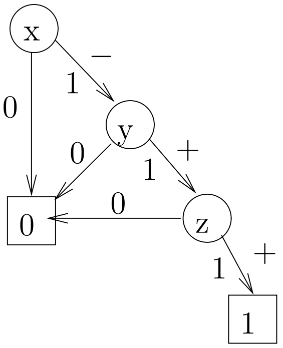


Figure 3.7: A decision graph with complement edges

(1) Define binary decision digrams with complement edges (CBDD) as a subclass of CBDGs such that every boolean function has a unique representation as a CBDD. (2) Is there a function whose CBDD representation is smaller than its BDD representation? (3) Suppose we store vertices of all the functions in the same global pool. Show that CBDD representation uses less space than BDDs. (4) Show that the canonicity property is not possible if we allow complementing left-edges also. ■

To be able to build a BDD-representation of a given predicate, and to implement the primitives of the symbolic reachability algorithm, we need a way to construct conjunctions and disjunctions of BDDs. We give a recursive algorithm for obtaining conjunction of BDDs. The algorithm is shown in Figure 3.8.

Consider two vertices v and w, and we wish to compute the conjunction $r(v) \wedge r(w)$. If one of them is a terminal vertex, then the result can be determined immediately. For instance, if v is the terminal vertex labeled with *false*, then the conjunction is also *false*. If v is the terminal vertex labeled with *true*, then the conjunction is equivalent to $r(w)$.

The interesting case is when both v and w are internal vertices. Let i be the minimum of the indices labeling v and w. Then, x_i is the least variable in the support-set of $r(v) \wedge r(w)$. The label of the root of the conjunction is i, the left-successor is the BDD for $(r(v) \wedge r(w))[x_i := 0]$, and the right-successor is the BDD for $(r(v) \wedge r(w))[x_i := 1]$. Let us consider the left-successor. Observe the equivalence

$$(r(v) \wedge r(w))[x_i := 0] \equiv r(v)[x_i := 0] \wedge r(w)[x_i := 0] \quad (1).$$

If v is labeled with i, the BDD for $r(v)[x_i := 0]$ is the left-successor of v. If the label of v exceeds i, then the support-set of $r(v)$ does not contain x_i, and the BDD for $r(v)[x_i := 0]$ is v itself. The BDD for $r(w)[x_i := 0]$ is computed similarly, and then the function *Conj* is applied recursively to compute the conjunction (1).

The above described recursion may call the function *Conj* repeatedly with the same two arguments. To avoid unnecessary computation, a table is used that stores the arguments and the corresponding result of each invocation of *Conj*. When *Conj* is invoked with input arguments v and w, it first consults the table to check if the conjunction of $r(v)$ and $r(w)$ was previously computed. The actual recursive computation is performed only the first time, and the result is entered into the table.

A *table* data structure stores values that are indexed by keys. If the type of values stored is **value**, and the type of the indexing keys is **key**, then the type of the table is **table of key** × **value**. The abstract type **table** supports the retrieval and update operations like arrays: $T[i]$ is the value stored in the table T with the key i, and the assignment $T[i] := m$ updates the value stored in T for the key i. The constant table *EmptyTable* has the default value $\perp$ stored with every key. Tables can be implemented as arrays or as hash-tables. The table used by the algorithm uses a pair of BDDs as a key, and stores BDDs as values.

Let us analyze the time-complexity of Algorithm 3.2. Suppose the BDD pointed to by B_0 has m vertices and the BDD pointed to by B_1 has n vertices. Let us assume that the implementation of the set *BddPool* supports constant time membership tests and insertions, and the table *Done* supports constant-time creation, access, and update. Then, within each invocation of *Conj*, all the steps, apart from the recursive calls, are performed within constant time. Thus, the time-complexity of the algorithm is the same, within a constant factor, of the total number of invocations of *Conj*. For any pair of vertices, the function *Conj* produces two recursive calls only the first time *Conj* is invoked with this pair as input, and zero recursive calls during the subsequent invocations. This gives an overall time-complexity of $O(m \cdot n)$.

Proposition 3.5 [BDD conjunction] *Given two BDDs B_0 and B_1, Algorithm 3.2 correctly computes the BDD for $r(B_0) \wedge r(B_1)$. If the BDD pointed to by B_0 has m vertices and the BDD pointed to by B_1 has n vertices, then the time-complexity of the algorithm is $O(m \cdot n)$.*

Exercise 3.18 {T3} [Quadratic lower bound] The time complexity of Algorithm 3.2 is proportional to the product of the number of vertices in the component BDDs. Show that the size of the BDD representing conjunction of two BDDs grows as the product of the sizes of the components, in the worst case. ■

Algorithm 3.2 [Conjunction of BDDs]
Input: B_0, B_1 : **bdd**.
Output: B : **bdd** such that $r(B)$ is equivalent to $r(B_0) \wedge r(B_1)$.
Local: *Done* : **table of** (**bdd** × **bdd**) × **bdd**.

```
begin
   $Done := EmptyTable$;
   return $Conj(B_0, B_1)$
   end.

function $Conj$
input $B_0, B_1$ : bdd
output $B$ : bdd
local $v_0, v_1$ : bddnode; $B, B_{00}, B_{01}, B_{10}, B_{11}$ : bdd; $i, j : [1 \ldots k]$
begin
   $v_0 := BddPool[B_0]$;
   $v_1 := BddPool[B_1]$;
   if $v_0 = 0$ or $v_1 = 1$ then return $B_0$ fi;
   if $v_0 = 1$ or $v_1 = 0$ then return $B_1$ fi;
   if $Done[(B_0, B_1)] \neq \perp$ then return $Done[(B_0, B_1)]$ fi;
   if $Done[(B_1, B_0)] \neq \perp$ then return $Done[(B_1, B_0)]$ fi;
   $i := Label(v_0)$; $B_{00} := Left(v_0)$; $B_{01} := Right(v_0)$;
   $j := Label(v_1)$; $B_{10} := Left(v_1)$; $B_{11} := Right(v_1)$;
   if $i = j$ then $B := MakeVertex(i, Conj(B_{00}, B_{10}),\ Conj(B_{01}, B_{11}))$
    fi;
   if $i < j$ then $B := MakeVertex(i, Conj(B_{00}, B_1), Conj(B_{01}, B_1))$ fi;
   if $i > j$ then $B := MakeVertex(j, Conj(B_0, B_{10}), Conj(B_0, B_{11}))$ fi;
   $Done[(B_0, B_1)] := B$;
   return $B$
   end.
```

Figure 3.8: Conjunction of BDDs

Exercise 3.19 {`T3`} [Cost of recomputation] Show that, removing the update step $Done[(B_0, B_1)] := B$ from Algorithm 3.2, makes the worst-case time complexity exponential. ■

Exercise 3.20 {`T3`} [From expressions to BDDs] Give an algorithm to construct the BDD-representation of a boolean expression given as a propositional formula. What is the time-complexity of the algorithm? ■

Exercise 3.21 {`T3`} [Substitution in BDDs] (1) Let p be a propositional formula, x be a variable, and $m \in \mathbb{B}$ be a value. Give an algorithm to construct the BDD representation of $p[x := m]$ from the BDD-representation of p. (2) Give algorithms for computing the disjunction and existential-quantifier elimination for BDDs. ■

Exercise 3.22 {`T3`} [BDD operations] The control schema underlying Algorithm 3.2 works for both conjunction and disjunction. What are the conditions on a binary operator on BDDs that make that control schema work? ■

3.2.4 Symbolic Search using BDDs

We have all the machinery to implement the symbolic search algorithm using BDDs as a representation for symbolic regions. It can be used immediately to solve the propositional invariant verification problem, and can be adopted to solve the enumerated invariant verification problem.

Exercise 3.23 {`T3`} [Symbolic verification of enumerated modules] An enumerated variable whose type contains k values can be encoded by $\lceil log\ k \rceil$ boolean variables. Give an algorithm which, given an enumerated invariant verification problem (P, p), constructs a propositional invariant verification problem (Q, q) such that (1) the answers to the two verification problems are identical, and (2) the description of the problem (Q, q) is at most $\lceil log\ k \rceil$ times the description of (P, p), where every variable of P has an enumerated type with at most k values. ■

We will consider some heuristics that are useful in different steps of applying Algorithm 3.1 for solving the invariant verification problem using BDDs.

Given the propositional invariant verification problem (P, p), the first step is to construct the symbolic representations for the target region p, the initial predicate q^I of P, and the transition predicate q^T of P. The BDD representations of a boolean expression can be exponentially larger, and is very sensitive to the ordering of variables. Heuristics are usually tailored to keep the representation of q^T small.

Algorithm 3.3 [Symbolic Search using the Frontier]

Input: a transition graph G, and a region σ^T of G.
Output: the answer to the reachability problem (G, σ^T).

```
input $G$: symgraph; $\sigma^T$: symreg;
local $\sigma^R$: symreg; $\sigma^F$: symreg;
begin
  $\sigma^R := \mathit{InitReg}(G)$; $\sigma^F := \sigma^R$;
  repeat
    if $\sigma^F \cap \sigma^T \neq \mathit{EmptySet}$ then return YES fi;
    if $\mathit{PostReg}(\sigma^F, G) \subseteq \sigma^R$ then return NO fi;
    $\sigma^F := \mathit{PostReg}(\sigma^F, G) \setminus \sigma^R$;
    $\sigma^R := \sigma^R \cup \sigma^F$
    forever
  end.
```

Figure 3.9: Symbolic search using the frontier

Computing with the frontier

Recall the symbolic algorithm of Figure 3.2 that searches the input graph in a breadth-first manner. A modified version of the algorithm is shown in Figure 3.9. In addition to the region σ^R containing the states known to be reachable, Algorithm 3.3 maintains an additional region σ^F called the *frontier.* In each iteration of the repeat loop, the frontier σ^F equals the subset of the reachable region σ^R containing only the newly discovered states. More precisely, after j iterations of the repeat loop, the region σ^R equals $\mathit{post}^{\leq j}(\sigma^I)$, and the frontier σ^F equals $\mathit{post}^{\leq j}(\sigma^I) \setminus \mathit{post}^{\leq j-1}(\sigma^I)$. Consequently, to find out which states are reachable in $j+1$ rounds, it suffices to compute the successor region of the frontier σ^F rather than the reachable region σ^R. In practice, Algorithm 3.9 typically outperforms Algorithm 3.2 in terms on computational resource requirements. The correctness statement for Algorithm 3.3 is identical to the one for Algorithm 3.1.

Theorem 3.2 [Symbolic graph search using the frontier] *Let G be a transition graph, and let σ^T be a region of G. Algorithm 3.3, if it terminates, correctly solves the reachability problem (G, σ^T). Furthermore, if there exists $j \in$ such that (1) every reachable state is the sink of some initialized trajectory of length at most j (i.e. $\sigma^R = (\cup i \leq j.\, \mathit{post}^i(\sigma^I))$), or (2) some state in the target region σ^T is the sink of some initialized trajectory of length j (i.e. $\sigma^T \cap \mathit{post}^j(\sigma^I)$ is nonempty), then the algorithm terminates within j iterations of the repeat loop.*

Remark 3.4 [Optimizing the frontier computation] For the correctness of Algorithm 3.3, it suffices if, after j iterations of the repeat loop, the frontier σ^A is any region that contains at least $post^{\leq j}(\sigma^I) \setminus post^{\leq j-1}(\sigma^I)$ and at most $post^{\leq j}(\sigma^I)$. That is, the frontier should at least contain the newly discovered states, and should not contain any state not known to be reachable. This gives us freedom to choose the frontier so as to reduce the sise of its representation. ■

Choice of variable ordering

First, we need to choose an ordering $\prec$ of the variables in $X_P \cup X'_P$. One of the steps in the computation of the successor-region is to rename all the primed variables to unprimed variables. This renaming step can be implemented by renaming the labels of the internal vertices of the BDD if the ordering of the primed variables is consistent with the ordering of the corresponding unprimed variables. This gives us our first rule for choosing $\prec$;

> *Variable Ordering Rule 1:* For a reactive module P, choose the ordering $\prec$ of the variables $X_P \cup X'_P$ so that for all variables $x, y \in X_P$, $x \prec y$ iff $x' \prec y'$.

As the second rule of thumb to minimize the size of B_{q^T}, a variable should appear only after all the variables it depends on:

> *Variable Ordering Rule 2:* For a reactive module P, choose the ordering $\prec$ of the variables $X_P \cup X'_P$ so that (1) for every atom U in atoms_P, if $x \in \mathsf{read}X_U$ and $y \in \mathsf{ctr}X_U$ then $x \prec y'$, and (2) if the variable y awaits the variable x then $x' \prec y'$.

Since the set of atoms of a module is consistent, there exists an ordering that satisfies both the above rules.

Exercise 3.24 {T2} [Disjoint Dependence] Let p be a boolean function with support-set X, and let $\prec_1 = x_1, x_2, \ldots x_k$ be an optimal ordering of X for p. Let q be a boolean function with support-set Y, and let $\prec_2 = y_1, y_2 \ldots y_l$ be an optimal ordering of Y for q. Suppose $X_1 \cap X_2$ is empty.

(1) Show that the ordering $x_1, x_2, \ldots x_k, y_1, y_2 \ldots y_l$ is an optimal ordering for $p \vee q$ as well as for $p \wedge q$. (2) If the optimal BDD for p has m_1 vertices and the optimal BDD for q has m_2 vertices, how many vertices does the optimal BDD for $p \wedge q$ have? ■

Exercise 3.24 suggests that the variables that are related to each other should be clustered together. In particular, instead of ordering all the primed variables after all the unprimed variables, we can try to minimize the distance between a primed variable and the unprimed variables it depends on.

> *Variable Ordering Rule 3:* For a reactive module P, choose the ordering $\prec$ of the variables $X_P \cup X'_P$ so as to minimize the sum of the differences $j - i$ such that j-th variable according to $\prec$ is a primed variable x' that depends on the i-th variable according to $\prec$.

Exercise 3.25 {P2} [Ordering of module variables] The BDD for the transition relation of an atom is the disjunction of the BDDs for individual guarded assignments in its update command. Give a heuristic to order the variables that attempts to exploit this structure. Write an algorithm that, given a module P, constructs a variable ordering according to the heuristics discussed so far. ■

Partitioned transition relation

Another approach to constructing the BDD for the transition predicate is to avoid building it a priori.

CONJUNCTIVE PARTITIONING

A *conjunctively-partitioned* representation of a boolean expression p is a set $\{B_1, \ldots B_k\}$ of BDDs such that p is equivalent to the conjunction $r(B_1) \wedge \cdots \wedge r(B_k)$.

The total number of vertices in a conjunctively partitioned representation can be exponentially smaller than the number of vertices in the BDD for p itself. Since the transition predicate of a module is the conjunction of the update commands of its atoms, it leads to a natural conjunctively partitioned representation. This approach avoids building the BDD for the entire transition relation. Let us revisit the computation of the reachable region using symbolic search. Starting from the initial predicate $q_0 = q^I$, we successively compute the predicates q_i using

$$q_{i+1} \;=\; (\exists X.\, q_i \,\wedge\, q^T)[X' := X] \quad (2).$$

The computation (2) involves obtaining the conjunction $q_i \wedge q^T$. If q^T is conjunctively partitioned, $q_1^T \wedge \cdots \wedge q_k^T$, then we need to compute the conjunction $q_i \wedge q_1^T \wedge \cdots \wedge q_k^T$. Thus, it appears that we have only postponed the complexity of conjoining multiple BDDs, and in fact, we are now required to construct the conjunction at each step. The advantage is that, the size of $B_{q_i \wedge q^T}$ can be much smaller than the size of B_{q^T}. This is because q_i contains only reachable states, and thus constrains the source states for q^T. Thus, the conjunctively partitioned representation is an *on-the-fly* symbolic representation. While computing the BDD for the conjunction $q_i \wedge q_1^T \wedge \cdots q_k^T$, we do not need to construct B_{q^T} first. We can compute the conjunction from left to right, starting with the construction of $B_{q_i \wedge q_1^T}$.

Remark 3.5 [Two-level representations] The idea of on-the-fly representation of the transition relation can be extended further. Each conjunct q_i^T, representing the update command of a single atom, is a disjunction of predicates obtained from individual guarded assignments. When the number of guarded assignments in a single update command is large, it may not be suitable to construct the BDD for the update command, and maintain it in a disjunctively-partitioned form. It is possible to maintain two-level, or even multi-level, BDD-representation of the transition predicate, and manipulate it only during the computation of (2). ■

Early quantifier elimination

Observe the equivalence

$$\exists x.\, p \wedge q \;\equiv\; p \wedge \exists x.\, q \qquad\qquad \text{if } x \text{ is not in the support-set of } p$$

If x is a support variable for q, then $B_{\exists x.\, q}$ can have less number of vertices than B_q. This implies that to compute $B_{\exists x.\, p \wedge q}$ from B_p and B_q, if x is not a support variable of p, the best strategy is to first compute $B_{\exists x.\, q}$ and then conjoin it with B_p. This strategy to apply the projection operation before conjunction is called *early quantifier-elimination*.

The computation (2) requires the computation of the projection of a conjunction of BDDs onto a set of variables, and thus, demands the use of early quantifier-elimination. It is even more effective if we are computing the reachable region of the reduced transition graph of a module P. Then the transition predicate q^T is itself a projection of the transition predicate of G_P onto the latched variables. The region predicate q_i is an expression over the latched variables $latchX_P$, and (2) is rewritten as

$$q_{i+1} \;=\; (\exists X_P.\, \exists X'_P \backslash latchX'_P.\, q_i \wedge q_1^T \wedge \cdots \wedge q_k^T)[latchX'_P := latchX_P] \qquad (3).$$

Example 3.6 [Early-quantifier elimination in computing *PostReg*] Consider a synchronous 3-bit counter that is incremented in every round. The variables of the module are out_0, out_1, out_2. The update of the bit out_0 is specified by

$$q_0^T:\; out'_0 \leftrightarrow \neg out_0,$$

of the bit out_1 by

$$q_1^T:\; out'_1 \leftrightarrow out_0 \oplus out_1$$

and of the bit out_2 by

$$q_2^T:\; out'_2 \leftrightarrow (out_0 \wedge out_1) \oplus out_2,$$

where $\oplus$ denotes exclusive-or operation. The transition predicate is the conjunction $q_0^T \wedge q_1^T \wedge q_2^T$. A good variable-ordering according to our rules is

$$out_0 \prec out_0' \prec out_1 \prec out_1' \prec out_2 \prec out_2'.$$

We choose not to construct q^T a priori, but to maintain it in a conjunctively partitioned form. Consider a predicate p, and we wish to compute the predicate

$$\exists\{out_0, out_1, out_2\}.\,(p \wedge q_0^T \wedge q_1^T \wedge q_2^T). \quad (1)$$

If we first compute the conjunction $p \wedge q_0^T$, we cannot eliminate any of the variables. However, since conjunction is associative and commutative, we can choose the ordering of the conjuncts. In particular, if we first conjunct p and q_2^T, then none of the remaining conjuncts depend on out_2, and hence, we can eliminate out_2. Thus, (1) can be rewritten to

$$\exists\, out_0.\,(q_0^T \wedge \exists\, out_1.\,(q_1^T \wedge \exists\, out_2.\,(q_2^T \wedge p))).$$

Thus, the support-sets of various BDDs are examined to determine an ordering of the conjuncts so as to eliminate as many variables as early as possible. ■

Exercise 3.26 {T3} [Don't care simplification] This exercise describes an effective heuristic for simplifying each conjunct of the transition relation with respect to the current reachable region Given predicates p and q over the set X of variables, the predicate r over X is said to be a p-simplification of q if $p \rightarrow (q \leftrightarrow r)$ is valid. Thus, a p-simplification of q must include states that satisfy both p and q, must exclude states that satisfy p but not q, and can treat the remaining states as "don't care" states. Observe that a p-simplification of q is not unique, and its BDD representation can have much smaller size compared to the BDD representation of q.

(1) Give an algorithm that computes, given the BDD representations of two predicates p and q, BDD representation of some p-simplification of q. The objective should be to reduce the size of the output BDD by exploiting the freedom afforded by "don't care" when p is false. (2) Show that the conjunction $q \wedge q_1^T \wedge \cdots q_k^T$ during the computation of the successor region can be replaced by $q \wedge r_1 \wedge \cdots r_k$ where each r_j is a q-simplification of the conjunct q_j^T. Observe that this strategy simplifies different conjuncts independently of each other, rather than sequentially as in early quantifier elimination, and hence, is not sensitive to the ordering of the conjuncts. ■

Dynamic variable reordering

As Algorithm 3.1 computes the reachable region using successive approximations, the BDD representing q_i grows with i, and successive applications of *PostReg* require more and more time. If the number of vertices exceeds beyond a threshhold, we can attempt to reduce its size by choosing a new ordering of the variables. This step is called *dynamic variable reordering.*

If we want to switch the ordering of two adjacent variables, its effect on the BDD is local: if we switch the variables x_i and x_{i+1}, then the structure of the BDD for vertices labeled less than i or greater than $i+1$ does not change.

Exercise 3.27 {T4} [Swapping of Variables] Consider a boolean function p, and its $B_{p,\prec}$ using the linear order $x_1 \prec \cdots \prec x_n$. For $1 \leq i < n$, let $\prec_{[i/i+1]}$ be the linear order

$$x_1 \prec_{[i/i+1]} \cdots \prec_{[i/i+1]} x_{i+1} \prec_{[i/i+1]} x_i \prec_{[i/i+1]} x_{i+1} \prec_{[i/i+1]} \cdots x_n$$

obtained by swapping the order of x_i and x_{i+1}. Show that (1) a vertex of $B_{p,\prec}$ that is labeled with an index j that is greater than $i+1$ is also a vertex in $B_{p,\prec_{[i/i+1]}}$, and (2) the subgraph of $B_{p,\prec}$ containing vertices labeled with indices less than i is isomorphic to the subgraph of $B_{p,\prec_{[i/i+1]}}$ containing vertices labeled with indices less than i. Give an algorithm to construct $B_{p,\prec_{[i/i+1]}}$. What is the complexity of your algorithm? ■

This suggests a variety of greedy heuristics for reordering. Suppose i is the index such that maximum number of vertices of B are labeled with i. Then, we can try swapping x_i with x_{i+1} or x_i with x_{i-1}. If one of the swaps reduces the size of the BDD, we choose the resulting order. Alternatively, one can try successive local swaps till BDD size is reduced. Note that if we update the ordering of the variables for one BDD, all the other BDDs need to be updated. Thus, dynamic variable reordering is a costly step, and is invoked only in extreme cases. Efficient memory management techniques for garbage collection of BDD-vertices not in use is also essential in practice.

Appendix: Notation

Orders and fixpoints

A binary relation $\preceq$ over a set A is a *preorder* if it is reflexive and transitive, and a *partial order* if it is reflexive, transitive, and antisymmetric. Let $\preceq$ be a preorder on A, and let $\succeq$ be the inverse of $\preceq$. If $\preceq$ is a partial order, then so is $\succeq$; if $\preceq$ is an equivalence, then $\succeq = \preceq$. The preorder $\preceq$ induces the equivalence $\preceq \cap \succeq$, which is called the *kernel* of $\preceq$. Given $B \subseteq A$ and $a \in A$, we write $B \preceq a$ if for all $b \in B$, $b \preceq a$; in this case, a is an *upper $\preceq$-bound* for B. An upper $\succeq$-bound for B is called a *lower $\preceq$-bound* for B. Moreover, a is a *least upper $\preceq$-bound* for B if (1) a is an upper $\preceq$-bound for B, and (2) a is a lower $\preceq$-bound for the set of upper $\preceq$-bounds for B. A least upper $\succeq$-bound is called a *greatest lower $\preceq$-bound.* If $\preceq$ is a partial order, then all least upper $\preceq$-bounds and all greatest lower $\preceq$-bounds are unique. The partial order $\preceq$ is a *complete lattice* if all subsets of A have least upper $\preceq$-bounds (and hence greatest lower $\preceq$-bounds); in this case we write $\bigvee B$ for the least upper $\preceq$-bound for B, and $\bigwedge B$ for the greatest lower $\preceq$-bound for B. Every complete lattice $\preceq$ has the unique lower $\preceq$-bound $\bigvee \emptyset$ for A, called *bottom*, and the unique upper $\preceq$-bound $\bigwedge \emptyset$ for A, called *top*. For example, the subset relation $\subseteq$ is a complete lattice on the powerset $\mathbf{2}^C$ of any set C. In this case, the least upper $\subseteq$-bound for a set E of subsets of C is the union $\bigcup E$; the greatest lower $\subseteq$-bound for E is the intersection $\bigcap E$; the bottom is the empty set $\emptyset$; and the top is the entire set C.

Let $\preceq$ be a complete lattice on A with the bottom $\perp$ and the top $\top$, and let f be a function from A to A. The function f is *monotonic* if for all $a, b \in A$, if $a \preceq b$ then $f(a) \preceq f(b)$. The argument $a \in A$ is a *fixpoint* of f if $f(a) = a$. If f is monotonic, then $\preceq$ is a complete lattice on the fixpoints of f.[1] The bottom fixpoint, denoted μf, is called the *least fixpoint* of f; the top fixpoint, denoted νf, is the *greatest fixpoint* of f. A *chain* is a set $B \subseteq A$ such that for all $a, b \in B$, either $a \preceq b$ or $b \preceq a$. The function f is *$\bigvee$-continuous* if for all chains $B \subseteq A$, $f(\bigvee B) = \bigvee f(B)$, and *$\bigwedge$-continuous* if for all chains $B \subseteq A$, $f(\bigwedge B) = \bigwedge f(B)$. If f is monotonic and $\bigvee$-continuous, then $\mu f = \bigvee \{f^\kappa(\perp) \mid \kappa \in \quad\}$.[2][3] If, in addition, A is countable, then $\mu f = \bigvee \{f^i(\perp) \mid i \in \quad\}$; if A is finite, then there is a natural number i such that $\mu f = f^i(\perp)$. Analogous results apply to the greatest fixpoint of a monotonic and $\bigwedge$-continuous function.

Exercise 3.28 {T4} [Fixpoint theorems] Prove all claims made in the previous paragraph. ■

[1] This is the *Knaster-Tarski fixpoint theorem.*

[2] By , we denote the set of ordinals. For a limit ordinal λ, let $f^\lambda(a) = \bigvee \{f^\kappa(a) \mid \kappa < \lambda\}$.

[3] This is the *Kleene fixpoint theorem.* It is usually stated for *complete partial orders* (c.p.o.s), which require only the existence of least upper bounds on chains, and no greatest lower bounds.

Contents

Chapter 4

Graph Minimization

This chapter defines observational equivalence among states and the resulting reductions in the state-space.

4.1 Graph Partitions

State-space abstraction decreases the size of a transition graph by collapsing equivalent states. We begin by defining the quotient graphs induced by equivalence relations on the states.

Let $G = (\Sigma, \sigma^I, \to)$ be a transition graph. An equivalence $\cong \subseteq \Sigma^2$ on the state space is called a *G-partition*. The *quotient* of G under $\cong$, denoted $G/_{\cong}$, is the transition graph $(\Sigma/_{\cong}, \sigma^I/_{\cong}, \to_{\cong})$, where $\sigma \to_{\cong} \tau$ iff there are two states $s \in \sigma$ and $t \in \tau$ such that $s \to t$.

In other words, the states of the quotient $G/_{\cong}$ are regions of the transition graph G, namely, the $\cong$-equivalence classes. A $\cong$-equivalence class is initial iff it contains an initial state. The $\cong$-equivalence class τ is a successor of the $\cong$-equivalence class σ iff a state in σ has a successor state in τ.

Let (P, p) be an invariant-verification problem. Instead of solving the reachability question $(G_P, [\![\neg p]\!])$, we choose a G_P-partition $\cong$, construct the quotient $G_P/_{\cong}$, and solve the reachability problem $(G_P/_{\cong}, [\![\neg p]\!]/_{\cong})$. If the answer to $(G_P/_{\cong}, [\![\neg p]\!]/_{\cong})$ is No, then the answer to the original question $(G_P, [\![\neg p]\!])$ is also No, and p is an invariant of the reactive module P. This verification technique is called *abstraction*, because the transition graph G_P is abstracted into the quotient $G_P/_{\cong}$ by omitting detail, such as the values of certain variables. If, on the other hand, the answer to the reachability problem $(G_P/_{\cong}, [\![\neg p]\!])$ is Yes, then p may or may not be an invariant of P. Abstraction, therefore, is a sound but incomplete verification technique for checking invariants.

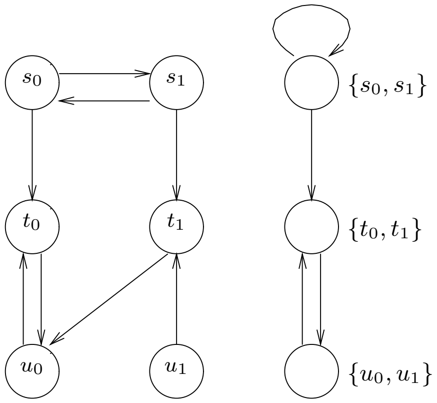


Figure 4.1: Quotient graph

Example 4.1 [Quotient graph] Consider the transition graph of Figure 4.1. The partition $\cong$ contains 3 equivalence classes $\{s_0, s_1\}$, $\{t_0, t_1\}$, and $\{u_0, u_1\}$. The corresponding quotient graph $G/_{\cong}$ has 3 states. To check whether the state s_0 is reachable from the state t_0 in G, we can check whether the state $\{s_0, s_1\}$ is reachable from the state $\{t_0, t_1\}$ in $G/_{\cong}$, and we get the correct answer No. On the other hand, to check whether the state u_1 is reachable from the state t_0 in G, we can check whether the state $\{u_0, u_1\}$ is reachable from the state $\{t_0, t_1\}$ in $G/_{\cong}$, and we get the wrong answer Yes. ■

4.1.1 Reachability-preserving Partitions

We are interested in conditions under which quotients preserve the reachability properties of a transition graph. These quotients, which are called stable, lead to abstractions that are both sound and complete for checking invariants.

> The G-partition $\cong$ is *stable* if for all states s, s', and t, if $s \cong t$ and $s \to s'$, then there is a state t' such that $s' \cong t'$ and $t \to t'$. The quotient $G/_{\cong}$ is *stable* if the partition $\cong$ is stable.

In other words, for two equivalence classes σ and τ of a stable partition $\cong$,

some state in σ has a successor in τ

is equivalent to

every state in σ has a successor in τ.

Example 4.2 [Stable partition] For the transition graph of Figure 4.1, the partition $\cong$ is stable. If we add a transition to G, from state t_0 to state t_1, the partition $\cong$ will no longer be stable. ■

Suppose that $\cong$ is a stable G-partition, and let σ^T be a block of $\cong$. The reachability problem (G, σ^T) can be reduced to a reachability problem over the quotient $G/_{\cong}$, whose state space may be much smaller than the state space of G. Indeed, $\Sigma/_{\cong}$ may be finite for infinite Σ.

Theorem 4.1 [Stable partitioning] *Let $G/_{\cong}$ be a stable quotient of the transition graph G, and let σ be a block of $\cong$. Then the two reachability problems (G, σ) and $(G/_{\cong}, \sigma/_{\cong})$ have the same answer.*

Proof. If the answer to (G, σ^T) is YES, then the answer to $(G/_{\cong}, \sigma^T/_{\cong})$ is also YES. This direction does not require $\cong$ to be stable or σ^T to be a block of $\cong$.

Suppose the answer to $(G/_{\cong}, \sigma^T/_{\cong})$ is YES. Consider the witness trajectory $\sigma_0 \rightarrow_{\cong} \cdots \rightarrow_{\cong} \sigma_m$ in $G/_{\cong}$ with $\sigma_o \in \sigma^I/_{\cong}$ and $\sigma_m \in \sigma^T/_{\cong}$. Since σ^T is a block of $\cong$, we know that $\sigma_m \subseteq \sigma^T$. Choose a state s_0 in the intersection $\sigma^I \cap \sigma_0$. Since $\cong$ is stable, we know that whenever $\tau \rightarrow_{\cong} \upsilon$, for all state $s \in \tau$, there exists a state $t \in \upsilon$ such that $s \rightarrow t$. Starting with $s_0 \in \sigma_0$, choose states $s_1, \ldots s_m$, one by one, such that, for every $1 \leq i \leq m$, $s_i \in \sigma_i$, and $s_{i-1} \rightarrow s_i$. Since s_m is in the target region σ^T, the trajectory $\overline{s}_{0\ldots m}$ is a witness to the reachability question (G, σ^T). ■

Exercise 4.1 {T2} [Inverse stability] The G-partition $\cong$ is *initialized* if the initial region σ^I is a block of $\cong$. The G-partition $\cong$ is *backstable* if for all states s, s', and t, if $s \cong t$ and $s' \rightarrow s$, then there is a state t' such that $s' \cong t'$ and $t' \rightarrow t$. Equivalently, $\cong$ is a backstable G-partition iff $\cong$ is a stable G^{-1}-partition.

Let G be a transition graph, let $\cong$ be an initialized backstable G-partition, and let σ^T be a region of G. Prove that the two reachability problems (G, σ^T) and $(G/_{\cong}, \sigma^T/_{\cong})$ have the same answer. ■

Projecting states of a module to a subset of variables gives a partition of the underlying transition graph. Let P be a module, and let X be a subset of its variables. For two states s and t of P, let $s \cong_{[X]} t$ if $X[s] = X[t]$. The equivalence $\cong_{[X]}$ is a G_P-partition.

Example 4.3 [Latched variables] Recall the definition of latched variables $latchX_P$ of a module P. The G_P-partition $\cong_{[latchX_P]}$ is a stable partition, and the reduced transition graph G_P^L is the resulting quotient graph. ■

Thus, projection to latched variables results in a stable partition. An orthogonal method to obtain a stable partition is to find a set of variables that is closed under dependencies.

> STABLE VARIABLE SETS
>
> Let P be a module. A subset $X \subseteq X_P$ of the module variables is *stable* if for every variable $x \in X$, if the variable x is controlled by the atom U of P then both $\mathsf{read}X_U \subseteq X$ and $\mathsf{await}X_U \subseteq X$.

Proposition 4.1 [Stable projections] *Let P be a module, and let X be a stable subset of its variables. Then, the G_P-partition $\cong_{[X]}$ is a stable partition.*

Exercise 4.2 {T2} [Elimination of redundant variables] Consider the invariant verification problem (P, p). Let X be a stable set of the module variables. Show that if the observation predicate p refers only to the variables in $X \cap \mathsf{obs}X_P$, then $[\![\neg p]\!]$ is a block of the G_P-partition $\cong_{[X]}$. Then, the invariant verification problem (P, p) reduces to the reachability problem $(G_P/_{\cong_{[X]}}, [\![\neg p]\!])$.

Give an algorithm to compute the minimal set X_p of variables that contains all the variables in p and is stable. Notice that the set X_p contains all the variables whose initialization and update influences the initialization and update of the variables in p, and thus, the remaining variables $X_P \setminus X_p$ are redundant for the verification of p. ■

4.1.2 Graph Symmetries

Stable quotients often arise from exploiting the symmetries of a transition graph. For instance, in the module *Pete* the individual processes are symmetric, resulting in the symmetry in the state-space of G_{Pete}. To formalize the reduction afforeded by symmetries, we beign by defining graph automorphisms. A graph automorphism is a one-to-one onto mapping from vertices to vertices that preserves the initial region as well as the transitions.

> GRAPH AUTOMORPHISM
>
> Consider two transition graphs $G_1 = (\Sigma_1, \sigma_1^I, \rightarrow_1)$ and $G_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2)$. A bijection f from Σ_1 to Σ_2 is an *isomorphism* from G_1 to G_2 if (1) $f(\sigma_1^I) = \sigma_2^I$ and (2) for all states $s, t \in \Sigma_1$, $s \rightarrow_1 t$ iff $f(s) \rightarrow_2 f(t)$. An isomorphism from G to G is called a *G-automorphism.*

Remark 4.1 [Graph Automorphisms] For every transition graph G, the identity function is a G-automorphism. If f is a G-automorphism, then so is its inverse f^{-1}. The (functional) composition of two G-automorphisms is a G-automorphism. The set of all binary functions over the state-space of G forms a group under functional composition. The set of G-automorphisms forms a subgroup. Furthermore, for a set F of G-automorphisms, the subgroup generated by F contains only G-automorphisms. ■

The group of G-automorphisms under functional composition is called the *symmetry group* of G. This symmetry group, or any of its subgroups, can be used to define a stable partition. Given a set F of generators that are G-automorphisms, we obtain the corresponding symmetric partition by considering all the automorphisms in the subgroup generated by F.

> Symmetric Partition
>
> Let G be a transition graph, and let F be a set of G-automorphisms. The *F-symmetric partition* $\cong^F$ is defined by: for all states s and t of G, let $s \cong^F t$ if there is an automorphism $f \in \mathit{closure}(F)$ such that $t = f(s)$.

Stability of the symmetric partition follows immediately from the definitions.

Theorem 4.2 [Symmetric partitioning] *Let G be a transition graph, and let F be a set of G-automorphisms. The induced G-partition $\cong^F$ is stable.*

Let G be a transition graph, let σ be a region of G, and let F be a set of G-automorphisms. If σ is a block of the induced G-partition $\cong^F$, then the quotient $G/_{\cong^F}$ can be used to solve the reachability problem (G, σ). Notice that σ is a block of $\cong^F$ iff for all G-automorphisms $f \in F$, $f(\sigma) = \sigma$.

Example 4.4 [Symmetry of mutual exclusion] Recall Peterson's mutual-exclusion protocol from Chapter 1. Consider the following bijection f on the state space Σ_{Pete} of the underlying transition graph G_{Pete}: let $t = f(s)$ iff

$x_1[t] = x_2[s]$ and $x_2[t] \neq x_1[s]$, and
$pc_1[t] = pc_2[s]$ and $pc_2[t] = pc_1[s]$.

The function f swaps the values of pc_1 and pc_2, swaps the values of x_1 and x_2, and toggles x_2. Note that the thruth of the condition $x_1 = x_2$ is toggled by the function f.

Verify that the function f is a G_{Pete}-automorphism. The composition $f \circ f$ simply toggles both x_1 and x_2: $t = f \circ f(s)$ iff

$x_1[t] \neq x_1[s]$ and $x_2[t] \neq x_2[s]$, and
$pc_1[t] = pc_1[s]$ and $pc_2[t] = pc_2[s]$.

The composition $f \circ f \circ f$ is the inverse of f: $t = f \circ f \circ f(s)$ iff

$x_1[t] \neq x_2[s]$ and $x_2[t] = x_1[s]$, and
$pc_1[t] = pc_2[s]$ and $pc_2[t] = pc_1[s]$.

It follows that f^4 equals the identity map. Consequently, the subgroup *closure*(f) generated by the automorphism f equals $\{f, f \circ f, id, f^{-1}\}$, where *id* the identity function. Consider the four initial states—s_1, s_2, s_3, and s_4—of *Pete*

$$\begin{array}{l}
s_1(pc_1) = outC,\ s_1(x_1) = true,\ \ s_1(pc_2) = outC,\ s_1(x_2) = true; \\
s_2(pc_1) = outC,\ s_2(x_1) = true,\ \ s_2(pc_2) = outC,\ s_2(x_2) = false; \\
s_3(pc_1) = outC,\ s_3(x_1) = false,\ s_3(pc_2) = outC,\ s_3(x_2) = true; \\
s_4(pc_1) = outC,\ s_4(x_1) = false,\ s_4(pc_2) = outC,\ s_4(x_2) = false.
\end{array}$$

Verify that $s_2 = f(s_1)$, $s_3 = f(s_2)$, $s_4 = f(s_3)$, and $s_1 = f(s_4)$. In the partition $\cong^f$, two states are equivalent if one can be obtained from the other by applying one of the automorphisms in *closure*(f). In particular, all the four initial states are equivalent. Verify that while G_{Pete} contains 36 states, the partition $\cong^f$ contains 9 classes; while the reachable subgraph of G_{Pete} contains 20 states, the number of reachable classes of $\cong^F$ is 5.

The region $[\![\neg(pc_1 = inC \wedge pc_2 = inC)]\!]$ is invariant under the function f, and hence, is a block of the stable partition $\cong^f$. It follows that the quotient $G_{Pete}/_{\cong^f}$ can be used to check that the protocol *Pete* enforces mutual exclusion. ■

In practice, the communication topology among different components yields graph automorphisms. Two typical examples are:

- *Star Topology:* The system consists of a module P (server) communicating with modules $P_1, \ldots P_n$ (clients). The client modules $P_1, \ldots P_n$ are renamed copies of each other, and thus, there is a one-to-one correspondence between the controlled variables of two client modules. Two client modules do not have any common variables, and thus, each client module communicates only with the server. In this situation, swapping the values of the controlled variables of two client modules results in an automorphism. In particular, the set F of generators contains for every pair $1 \leq i, j \leq n$, the automorphism f_{ij} that swaps the values of the controlled variables of P_i with the values of the corresponding controlled variables of P_j.

- *Ring Topology:* The system consists of modules $P_1, \ldots P_n$ connected in a ring, that is, every module P_i communicates only with its neighboring modules P_{i-1} and P_{i+1} (where increments and decrements are modulo n). All the modules are renamed copies of each other. In this situation, every rotation of the indices yields an automorphism. That is, for every i, the function f_i is an automorphism, where $t = f_i[s]$ if the values of the controlled variables of the module P_j in state t equal the values of the corresponding controlled variables of the module P_{j+i} in state s.

Exercise 4.3 {`T2`} [Symmetry in Railroad controller] Consider the module *RailroadSystem* from Chapter 2. Find a suitable set F of automorphisms. What is the equivalence $\cong^F$ induced on the state-space? ■

To apply symmetric reduction to the invariant verification problem (P, p), we first find a suitable set of G_P-automorphisms. The next step is to find a mapping *rep* that maps every state s to a unique representative of the equivalence class of $\cong^F$ that contains s: if $s \cong^F t$, then $\mathit{rep}(s) = \mathit{rep}(t)$. If we have such a function *rep*, then the depth-first search algorithm is modified so that only the representative states are explored. This is achieved by replacing the initial region σ^I by the set $\mathit{rep}(\sigma^I)$ of representative initial states, and replacing the successor function *post* by the function $\mathit{rep} \circ \mathit{post}$ that considers only representative states. Consequently, the complexity of the search is proportional the size of the quotient graph with respect to $\cong^F$.

Exercise 4.4 {`T3`} [Representative states in mutual exclusion] Consider the automorphism f, and the induced equivalence $\cong^f$, on the state-space of the module *Pete* considered in Example 4.4. Suggest a suitable set of representative states and the function *rep* that maps each state to its representative. ■

4.2 Partition Refinement

Suppose we wish to solve multiple verification problems involving a transition graph G. Then, it is prudent to find a stable G-partition $\cong$ such that there are as few $\cong$-equivalence classes as possible. Then the quotient $G/_{\cong}$ can be used to solve the verification problems concerning G.

4.2.1 The Structure of Stable Partitions

If $\cong_1$ and $\cong_2$ are stable partitions of a transition graph, then so is their join:

Lemma 4.1 [Union-closure of stable partitions] *Let G be a transition graph. If E is a set of stable G-partitions, then the join $\bigcup^* E$ is a stable G-partition.*

Proof. Let G be a transition graph, let E is a set of stable G-partitions, and let $\cong$ be the join $\bigcup^* E$ of all partitions in E. Suppose $s \cong t$ and $s \rightarrow s'$. Since $\cong$ is the transitive closure of the union of the equivalence relations in E, there are states $s_0, \ldots s_n$ and partitions $\cong_1, \cong_n$ in E such that $s_0 = s$, $s_n = t$, and $s_{i-1} \cong_i s_i$ for $1 \leq i \leq n$. Let $s'_0 = s'$. We have $s_0 \rightarrow s'_0$. Since each partition $\cong_i$ is stable, by induction on i, there exist states $s'_1, \ldots s'_n$ such that for $1 \leq i \leq n$, $s_i \rightarrow s'_i$ and $s'_{i-1} \cong_i s'_i$. Choose $t' = s'_n$. We have $t \rightarrow t'$ and $s' \cong t'$. ■

Corollary 4.1 [CPO of stable partitions] *For every transition graph G, the refinement relation $\preceq$ is a complete lattice on the stable G-partitions.*

Exercise 4.5 {T2} [Complete lattice of stable partitions] Consider the complete lattice $\preceq$ on the stable G-partitions of the transition graph G. Let E be a set of stable G-partitions. The least upper $\preceq$-bound for E is the join $\bigcup^* E$. What is the greatest lower $\preceq$-bound for E? ■

Let $\cong$ be a partition of the transition graph G. Consider the set E of all stable partitions that refine $\cong$. The join $\bigcup^* E$, which is guaranteed to exist, is a stable partition. Furthermore, since every partition in E refines $\cong$, so does $\bigcup^* E$. Consequently, the join $\bigcup^* E$ is the coarsest partition that is both stable and is finer than $\cong$.

Let G be a transition graph, and let $\cong$ be a G-partition. The *coarsest stable refinement* of $\cong$, denoted $\min_G(\cong)$, is the join of all stable G-partitions that refine $\cong$. The quotient $G/_{\min(\cong)}$ is called $\cong$-*minimal*.

It follows that $\min(\cong)$ is a stable G-partition that refines $\cong$, and that all stable G-partitions that refine $\cong$ also refine $\min(\cong)$.

Remark 4.2 [Refinement of identity and universal partitions] If $\cong$ is the identity partition (i.e. all equivalence classes of $\cong$ are singletons), then $\min(\cong)$ equals $\cong$. If G is a serial transition graph, and $\cong$ is the universal partition (i.e. contains a single equivalence class containing all states), then $\min(\cong)$ equals $\cong$. ■

The partition-refinement problem

An instance $(G, \cong^I)$ of the *partition-refinement problem* consists of (1) a transition graph G and (2) [the *initial partition*] a G-partition $\cong^I$. The answer to the partition-refinement problem $(G, \cong^I)$ is the coarsest stable refinement $\min(\cong^I)$ of the initial partition $\cong^I$.

Example 4.5 [Coarsest stable refinement] Consider the transition graph of Figure 4.1. Suppose the initial partition $\cong^I$ contains two regions; $\{s_0, s_1, t_0, t_1\}$ and $\{u_0, u_1\}$. The initial partition itself is not stable. Its coarsest stable refinement contains three regions $\{s_0, s_1\}$, $\{t_0, t_1\}$, and $\{u_0, u_1\}$. ■

Minimal reachability-preserving quotients

Let G be a transition graph with the state space Σ. For a region σ of G, let $\cong^\sigma$ denote the binary G-partition $\{\sigma, \Sigma \backslash \sigma\}$. The partition $\cong^\sigma$ is the coarsest partition that has σ as a block. The $\cong^\sigma$-minimal quotient $G/_{\min(\cong^\sigma)}$ of the transition graph G can be used to solve the reachability problem (G, σ), because $\min(\cong^\sigma)$ is stable and σ is a block of $\min(\cong^\sigma)$. For a set R of regions, let $\cong^R$ denote the G-partition $(\cap \sigma \in R.\ \cong^\sigma)$. The $\cong^R$-minimal quotient $G/_{\min(\cong^R)}$

can then be used to solve all reachability problems of the form (G, σ) for $\sigma \in R$. For example, let P be a module. Copnsider the equivalence $\cong$ on the state-space of P induced by the observations: $s \cong t$ iff $\mathsf{obs}X_P[s] = \mathsf{obs}X_P[t]$. Then, two $\cong$-equivalent states satisfy the same set of observation predicates. The quotient $G_P/_{\min(\cong)}$ can then be used to solve all invariant-verification problems for the module P.

Exercise 4.6 {`T3`} [Reachable portion of minimal quotients] Let G be a transition graph, and let σ be a region of G. To solve the reachability problem (G, σ), it suffices to consider the reachable region σ^R of G. We may first find a minimal quotient of G and then construct the reachable subquotient, or we may first construct the reachable subgraph of G and then find a minimal quotient. Both methods lead to isomorphic results. Let G_1 be the reachable subgraph of the $\cong^\sigma$-minimal quotient $G/_{\min(\cong^\sigma)}$ of G. For $\cong = \{\sigma^R \cap \sigma, \sigma^R \backslash \sigma\}$, let G_2 be the $\cong$-minimal quotient $G^R/_{\min(\cong)}$ of the reachable subgraph G^R of G. Prove that the two transition graphs G_1 and G_2 are isomorphic. ■

Exercise 4.7 {`T4`} [Inverse minimal quotients] Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph, and let $\cong$ be a G-partition. The *coarsest backstable refinement* of $\cong$, denoted $\min^{-1}(\cong)$, is the join of all backstable G-partitions that refine $\cong$. The quotient $G/_{\min^{-1}(\cong^{\sigma^I})}$ of the transition graph G can be used to solve the reachability problem (G, σ), for any region σ of G. Prove that the unreachable region $\Sigma \backslash \sigma^R$ is a $\min^{-1}(\cong^{\sigma^I})$-equivalence class.

Let σ be a region of G. To solve the reachability problem (G, σ), we may compute (the reachable portion of) a stable refinement of $\cong^\sigma$, or a backstable refinement of $\cong^{\sigma^I}$. Depending on the given reachability problem, either method may be superior to the other method. Consider two state spaces: (A) the quotient $\sigma^R/_{\min(\cong^\sigma)}$ of the reachable region σ^R with respect the coarsest stable refinement $\min(\cong^\sigma)$; (B) the coarsest backstable refinement $\min^{-1}(\cong^{\sigma^I})$. Give an example of a reachability problem for which state space (A) is finite and state space (B) is infinite, and an example for which state space (A) is infinite and state space (B) is finite. ■

Exercise 4.8 {`T3`} [Symbolic reachability versus partition refinement] Let G be a transition graph, and let σ be a region of G. Prove that for all natural numbers i, the region $pre^i(\sigma)$ is a block of the coarsest stable refinement $\min(\cong^\sigma)$, and the region $post^i(\sigma^I)$ is a block of the coarsest backstable refinement $\min^{-1}(\cong^{\sigma^I})$. Conclude that if the coarsest backstable refinement $\min^{-1}(\cong^{\sigma^I})$ is finite, then the transition graph G is finitely reaching. ■

4.2.2 Partition-refinement Algorithms

We first develop a schematic algorithm for solving the partition-refinement problem, and prove it correct. For a running-time analysis, we then present several concrete instantiations of the schematic partition-refinement algorithm.

A region characterization of stability

Let G be a transition graph, and let σ and τ be two regions of G. The region σ is *stable with respect to* the region τ if either $\sigma \subseteq \mathit{pre}(\tau)$ or $\sigma \cap \mathit{pre}(\tau) = \emptyset$. Let $\cong$ be a G-partition. The partition $\cong$ is *stable with respect to* the region τ if all $\cong$-equivalence classes are stable with respect to τ. This region-based definition gives an alternative characterization of stability.

Lemma 4.2 [Stability with respect to regions] *Let G be a transition graph, and let $\cong$ be a G-partition. Then $\cong$ is stable iff $\cong$ is stable with respect to all $\cong$-equivalence classes.*

Stabilization of a partition with respect to a region

Partition-refinement algorithms stabilize the given initial partition by repeatedly splitting equivalence classes. Consider a partition $\cong$. If $\cong$ is not stable, then, by Lemma 4.2, there are two equivalence classes σ and τ of $\cong$ such that σ is not stable with respect to τ. In such a case, we can split σ into two regions, one that contains states which have successors in τ and the other one that contains states with no successors in τ. That is, we split σ at the boundary of the predecessor region of τ.

Let τ be a region of the transition graph G. For a region σ of G, let

$$\mathit{Split}(\sigma,\tau) \;=\; \begin{cases} \{\sigma\} & \text{if } \sigma \subseteq \tau \text{ or } \sigma \cap \tau = \emptyset, \\ \{\sigma \cap \tau, \sigma \backslash \tau\} & \text{else,} \end{cases}$$

be the result of splitting σ at the boundary of τ. For a G-partition $\cong$, let

$$\mathit{Split}(\cong,\tau) \;=\; (\cup \sigma \in \cong .\, \mathit{Split}(\sigma,\tau))$$

be the result of splitting $\cong$ at the boundary of τ. The result $\mathit{Split}(\cong,\tau)$ is a G-partition that refines $\cong$ and contains at most twice as many equivalence classes as $\cong$. To stabilize $\cong$ with respect to τ, we split $\cong$ at the boundary of $\mathit{pre}(\tau)$:

$$\mathit{Stabilize}(\cong,\tau) \;=\; \mathit{Split}(\cong,\mathit{pre}(\tau)).$$

The stablization of a region σ with respect to τ is depicted pictorially in Figure 4.2. The *Stabilize* operation can be implemented either symbolically or enumeratively.

Symbolic stabilization. Suppose that the region τ is given by a symbolic region representation $\{\tau\}_s$, and the partition $\cong$ is given by a list $\langle \{\sigma\}_s \mid \sigma \in \cong \rangle$

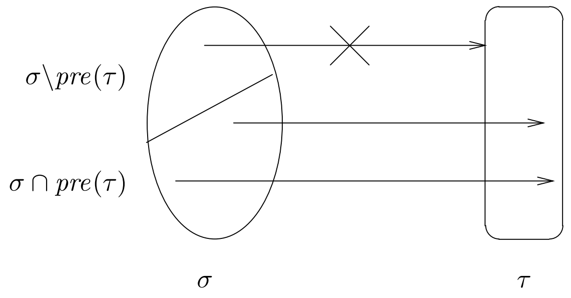


Figure 4.2: Stabilizing one region with respect to another

of symbolic region representations. The operation *Stabilize*$(\cong, \tau)$ can then be performed using boolean operations, emptiness checking, and the *pre* operation on symbolic region representations.

Enumerative stabilization. We are given an abstract data type **partition** that maintains a collection of nonempty, disjoint subsets of the state-space. The data type **partition** is like **set of region**, but supports the following additional operations:

Find: For a state s and a partition $\cong$, the operation *Find*$(s, \cong)$ returns the (name of) the region that contains s, if such a region exists; otherwise *Find*$(s, \cong)$ returns *nil*.

Create: For a state s and a partition $\cong$, the operation *Create*$(s, \cong)$ removes s from any existing region in $\cong$, creates a singleton set containing s, and returns the newly created set. Note that *Create*$(s, \cong)$ destructively updates the partition $\cong$.

Move: For a state s, a partition $\cong$, and a set σ in $\cong$, the operation *Move*$(s, \sigma, \cong)$ removes s from any existing set in $\cong$ and adds s to the set σ; if the result of removing s from an existing set results in an empty set, that set is destroyed.

Exercise 4.9 {P2} [Abstract data type **partition**] Implement the abstract data type **partition** so that each of the three operations *Find*, *Create*, and *Move* take constant time. ■

Let G be a finite transition graph. Suppose that the region τ is given by a list $\{\tau\}_e$ of states, and the partition $\cong$ is given using the abstract data type **partition**. Furthermore, with each state s we are given a list of all predecessor states in *pre*(s), and with each set σ in **partition** we are given the name *new*(σ) of another set in **partition**. When stabilizing the region σ with respect to τ, the states in $\sigma \cap pre(\tau)$ are moved to the set $new(\sigma)$. Initially, all *new* pointers are *nil*, and they are reset after stabilization. The operation $\mathit{Stabilize}(\cong, \tau)$ can then be performed as follows:

```
foreach $s \in \tau$ do foreach $t \in pre(s)$ do $Update(t, \cong)$ od od;
foreach $s \in \tau$ do foreach $t \in pre(s)$ do $Reset(t, \cong)$ od od,
```

where both

```
$Update(t, \cong)$:
  if $new(Find(t, \cong)) = nil$
    then $new(Find(t, \cong)) := Create(t, \cong)$
    else $Move(t, new(Find(t, \cong)), \cong)$
    fi
```

and

```
$Reset(t, \cong)$:
  $new(Find(t, \cong)) := nil$
```

take constant time. Let n_τ be the number of states in the region τ, and let m_τ be the number of transitions whose target lies in τ. The time required by the operation $\mathit{Stabilize}(\cong, \tau)$ is $\mathit{stabcost}(\tau) = O(m_\tau + n_\tau)$. We charge the stabilization cost $\mathit{stabcost}(\tau)$ to the individual states in τ. If m_s is the number of transitions with target s (i.e. $m_s = |pre(s)|$), then we charge $\mathit{stabcost}(s) = O(m_s + 1)$ to each state $s \in \tau$. Then $\mathit{stabcost}(\tau) = (+s \in \tau.\, \mathit{stabcost}(s))$.

Iterative stabilization of a partition

The key properties of the operation of stabilizing a partition with respect to a region are summarized in the next lemma.

Lemma 4.3 [Stabilization for partition refinement] *Let G be a transition graph, let $\cong$ be a G-partition, and let τ be a region of G. (1) If τ is a block of $\cong$, then* $\min(\cong) \preceq \mathit{Stabilize}(\cong, \tau)$. *(2) Every G-partition that refines $\mathit{Stabilize}(\cong, \tau)$ is stable with respect to τ.*

Exercise 4.10 {T2} [Stabilization for partition refinement] Prove Lemma 4.3. ■

Lemma 4.3 suggests a partition-refinement algorithm that, starting from the given initial partition, repeatedly stabilizes the partition with respect to one of its blocks. Part (1) ensures that stabilization with respect to a block never causes unnecessary splitting. Part (2) ensures that every block needs to be considered for stabilization at most once. The resulting scheme is shown in Figure 4.3.

In Algorithm 4.1, at the beginning of each execution of the while-loop, we know that

Algorithm 4.1 [Schematic Partition Refinement]

Input: a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$ and an initial G-partition $\cong^I$.
Output: the coarsest stable refinement $\min(\cong^I)$.
Local: a G-partition $\cong$ and a region set *done*.

```
$\cong := \cong^I$; $done := \{\Sigma\}$;
while $\cong \not\subseteq done$ do
  {assert $\min(\cong^I)$ refines $\cong$, and $\cong$ is stable w.r.t. all regions in $done$}
  Choose a block $\tau$ of $\cong$ such that $\tau \notin done$;
  $\cong := Stabilize(\cong, \tau)$;
  $done := Insert(\tau, done)$
  od;
return $\cong$.
```

Figure 4.3: Partition refinement

1. the coarsest stable refinement $\min(\cong^I)$ is a refinement of the current partition $\cong$,
2. every region in the set *done* is a block of the current partition $\cong$, and
3. the current partition $\cong$ is stable with respect to every region in *done*.

Algorithm 4.1 terminates iff $\min(\cong^I)$ has finitely many equivalence classes. Suppose that $\min(\cong^I)$ has n equivalence classes and, therefore, 2^n blocks. With every iteration of the while-loop, a block of $\min(\cong^I)$ is added to the set *done*. It follows that the while-loop is executed at most 2^n times.

Theorem 4.3 [Schematic partition refinement] *Let G be a transition graph, and let $\cong^I$ be a G-partition. If the coarsest stable refinement* $\min(\cong^I)$ *is finite, then Algorithm 4.1 solves the partition-refinement problem* $(G, \cong^I)$.

A quadratic partition-refinement algorithm

If we carefully choose the region τ in each iteration of Algorithm 4.1, we obtain polynomial-time implementations. A quadratic running time is achieved if, during consecutive iterations, we systematically stabilize the initial partition $\cong^I$ first with respect to all $\cong^I$-equivalence classes, then with respect to all equivalence classes of the resulting partition, etc. The resulting algorithm is shown in Figure 4.4.

Observe that during the execution of the for-loop, every equivalence-class of $\cong^{prev}$ is a block of the current partition $\cong$, and at the beginning of the while-loop the current partition $\cong$ is stable with respect to every region in $\cong^{prev}$. Thus,

Algorithm 4.2 [Quadratic Partition Refinement]

Input: a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$ and an initial G-partition $\cong^I$.
Output: the coarsest stable refinement $\min(\cong^I)$.
Local: two G-partitions $\cong$ and $\cong^{prev}$.

```
$\cong := \cong^I$; $\cong^{prev} := \{\Sigma\}$;
**while** $\cong \neq \cong^{prev}$ **do**
  {**assert** $\min(\cong^I)$ refines $\cong$, and $\cong$ is stable w.r.t. all regions in $\cong^{prev}$}
  $\cong^{prev} := \cong$;
  **for** each $\tau \in \cong^{prev}$ **do** $\cong := \mathit{Stabilize}(\cong, \tau)$ **od**
  **od**;
**return** $\cong$.
```

Figure 4.4: Quadratic algorithm for partition refinement

Algorithm 4.2 is an instance of Algorithm 4.1, and its correctness follows immediately. With every iteration of the while-loop, the number of $\cong$-equivalence classes increases. Hence, if $\min(\cong^I)$ has n equivalence classes, the while-loop is executed at most n times. The for-loop can be implemented either symbolically or enumeratively. Consider an enumerative implementation of Algorithm 4.2 for an input graph G with n states and $m \geq n$ transitions. Then the coarsest stable refinement $\min(\cong^I)$ has at most n equivalence classes, and the time required by the for-loop is

$$\begin{aligned}(+\tau \in \cong^{prev} .\, \mathit{stabcost}(\tau)) &= (+s \in \Sigma.\, \mathit{stabcost}(s)) \\ &= (+s \in \Sigma.\, O(m_s + 1)) \\ &= O(m).\end{aligned}$$

Theorem 4.4 [Quadratic partition refinement] *Let G be a finite transition graph with n states and m transitions. The running time of Algorithm 4.2 on input G is $O(m \cdot n)$.*

Exercise 4.11 {`P3`} [Quadratic partition refinement] Write a program that implements Algorithm 4.2 symbolically, and a program that implements Algorithm 4.2 enumeratively. For your symbolic program, assume that the input graph G is given by a symbolic graph representation $\{G\}_s$, and the input partition $\cong^I$ is given by a list $\langle \{\sigma\}_s \mid \sigma \in \cong^I \rangle$ of symbolic region representations. For your enumerative program, assume that the input graph G is given by an enumerative graph representation $\{G\}_e$, and the input partition $\cong^I$ is given by a list $\langle \{\sigma\}_e \mid \sigma \in \cong^I \rangle$ of enumerative region representations. The asymptotic running time of your enumerative program should be quadratic in the size of the input. ■

Algorithm 4.3 [Paige-Tarjan Partition Refinement]

Input: a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$, and an initial G-partition $\cong^I$.
Output: the coarsest stable refinement $\min(\cong^I)$.
Local: two G-partitions $\cong$ and $\cong^{done}$.

```
$\cong := \cong^I$; $\cong^{done} := \{\Sigma\}$;
while $\cong \subset \cong^{done}$ do
  {assert $\min(\cong^I) \preceq \cong$, and $\cong$ is stable w.r.t. all regions in $\cong^{done}$}
  Choose $\sigma \in (\cong^{done} \setminus \cong)$;
  Choose $\tau \in \cong$ such that $\tau \subseteq \sigma$ and $|\tau| \leq |\sigma|/2$;
  $\cong := \mathit{Stabilize}(\mathit{Stabilize}(\cong, \tau), \sigma \backslash \tau)$;
  $\cong^{done} := \mathit{Insert}(\sigma \backslash \tau, \mathit{Insert}(\tau, \mathit{Delete}(\sigma, \cong^{done})))$
  od;
return $\cong$.
```

Figure 4.5: Paige-Tarjan algorithm for partition refinement

The Paige-Tarjan partition-refinement algorithm

To improve the time complexity of partition refinement, we need an improved strategy to choose the splitting block. The number of stabilization operations required can equal the number of equivalence-classes in the coarsest stable refinement, which can, in turn, be equal to the number of states in the transition graph, in the worst case.

Exercise 4.12 {T3} [Worst-case for quadratic partition refinement] Give an instance $(G, \cong^I)$ of the partition refinement problem such that the execution of Algorithm 4.1 on this instance, requires n iterations of the while-loop, irrespective of the choices of the splitting blocks τ. ■

If, at each iteration of Algorithm 4.1, we carefully choose a "small" block τ of $\cong$ for the operation $\mathit{Stabilize}(\cong, \tau)$, we arrive at a subquadratic running time. A suitable criterion for "small" is that τ is a $\cong$-equivalence class that contains at most half the states of any $\cong$-block σ if $\cong$ is known to be stable with respect to σ. This criterion is enforced by maintaining a second partition, $\cong^{done}$, such that $\cong$ refines $\cong^{done}$ and is stable with respect to all $\cong^{done}$-equivalence classes. The algorithm is shown in Figure 4.5. Observe that Algorithm 4.3 is an instance of Algorithm 4.1.

Consider an enumerative implementation of Algorithm 4.3 for an input graph G with n states and $m \geq n$ transitions. Since the number of $\cong^{done}$-equivalence classes increases with every iteration, the while-loop is executed at most n times. Let σ_i and τ_i denote the equivalence classes of $\cong^{done}$ and $\cong$, respectively, that are chosen in the i-th iteration of the while-loop. An appropriate choice of τ_i can be performed by maintaining for each $\cong$-equivalence class υ a counter that indicates the number of states in υ. Suppose that a state $s \in \Sigma$ belongs to both τ_i and τ_j, for $j > i$. Since $\sigma_j \subseteq \tau_i$ and $|\tau_j| \leq |\sigma_j|/2$, also $|\tau_j| \leq |\tau_i|/2$. It follows that there are at most $\log n + 1$ iterations i such that $s \in \tau_i$.

The i-th iteration of the while-loop consists of two stabilizing operations, one with respect to τ_i and one with respect to $\sigma_i \backslash \tau_i$. Since each state belongs only to $O(\log n)$ many regions τ_i, the cumulative cost of the stabilization operations with respect to all regions τ_i is $(+s \in \Sigma.\, O(\log n) \cdot \mathit{stabcost}(s)) = O(m \cdot \log n)$. A state, however, may belong to $O(n)$ many regions of the form $\sigma_i \backslash \tau_i$. The following lemma states that to stabilize $\cong$ with respect to $\sigma_i \backslash \tau_i$, instead of splitting $\cong$ with respect to $\mathit{pre}(\sigma_i \backslash \tau_i)$, we can split it with respect to $\mathit{pre}(\tau_i) \backslash \mathit{pre}(\sigma_i \backslash \tau_i)$, thereby, avoiding the computation of $\mathit{pre}(\sigma_i \backslash \tau_i)$. This observation allows us to implement the operation $\mathit{Stabilize}(\cong, \sigma_i \backslash \tau_i)$ in time $\mathit{stabcost}(\tau_i)$, that is, at the same cost as the operation $\mathit{Stabilize}(\cong, \tau_i)$.

Lemma 4.4 [Efficient stabilization for Paige-Tarjan] *Let G be a transition graph, let $\cong$ be a G-partition, and let σ and τ be two blocks of $\cong$. If $\cong$ is stable with respect to σ and τ, then*

$$\mathit{Stabilize}(\cong, \sigma \backslash \tau) \;=\; \mathit{Split}(\cong, \mathit{pre}(\tau) \backslash \mathit{pre}(\sigma \backslash \tau)).$$

Exercise 4.13 {T3} [Efficient stabilization for Paige-Tarjan] Prove Lemma 4.4. ■

For every state $s \in \Sigma$ and every $\cong^{done}$-equivalence class σ, we maintain a counter $\mathit{tcount}(s, \sigma)$ that indicates the number of transitions from s to a state in σ; that is, $\mathit{tcount}(s, \sigma) = |\sigma \cap \mathit{post}(s)|$. The operation $\mathit{Stabilize}(\cong, \sigma \backslash \tau)$ can then be performed in time $\mathit{stabcost}(\tau) = O(m_\tau + n_\tau)$:

```
foreach $s \in \tau$ do
  foreach $t \in pre(s)$ do $tcount(t, \tau) := 0$ od
  od;
foreach $s \in \tau$ do
  foreach $t \in pre(s)$ do $tcount(t, \tau) := tcount(t, \tau) + 1$ od
  od;
foreach $s \in \tau$ do
  foreach $t \in pre(s)$ do
    $tcount(t, \sigma \backslash \tau) := tcount(t, \sigma) - tcount(t, \tau)$;
    if $tcount(t, \sigma \backslash \tau) = 0$ then $Update(t, \cong)$ fi
    od
  od;
foreach $s \in \tau$ do foreach $t \in pre(s)$ do $Reset(t, \cong)$ od od.
```

If we charge the cost of both parts of the operation $\mathit{Stabilize}(\mathit{Stabilize}(\cong,\tau),\sigma\backslash\tau)$ to the states in τ, it follows that the time required by Algorithm 4.3 is

$$(+s \in \Sigma.\, O(\log n)\cdot 2\cdot \mathit{stabcost}(s)) \;=\; O(m\cdot\log n).$$

Theorem 4.5 [Paige-Tarjan partition refinement] *Let G be a finite transition graph with n states and m transitions. The running time of Algorithm 4.3 on input G is $O(m\cdot\log n)$.*

Exercise 4.14 {P2} [Mutual exclusion] Recall Peterson's mutual-exclusion protocol from Chapter 1. In the initial partition $\cong^I$, two states are equivalent iff they agree on all the observation predicates: $s \cong^I t$ iff $pc_1[s] = pc_1[t]$ and $pc_2[s] = pc_2[t]$. Construct the $\cong^I$-minimal quotient of G_{Pete} using first Algorithm 4.2 and then Algorithm 4.3. In both cases, show the intermediate results after each iteration of the while-loop. ■

4.3 Reachable Partition Refinement*

Consider a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$ and an initial partition $\cong^I$. The $\cong^I$-minimal quotient is the graph $G/_{\min(\cong^I)}$ with state space $\Sigma/_{\min(\cong^I)}$ and initial states $\sigma^I/_{\min(\cong^I)}$. For verification, we need to compute only the reachable states of the $\cong^I$-minimal quotient. This suggests reformulating the partition refinement problem to account for reachability.

> Minimal reachable quotient
>
> Let G be a transition graph and let $\cong$ be a G-partition. The *reachable stable partition* of $\cong$, denoted $\min^R(\cong)$, is the reachable region of the $\cong$-minimal quotient $G/_{\min(\cong)}$. The reachable subgraph of $G/_{\min(\cong)}$ is called the $\cong$*-minimal-reachable quotient.*

Remark 4.3 [Minimal reachable quotient] Let G be a transition graph with states Σ and reachable region σ^R. Let $\cong$ be a G-partition. The region σ^R need not be a block of $\min(\cong)$. The reachable stable partition $\min^R(\cong)$ is a partitioning of some region σ of G such that $\sigma^R \subseteq \sigma \subseteq \Sigma$. Thus, $\min^R(\cong)$ is not necessarily a G-partition, nor a refinement of $\cong$. A region τ in $\min^R(\cong)$ is contained in some $\cong$-equivalence class, and is stable with respect to every region in $\min^R(\cong)$. ■

To solve a reachability problem for the transition graphs, it suffices to construct the minimal-reachable quotient with respect to a suitably chosen initial partition.

Proposition 4.2 [Reachability] *Let G be a transition graph, $\cong$ be a G-partition, and σ be a block of $\cong$. Then, the answer to the reachability problem (G,σ) is* Yes *iff $\sigma \cap \tau$ is nonempty for some $\tau \in \min^R(\cong)$.*

Algorithm 4.4 [Minimization with reachability]

```
Input: a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$ and a $G$-partition $\cong^I$.
Output: the answer to the reachable-partition-refinement problem
    $(G, \cong^I)$.
$\cong := \cong^I$;  $\sigma^R := \emptyset$
repeat
  {assert $\min(\cong^I)$ is a refinement of $\cong$ }
  $\cong_R := \{\sigma \in \cong \mid \sigma \cap \sigma^R \neq \emptyset\}$
  {assert $\sigma^R \subseteq post^*(\sigma^I)$, and for $\sigma \in \cong_R$, $|\sigma \cap \sigma^R| = 1$ }
  $U := \{\sigma \in \cong \setminus \cong_R \mid \sigma \cap (\sigma^I \cup post(\sigma^R)) \neq \emptyset\}$
  $V := \{(\tau, \upsilon) \in \cong_R \times \cong \mid \tau$ is unstable with respect to $\upsilon\}$
    Search:                                      or   Split:
    Choose $\sigma \in U$                             Choose $(\tau, \upsilon) \in V$
    Choose $s \in \sigma \cap (\sigma^I \cup post(\sigma^R))$      $\cong := Delete(\tau, \cong)$
    $\sigma^R := Insert(s, \sigma^R)$                  $\cong := \cong \cup Split(\tau, pre(\upsilon))$
  until $U = \emptyset$ and $V = \emptyset$
return $\cong_R$.
```

Figure 4.6: Simultaneous minimization and reachability

Reachable partition refinement

An instance $(G, \cong^I)$ of the *reachable-partition-refinement problem* consists of (1) a transition graph G and (2) [the *initial partition*] a G-partition $\cong^I$. The answer to the reachable-partition-refinement problem $(G, \cong^I)$ is the reachable stable partition $\min^R(\cong)$.

One possible solution to the reachable-partition-refinement problem is to first compute the $\cong^I$-minimal quotient and then analyze reachability. However, there are instances of the problem for which $\min(\cong^I)$ contains large, or even infinite, number of regions, but only a small number of them are reachable. Thus, the problem demands a solution that performs both the stabilization and reachability analysis simultaneously. An alternative strategy is shown in Figure 4.6.

As in the previous partition refinement algorithms, Algorithm 4.4 maintains a current partition $\cong$. The set σ^R contains states reachable from σ^I (at most one state per region of $\cong$). The set $\cong_R$ contains those regions of $\cong$ that are already known to be reachable. The algorithm computes the set U of regions that can be added to $\cong_R$ and the set V of unstable pairs of regions. A region σ belongs to U if it contains an initial state or a successor of a state already known to be reachable. A pair (τ, υ) belongs to the set V if τ is known to be

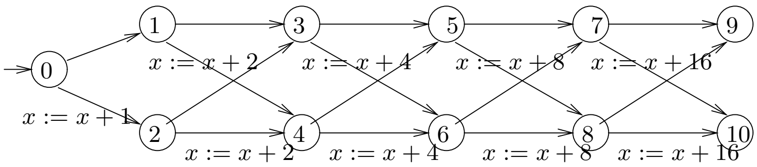


Figure 4.7: Example for computing minimal-reachable quotient

reachable, and is unstable with respect to υ. The algorithm either updates the reachability information for some region in U, or stabilizes some pair (τ, υ) in V by splitting τ. Thus searching is interleaved with stabilization in an arbitrary fashion. Stabilization involves splitting a reachable region τ with respect to $pre(\upsilon)$ for some $\cong$-equivalence class υ. Observe that a region is split only if it is known to be reachable. The algorithm terminates when neither search nor split is enabled. As in partition refinement, the coarsest stable partition $\min(\cong^I)$ is a refinement of the current partition $\cong$. Upon termination, $\cong_R$ is a subset of the coarsest stable partition $\min(\cong^I)$, and contains its reachable states. However, $\cong$ may contain unstable unreachable regions, and thus, need not equal $\min(\cong^I)$.

Theorem 4.6 [Minimization with reachability] *On an instance* $(G, \cong^I)$ *of reachable-partition refinement problem, if Algorithm 4.4 terminates, it outputs the reachable stable partition* $\min^R(\cong^I)$.

The size of σ^R, and hence, the number of regions in $\cong_R$, is nondecreasing, and is bounded by the number of regions in the output $\min^R(\cong^I)$. Every iteration either adds one more state to σ^R, or one more region to the partition $\cong$. It follows that if the coarsest stable refinement $\min(\cong^I)$ has finitely many regions, then Algorithm 4.4 is guaranteed to terminate. The algorithm may terminate even if $\min(\cong^I)$ has infinitely many regions. However, there are cases when $\min^R(\cong^I)$ has finitely many regions, and yet, the algorithm may execute forever. While the output does not depend upon the strategy used to choose between searching and splitting, the final partition $\cong$, and the number of iterations before termination, depend on the strategy.

Exercise 4.15 {P3} [Computing minimal-reachable quotient] Consider a symbolic transition graph with four boolean variables x, y, z, and w. The initial predicate is $x = \mathit{true} \wedge y = \mathit{false}$. The transition predicate is $(w' = x) \wedge (x' = \neg y) \wedge (y' = w \vee z)$. The initial partition contains two regions $[\![x \vee y]\!]$ and $[\![\neg x \wedge \neg y]\!]$. Compute the minimal-reachable quotient by executing Algorithm 4.4. How many regions does the output have? ■

Exercise 4.16 {T2} [Worst-case scenario for computing minimal-reachable quotient] Consider the symbolic transition graph shown in Figure 4.7. The graph has two variables, the location variable pc that ranges over the set $\{0 \ldots 10\}$, and a variable x that ranges over $\{0 \ldots 31\}$. The transitions are as shown. The assignments require that the updated value lies in the range $\{0 \ldots 31\}$ (e.g., the assignment $x := x+1$ stands for the guarded assignment $x < 31 \rightarrow x := x+1$). The initial predicate is $pc = 0 \wedge x = 0$. The initial partition $\cong^I$ contains one region $[\![pc = i]\!]$ per location $0 \leq i \leq 10$. How many regions does a $\cong^I$-minimal-reachable quotient have? Consider an execution of Algorithm 4.4, where splitting is preferred over searching. Show that, upon termination, for every value $0 \leq i \leq 31$ of x, the partition $\cong$ contains the singleton region $pc = 0 \wedge x = i$. ■

Lee-Yannakakis algorithm

The Lee-Yannakakis algorithm for constructing the minimal-reachable quotient modifies Algorithm 4.4 by imposing a deterministic strategy for searching and splitting. The algorithm is shown in Figure 4.8. The type of a graph partition is **partition**, and it supports insertion (*Insert*), deletion (*Delete*), enumeration (**foreach**), and the mapping *Find*.

Each iteration of the outer repeat-loop in Algorithm 4.5 consists of a searching phase, followed by the splitting phase. Search is performed until no more regions can be found reachable, thus, search has a priority over splitting.

As in Algorithm 4.4 the set σ^R contains reachable states, at most one per region of $\cong$. In the searching phase, the algorithm constructs the reachable regions $\cong_R$ by exploring the successors of states in σ^R. The set E contains the edges between the reachable regions. The search is performed in a depth first manner using the stack U.

The computation of the algorithm can be understood from the illustration of Figure 4.9. The partition contains 7 regions $\sigma_0, \ldots \sigma_6$. The regions σ_0, σ_1, σ_4 and σ_5 are found to be reachable in the searching phase. Each reachable region has a unique representative state in σ^R, for example, state s_0 for region σ_0. The reachability information is computed by considering the initial regions and by exploring successors of the representatives. Thus, at the end of the searching phase, we are guaranteed that the regions σ_2, σ_3 and σ_6 contain neither initial states nor successors of the representative states of the reachable regions.

In the splitting phase, the algorithm computes, for each reachable region σ, the subregion σ' that is stable with respect to the partition $\cong^{prev}$, and contains the reachable state $\sigma^R \cap \sigma$. Instead of splitting σ with respect to each region of $\cong^{prev}$, σ is split in at most two regions to avoid proliferation of regions.

To understand the splitting, reconsider the illustration of Figure 4.9. For the region σ_0, the algorithm computes the subregion σ'_0 (shown by the dotted lines) that contains states that agree with the representative s_0: for every state t in σ'_0, $post(t)$ intersects with σ_1 and σ_4, and does not intersect with σ_0, σ_2, σ_3, σ_5

Algorithm 4.5 [Lee-Yannakakis Algorithm]

Input: a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$ and a G-partition $\cong^I$.
Output: the answer to the reachable-partition-refinement problem $(G, \cong^I)$.

```
local $\cong, \cong^{prev}$: partition; $\sigma^R, \sigma, \tau, \upsilon, \sigma'$: region; $E$: set of
    region×region; $s, t$: state, $U$: stack of state
$\cong := \cong^I$;  $\sigma^R := \mathit{EmptySet}$
foreach $\sigma$ in $\cong^I$ do
  if $\sigma \cap \mathit{InitQueue}(G) \neq \emptyset$ then
    $\mathit{Insert}(\mathit{Element}(\sigma \cap \mathit{InitQueue}(G)), \sigma^R)$ fi od
repeat
Search:
  $U := \mathit{EmptyStack}$; $E := \mathit{EmptySet}$; $\cong_R := \mathit{EmptySet}$
  foreach $s$ in $\sigma^R$ do
    $U := \mathit{Push}(s, U)$; $\cong_R := \mathit{Insert}(\mathit{Find}(s, \cong), \cong_R)$ od
  while not $\mathit{EmptySet}(U)$ do
    $s := \mathit{Top}(U)$; $U := \mathit{Pop}(U)$; $\sigma := \mathit{Find}(s, \cong)$
    foreach $t$ in $\mathit{PostQueue}(s, G)$ do
      $\tau := \mathit{Find}(t, \cong)$; $E := \mathit{Insert}((\sigma, \tau), E)$
      if not $\mathit{IsMember}(\tau, \cong_R)$ then
        $\sigma^R := \mathit{Insert}(t, \sigma^R)$; $U := \mathit{Push}(t, U)$;
        $\cong_R := \mathit{Insert}(\tau, \cong_R)$ fi
      od
    od
Split:
  $\cong^{prev} := \cong$
  foreach $\sigma$ in $\cong_R$ do
    $\sigma' := \sigma$; $\tau := \mathit{PostQueue}(\sigma, G)$
    foreach $(\sigma, \upsilon)$ in $E$ do $\sigma' := \sigma' \cap \mathit{PreQueue}(\upsilon, G)$; $\tau := \tau \backslash \upsilon$ od
    $\sigma' := \sigma' \backslash pre(\tau)$
    if $\sigma \neq \sigma'$ then
      $\cong := \mathit{Insert}(\sigma', \mathit{Insert}(\sigma \backslash \sigma', \mathit{Delete}(\sigma, \cong)))$
      if $\sigma \backslash \sigma' \cap \mathit{InitQueue}(G) \neq \emptyset$ then
        $\sigma^R := \mathit{Insert}(\mathit{Element}(\sigma \backslash \sigma'), \sigma^R)$ fi fi
    od
  od
until $\cong^{prev} = \cong$
return $\cong_R$.
```

Figure 4.8: Lee-Yannakakis algorithm for partition refinement

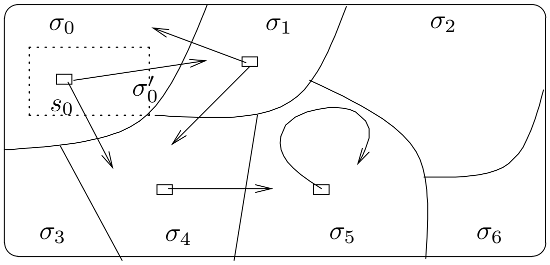


Figure 4.9: Computation of Lee-Yannakakis Algorithm

and σ_6. Clearly, σ_0' is nonempty as it contains s_0. Furthermore, σ_0' is known to be reachable with the representative state s_0. If it differs from σ_0, then σ_0 is split at the boundary of σ_0'. If the split part $\sigma_0 \backslash \sigma_0'$ contains an initial state, then it is declared reachable by choosing a representative state.

Lemma 4.5 [Stabilization in Lee-Yannakakis algorithm] *Let $\cong^{prev}$ be the value of the partition at the beginning of the splitting phase during some iteration of Algorithm 4.5. Let σ be a region in $\cong_R$, and let s be the unique state in $\sigma^R \cap \sigma$. Then, the subregion σ' computed at the end of the for-loop contains precisely those states t such that $t \rightarrow_G \tau$ iff $s \rightarrow_G \tau$ for all τ in $\cong^{prev}$.*

Exercise 4.17 {T3} [Stabilization in Lee-Yannakakis] Prove Lemma 4.5. ■

Once the subregion σ' is computed, the region σ is split at the boundary of σ'. The crucial aspect of the splitting strategy is that all regions are given a fair chance, in a round-robin order, to split.

Exercise 4.18 {P2} [Computing minimal-reachable quotient by Lee-Yannakakis] Execute Algorithm 4.5 on the input of Exercise 4.16. How many iterations are required before termination? ■

Suppose the reachable stable partition $\min^R(\cong^I)$ has n regions. During the execution of Algorithm 4.5, the number of regions that contain some reachable state of G is bounded by n. The convergence is established by the following lemma.

Lemma 4.6 [Convergence] *At the end of an iteration of the repeat-loop of Algorithm 4.5, the number of regions σ in $\cong$ with $\sigma \cap post_G^*(\sigma^I) \neq \emptyset$ either equals the number of regions in $\min^R(\cong^I)$, or exceeds the number of regions τ in $\cong^{prev}$ with $\tau \cap post_G^*(\sigma^I) \neq \emptyset$.*

Proof. Let $\upsilon = post^*_G(\sigma^I)$ be the reachable region of G. During the splitting phase each region σ in $\cong_R$ is split into two regions σ' and $\sigma \backslash \sigma'$ such that σ contains a reachable state and is stable with respect to each τ in $\cong^{prev}$ (Lemma 4.5). There are two cases to consider.

Case 1: for some σ in $\cong_R$, $\sigma \backslash \sigma'$ contains a state in υ. Then, the number of regions in the new partition containing reachable states exceeds the number of regions in the old partition containing reachable states.

Case 2: for all regions $\sigma \in \cong_R$, $\sigma \backslash \sigma'$ does not contain a state in υ. Let $\cong'_R$ be the set of regions σ'. We show that every region of the reachable stable partition $\min^R(\cong^I)$ is contained in a region of $\cong'_R$, and thus, the sets $\cong_R$, $\cong'_R$, and $\min^R(\cong^I)$ have the same cardinality.

First, we prove that every state in υ belongs to some σ'. We already know that, for all σ in $\cong_R$, $(\sigma \backslash \sigma') \cap \upsilon$ is empty. It suffices to show that for every region τ in $\cong^{prev} \setminus \cong_R$, $\tau \cap \upsilon$ is empty. Consider a region τ in $\cong^{prev} \setminus \cong_R$. Whenever a newly created region contains an initial state, one of its state is added to σ^R, and hence, the region gets added to $\cong_R$. Hence, $\tau \cap \sigma^I$ is empty. During the searching phase, all successors of all the states in σ^R are explored, and hence, $\tau \cap post(\sigma^R)$ is empty. Since every σ' in $\cong'_R$ is stable with respect to τ, and contains a state in σ^R, it follows that $\tau \cap post(\sigma')$ is empty for all σ' in $\cong'_R$. It follows that $\tau \cap \upsilon$ is empty.

For every σ' in $\cong'_R$, let $\sigma'' = \sigma' \cap \upsilon$. Consider two regions σ and τ in $\cong_R$. We know that σ' is stable with respect to τ. It follows that σ'' is also stable with respect to τ. Since $\sigma'' \subseteq \upsilon$, $post(\sigma'') \cap \tau = post(\sigma'') \cap \tau''$. It follows that σ'' is stable with respect to τ''. We conclude that the final output $\min^R(\cong^I)$ contains as many regions as $\cong_R$. ■

The running time of the algorithm depends upon the time complexity of the primitive operations on regions and partitions. If the number of successors of a state in G is bounded by k, then each searching phase requires at most kn operations. The number of operations during a splitting phase is bounded by the number of regions in $\cong_R$ and the number of edges in E. If the number of edges in the minimal-reachable quotient is m, then the size of E is bounded by m, and the splitting phase requires at most $m + n$ primitive operations. If $\min(\cong^I)$ has ℓ equivalence classes then Algorithm 4.5 is guaranteed to terminate after ℓ iterations.

Exercise 4.19 {T3} [Optimization of Lee-Yannakakis algorithm] The searching phase of Algorithm 4.5 builds the graph $(\cong_R, E)$ from scratch in each iteration. Suggest a modification so that the computation of one iteration is reused in the next. ■

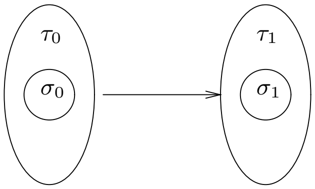


Figure 4.10: Reachable semistable quotient

Early termination

Lemma 4.6 ensures that the size of $\cong_R$ does not change after n iterations. Indeed, the graphs $(\cong_R, E)$ computed in searching phases are isomorphic after the n-th iteration. The splitting phase only removes subregions from each of the regions in $\cong_R$ without influencing the structure of the graph.

> Reachable semistable quotient
>
> Let G be a transition graph and let $\cong^I$ be a G-partition. The pair $(\cong, E)$, for a set $\cong$ of regions of G and a set $E \subseteq \cong \times \cong$ of edges between the regions of $\cong$, is a *reachable* $\cong^I$*-semistable quotient* of G if (1) every region σ of the reachable stable partition $\min^R(\cong^I)$ is contained in a region $f(\sigma)$ of $\cong$, and (2) for two regions σ and τ of the reachable stable partition $\min^R(\cong^I)$, $\sigma \rightarrow_{\min(\cong^I)} \tau$ iff $(f(\sigma), f(\tau)) \in E$.

To understand the definition consider Figure 4.10 which shows a transition of a reachable semistable quotient of a transition graph G. Each region τ_i of a reachable semistable quotient contains a nonempty region σ_i that contains reachable states of G (the union of all σ_i equals the reachable region of G). The definition requires the transition from τ_0 to τ_1 to be stable with respect to the reachable subregions σ_i: from every state s in σ_0, $post_G(s) \cap \sigma_1$ is nonempty. However, there may be a state $s \in \tau_0 \backslash \sigma_0$ such that $post_G(s) \cap \tau_1$ is empty.

Example 4.6 [Semistable quotient] Consider the symbolic graph G with an integer variable x and a variable y that ranges over the interval $[0, 1]$ of real numbers. The initial predicate is $x = 0 \wedge y = 0$, and the transition predicate is

$$(x' = x + 1) \;\wedge\; (2y \leq 1 \;\rightarrow\; y' = 2y).$$

The initial partition $\cong^I$ contains the single region with all the states. In this case, $\min(\cong^I)$ has infinitely many regions. The reachable region is $[\![y = 0]\!]$, and contains infinitely many states. However, the minimal-reachable quotient is finite: for $\sigma = [\![y = 0]\!]$, the graph with the single state σ, and the single transition from σ to σ, is the minimal-reachable-quotient. There are infinitely many reachable semistable quotients. Executing Algorithm 4.5 for i iterations yields the semistable quotient that contains a single region $[\![y = 1/2^i]\!]$ with a single self-loop. ■

It follows that for a given transition graph and an initial partition, the corresponding reachable semistable partition is not uniquely defined. Two such reachable semistable partitions are isomorphic graphs. To solve a reachability problem for the transition graphs, it suffices to construct any reachable semistable quotient with respect to a suitably chosen initial partition.

Proposition 4.3 [Reachability and semistable quotients] *Let G be a transition graph, $\cong^I$ be a G-partition, and σ be a block of $\cong^I$. If $(\cong, E)$ is a reachable $\cong^I$-semistable quotient of G, then the answer to the reachability problem (G, σ) is* YES *iff $\sigma \cap \tau$ is nonempty for some τ in $\cong$.*

The Lee-Yannakakis algorithm is guaranteed to compute a reachable semistable quotient after linearly many iterations.

Theorem 4.7 [Computation of semistable partition] *Given an instance $(G, \cong^I)$ of reachable-partition-refinement, if the reachable stable partition $\min^R(\cong^I)$ has n regions, the pair $(\cong_R, E)$ at the end of the n-th iteration of the repeat-loop of Algorithm 4.5 is a reachable $\cong^I$-semistable quotient of G.*

Exercise 4.20 {T3} [Convergence to semistable quotient] Modify the proof of Lemma 4.6 to prove Theorem 4.7. ■

Since it suffices to compute a reachable semistable quotient to solve reachability problems, the execution of Algorithm 4.5 can be aborted, if there is a procedure that determines whether $(\cong_R, E)$ is a reachable semistable quotient. Observe that deciding whether $(\cong_R, E)$ is a reachable semistable quotient is an easier (static) problem compared to the dynamic problem of constructing one. While there are no general algorithms for this purpose, specialized solutions can be employed to exploit the structure of the update commands.

Exercise 4.21 {T4} [Cylinder-based refinement computation] Consider an instance $(P, \cong^I)$ of the reachable-partition-refinement problem with the following characteristics. The module P is a ruleset with a single enumerated variable x and k real-valued variables Y. Thus, the state-space Σ_P is the produce $x \times \mathbb{R}^k$. A *rational interval* I is a convex subset of $\mathbb{R}$ with rational endpoints. A region σ of P is *convex* if for all s and t in σ with $s(x) = t(x)$, for all $0 \leq \delta \leq 1$, the state u is also in σ, where $u(pc) = s(pc)$ and $u(y) = \delta \cdot s(y) + (1 - \delta) \cdot t(y)$ for all $y \in Y$. A region σ of P is a *cylinder* if there exist a value $m \in x$ and intervals I_y for variables $y \in Y$ such that a state s of P belongs to σ iff $s(x) = m$ and $s(y) \in I_y$ for $y \in Y$. Assume that the initial region σ_P^I is a cylinder, and every region in the initial partition $\cong^I$ is a cylinder. Furthermore, for every guarded assignment γ in the update command of P, the guard p_γ is a cylinder, and for all $y \in Y$, $e_\gamma^{y'}$ is of the form $az + b$ for some rational numbers a, b, and a variable $z \in Y$.

(1) Show that if a region σ of P is a cylinder, then $post_P(\sigma)$ is a finite union of cylinders, and $pre_P(\sigma)$ is a finite union of cylinders. (2) Show that every region in $\min(\cong^I)$ is convex. (3) Show that every region in $\min(\cong^I)$ is a cylinder. (4) Show that the problem of checking whether $(\cong, E)$ is a reachable $\cong^I$-semistable quotient can be formulated as a linear programming problem. What is the time-complexity of your test for semistability? ■

Appendix: Notation

Equivalences and partitions

A *partition* of a set A is a set of nonempty, pairwise disjoint subsets of A whose union is A. There is a one-to-one correspondence between the equivalences on A and the partitions of A. Given an equivalence $\cong$ on A and an element $a \in A$, we write $a/_{\cong}$ for the $\cong$-*equivalence class* $\{b \in A \mid b \cong a\}$ of a. The set $A/_{\cong}$ of $\cong$-equivalence classes is a partition of A. In this way, each partition of A is induced by a unique equivalence on A. Therefore, whenever we refer to a partition of A, we use a notation like $A/_{\cong}$, which indicates the corresponding equivalence. We also freely attribute properties and derivatives of equivalences to partitions, and vice versa.

Let $\cong$ be an equivalence on A. The equivalence $\cong$ is *finite* if $\cong$ has finitely many equivalence classes. A union of $\cong$-equivalence classes is called a *block* of $\cong$. If $\cong$ is finite with n equivalence classes, then $\cong$ has 2^n blocks. Given two equivalences $\cong_1$ and $\cong_2$ on A, the equivalence $\cong_1$ *refines* the equivalence $\cong_2$, written $\cong_1 \preceq \cong_2$, if $a \cong_1 b$ implies $a \cong_2 b$. If $\cong_1$ refines $\cong_2$, then every block of $\cong_2$ is a block of $\cong_1$. For a set E of equivalence relations on A, the *join* $\bigcup^* E$ is the transitive closure of the union $\bigcup E$ of the relations in E; the join $\bigcup^* E$ is an equivalence on A. The refinement relation $\preceq$ is a complete lattice on the set of equivalences on A. The least upper $\preceq$-bound for a set E of equivalences on A is the join $\bigcup^* E$; the greatest lower $\preceq$-bound for E is the intersection $\bigcap E$.

Exercise 4.22 $\{\}$ [Partition theorems] Prove all claims made in the previous paragraph. ■

Groups

A *goup* is a set A with a binary function $\circ \colon A^2 \mapsto A$, called the multiplication operation, such that (1) $\circ$ is associative, (2) there exists an element that is identity for $\circ$, and (3) every element of A has an inverse with respect to $\circ$. Consider a group $(A, \circ)$. A *subgroup* of A is a subset $B \subseteq A$ such that $(B, \circ)$ is a group. For a subset $B \subseteq A$, the subgroup generated by B, denoted $\mathit{closure}(B)$, is the smallest subgroup of $(A, \circ)$ that contains B. The elements in B are called *generators* for the group $\mathit{closure}(B)$.

Contents

Chapter 5

Real-Time Modules

In reactive modules, the progress of time is abstracted into a sequence of rounds. The abstraction of time in this fashion is convenient in many circumstances. First, every round may represent a clock cycle, as in our model of synchronous circuits. Second, a special tick event may represent a clock cycle, with an arbitrary number of rounds between ticks, as in our model of timed asynchronous circuits. Third, quantitative time may be irrelevant altogether to the problem at hand, as in our model of mutual-exclusion protocols, whose correctness ought to be independent of the relative speeds of the processes. Sometimes, however, it is necessary to take a more accurate, *real-numbered* view of time. For this purpose, we introduce an extension of reactive modules called *real-time modules*.

While the behavior of a reactive module is a sequence of update rounds, the behavior of a real-time module is a sequence of update and delay rounds. During an update round, the values of variables are updated; during a delay round, the values of ordinary variables remain unchanged, while the values of special variables called *clocks* measure the amount of time that elapses. We assume the *synchrony hypothesis*, that no time elapses during update rounds. The synchrony hypothesis is a modeling assumption; it simply asserts that all time delays must be modeled explicitly by delay rounds. For example, if an assignment to a variable takes 1 time unit, it must be modeled as a delay round of duration 1 followed by an update round that changes the value of the variable. For mathematical purposes, it will be convenient to separate updates from delays in this fashion.

5.1 Clock Variables

Real-time modules have two kinds of variables: *discrete variables* and *clock variables* (or *clocks*, for short). Discrete variables are updated by guarded assignments, as before. Clock variables range over the nonnegative real numbers, and may change in two different ways. First, like a discrete variable, a clock may be updated by a guarded assignment. Second, while time elapses, the value of a clock increases, measuring the progress of time. We declare clock variables using the type ; all other declared variables are discrete by default. A reactive module, then, is simply the special case of a real-time module without clocks.

Consider two sets X and Y of typed variables with $Y \subseteq X$. A *guarded delay* γ *from* X *to* Y is a boolean expression p_γ over X, written

$$\gamma \;\rightarrow\; \textbf{wait}.$$

Informally, the guarded delay γ can be executed if the boolean expression p_γ evaluates to true. The execution of γ leaves the value of each discrete variable in Y unchanged, and advances the value of each clock variable in Y by some uniform real value δ such that the truth value of p_γ remains true throughout the advancement. Given a valuation s for X, and a real number ϵ, by $s + \epsilon$ we denote the valuation for X that maps each discrete variable x in X to $x[s]$, and each clock variable y in X to $y[s] + \epsilon$. The guarded delay γ defines a ternary relation $[\![\gamma]\!] \subseteq \Sigma_X \times \mathbb{R}^{\geq 0} \times \Sigma_Y$: $(s, \delta, t) \in [\![\gamma]\!]$ iff (1) $t = Y[s] + \delta$ and (2) for every nonnegative real $\epsilon \leq \delta$, the valuation $s + \epsilon$ satisfies γ.

A *guarded real-time command* Γ *from* X *to* Y is a finite set $\{\gamma_i \mid i \in I\}$ of guarded assignments and guarded delays from X to Y such that the disjunction $(\vee i \in I.\, p_{\gamma_i})$ of the guards is valid. The guarded real-time command Γ defines a ternary relation $[\![\Gamma]\!] \subseteq \Sigma_X \times \mathbb{R}^{\geq 0} \times \Sigma_Y$: $(s, \delta, t) \in [\![\Gamma]\!]$ iff for some $i \in I$, either (1) γ_i is a guarded assignment, $\delta = 0$, and $[\![\gamma_i]\!](s) = t$, or (2) γ_i is a guarded delay and $(s, \delta, t) \in [\![\gamma_i]\!]$.

REAL-TIME MODULES

A *real-time atom* is an atom whose controlled variables are read, and whose update command is a guarded real-time command. A *real-time module* is a reactive module whose atoms are real-time atoms.

Each real-time module defines a transition graph. The states of a real-time modules are the valuations for the module variables. Since the values of clock variables are real numbers, the state space of a real-time module is usually infinite. A real-time module has two kinds of transitions. Update transitions correspond to guarded assignments, which update the values of discrete and clock variables, and time transitions correspond to guarded delays, which advance the values of clock variables and leave the values of discrete variables unchanged.

TRANSITION RELATION OF A REAL-TIME MODULE

Let P be a real-time module, let s and t be two states of P, and let $t' = t[X_P := X'_P]$. The state pair (s, t) is a *transition* of P if there is a nonnegative real δ such that for every atom U of P,

$$(\mathsf{read}X_U[s] \uplus \mathsf{await}X'_U[t'], \delta, \mathsf{ctr}X'_U[t']) \in [\![\mathsf{update}_U]\!],$$

and if $\delta > 0$, then for every external discrete variable x of P, $x[t] = x[s]$, and for every external clock variable y of P, $y[t] = y[s] + \delta$. If $\delta = 0$, then (s, t) is an *update transition*; if $\delta > 0$, then (s, t) is a *time transition of duration* δ.

Remark 5.1 [Time transitions] If (s, t) is a time transition of duration δ, then $t = s + \delta$. Moreover, for each nonnegative real $\epsilon < \delta$, $(s, s + \epsilon)$ is also a time transition. It follows that the transition graph of a real-time module is usually infinitely branching. ■

Operationally, in each update round the atoms of a module are sorted topologically with respect to the precedes relation. In each subround, an atom either updates its controlled variables, or proposes a time delay (the proposed time delay is not known to the other atoms). After all subrounds, if some atom has updated its controlled variables, then the module moves instantaneously to the next update round. If, on the other hand, all atoms have proposed time delays, then the module waits for the amount of time equal to the minimum of the proposed time delays, before moving to the next update round.

Example 5.1 [Real-time counter] The following module increments a counter x every 3 to 5 time units:

```
module RealTimeCount
  interface $x$ :
  private $y$ :
  atom controls $x, y$ reads $x, y$
    init
      [] $true \to x' := 0;\ y' := 0$
    update
      [] $y \geq 3 \to x' := x + 1;\ y' := 0$
      [] $y \leq 5 \to$ wait
```

■

5.2 Real-time Invariant Verification

The invariant verification problem is decidable for an interesting class of real-time modules, because we can construct finite stable partitions of the infinite state spaces.

```
module $RealTimeTrain$
  interface $pc : \{far, near, gate\};\ arrive :$
  private $x :$
  atom controls $pc, x, arrive$ reads $pc, x, arrive$
    init
      $\| \ true \rightarrow pc' := far$
    update
      $\| \ pc = far \rightarrow pc' := near;\ arrive!;\ x' := 0$
      $\| \ pc = far \rightarrow$ wait
      $\| \ pc = near \wedge x \geq 3 \rightarrow pc' := gate;\ x' := 0$
      $\| \ pc = near \wedge x \leq 5 \rightarrow$ wait
      $\| \ pc = gate \wedge x \geq 1 \rightarrow pc' := far$
      $\| \ pc = gate \wedge x \leq 2 \rightarrow$ wait

module $RealTimeGate$
  external $arrive :$
  interface $pc : \{open, closing, closed\}$
  private $y :$
  atom controls $pc, y$ reads $pc, y, arrive$ awaits $arrive$
    init
      $\| \ true \rightarrow pc' := open$
    update
      $\| \ pc = open \wedge arrive? \rightarrow pc' := closing;\ y' := 0$
      $\| \ pc = open \wedge \neg arrive? \rightarrow$ wait
      $\| \ pc = closing \wedge y \geq 1 \rightarrow pc' := closed;\ y' := 0$
      $\| \ pc = closing \wedge y \leq 2 \rightarrow$ wait
      $\| \ pc = closed \wedge y \geq 7 \rightarrow pc' := open$
      $\| \ pc = closed \wedge y \leq 7 \rightarrow$ wait
```

Figure 5.1: Real-time railroad gate control

Propositional real-time modules

The *clock constraints* are the boolean expressions generated by the grammar

$$p \;::=\; x \leq c \mid c \leq x \mid x + c \leq y + d \mid p \wedge p,$$

where x and y are clock variables, and c and d are integer constants. A *clock formula* is a boolean combination of propositional formulas and clock constraints. A *propositional real-time module* is a real-time module P such that (1) all discrete variables of P are propositions, and (2) every expression that appears in the guards of initial and update commands of P is a clock formula, and (3) every expression that appears in the assignments of initial and update commands of P is a propositional formula or an integer constant.

Propositional real-time invariant verification

An instance (P, r) of the *propositional invariant-verification problem* consists of a propositional real-time module P and a clock formula r that is an observation predicate for P. The instance (P, r) *has k boolean variables and l clock variables and maximal constant c_{max}* if the module P has k boolean variables and l clock variables, and the maximal integer constant that occurs in either P or r is c_{max}.

Example 5.2 [Real-time train] Figure 5.1 shows an example of two propositional real-time modules that model a train and a gate controller. Consider the train system

$$\textbf{module } \mathit{RailRoadXing} \textbf{ is } \mathit{RealTimeTrain}[pc := pc_T] \parallel \mathit{RealTimeGate}[pc := pc_G].$$

We want to show that the clock formula

$$pc_T = gate \;\rightarrow\; pc_G = closed$$

is an invariant of *RailRoadXing*. ■

5.2.1 Partition Refinement

Theorem 5.1 [Propositional real-time modules] [Alur and Dill] *Let P be a propositional real-time module, and let $\cong^I$ be a finite partition of G_P such that every equivalence class of $\cong^I$ can be defined by an observation predicate for P that is a clock formula. Then the coarsest stable refinement* $\min(\cong^I)$ *has finitely many equivalence classes.*

Proof. Let c be the largest constant occuring in P and the formulas that define the equivalence classes of $\cong^I$. Define the *region equivalence* $s \cong t$ iff (1) for all discrete variables x of P, $x[s] = x[t]$; (2) for all clock variables y of P, either $\lfloor y[s] \rfloor = \lfloor y[t] \rfloor$ and $\lceil y[s] \rceil = \lceil y[t] \rceil$, or $\lceil y[s] \rceil > c$ and $\lceil y[t] \rceil > c$; and

(3) for all clock variables y and z of P, $\langle x[s]\rangle < \langle y[s]\rangle$ iff $\langle x[t]\rangle < \langle y[t]\rangle$ (where $\langle x\rangle = x - \lfloor x\rfloor$). Then $\cong$ is a stable refinement of $\cong^I$. ■

It follows that symbolic backward reachability checking and symbolic partition refinement terminates on the transition graphs of propositional real-time modules. The number of equivalence classes is $O(2^{k+l} \cdot l! \cdot (c_{max} + 1))$, which is exponential in the size of the module P.

Corollary 5.1 [Real-time invariant verification] *The propositional real-time invariant verification problem can be solved in exponential time.*

Exercise 5.1 {T3} [PSPACE invariant verification] Prove that the propositional real-time invariant verification problem can be solved polynomial space. ■

Exercise 5.2 {P2} [Region graph] The quotient graph $G_P/_\pi$, where P is a propositional real-time module and π is the equivalence relation from the proof of Theorem 5.1, is called the *region graph* of P. Draw the region graph of the real-time module *RailRoadXing*. ■

Exercise 5.3 {P3} [Coarsest stable refinement] Consider again the real-time module *RailRoadXing* and the initial partition $\hat{\pi}$, with two equivalence classes, that separates the states in which the train is inside a non-closed gate from all other states. (1) Step through partition refinement to find the coarsest stable refinement of $\hat{\pi}$. (2) Step through the Lee-Yannakakis algorithm. ■

5.2.2 Symbolic Analysis of Propositional Real-time Modules

The transition predicates of real-time modules can be constructed using quantifiers over the reals. For example, the transition predicate of the module *RealTimeTrain* is

$$\begin{array}{l}
\vee\,(pc = far \,\wedge\, pc' = near \,\wedge\, x' = 0 \,\wedge\, arrive' \neq arrive) \\
\vee\,(\exists \delta \geq 0.\, pc' = pc \,\wedge\, x' = x + \delta \,\wedge\, arrive' = arrive \,\wedge\, (\forall 0 \leq \epsilon \leq \delta.\, pc = far)) \\
\vee\,(pc = near \,\wedge\, x \geq 3 \,\wedge\, pc' = gate \,\wedge\, x' = 0 \,\wedge\, arrive' = arrive) \\
\vee\,(\exists \delta \geq 0.\, pc' = pc \,\wedge\, x' = x + \delta \,\wedge\, arrive' = arrive \,\wedge\, (\forall 0 \leq \epsilon \leq \delta.\, pc = near \,\wedge\, x + \epsilon \leq 5)) \\
\vee\,(pc = gate \,\wedge\, x \geq 2 \,\wedge\, pc' = far \,\wedge\, x' = x \,\wedge\, arrive' = arrive) \\
\vee\,(\exists \delta \geq 0.\, pc' = pc \,\wedge\, x' = x + \delta \,\wedge\, arrive' = arrive \,\wedge\, (\forall 0 \leq \epsilon \leq \delta.\, pc = gate \,\wedge\, x + \epsilon \leq 2))
\end{array}$$

This transition predicate is not a clock formula, because it contains existential quantifiers, and subformulas of the form $x' = x + \delta$, which are not clock constraints. By eliminating the existential quantifiers, we obtain the equivalent formula

$$\begin{array}{l}
\vee\,(pc = \mathit{far} \,\wedge\, pc' = \mathit{near} \,\wedge\, x' = 0 \,\wedge\, \mathit{arrive}' \neq \mathit{arrive}) \\
\vee\,(pc = \mathit{far} \,\wedge\, pc' = \mathit{far} \,\wedge\, x' \geq x \,\wedge\, \mathit{arrive}' = \mathit{arrive}) \\
\vee\,(pc = \mathit{near} \,\wedge\, x \geq 3 \,\wedge\, pc' = \mathit{gate} \,\wedge\, x' = 0 \,\wedge\, \mathit{arrive}' = \mathit{arrive}) \\
\vee\,(pc = \mathit{near} \,\wedge\, pc' = \mathit{near} \,\wedge\, x \leq x' \leq 5 \,\wedge\, \mathit{arrive}' = \mathit{arrive}) \\
\vee\,(pc = \mathit{gate} \,\wedge\, x \geq 2 \,\wedge\, pc' = \mathit{far} \,\wedge\, x' = x \,\wedge\, \mathit{arrive}' = \mathit{arrive}) \\
\vee\,(pc = \mathit{gate} \,\wedge\, pc' = \mathit{gate} \,\wedge\, x \leq x' \leq 2 \,\wedge\, \mathit{arrive}' = \mathit{arrive}).
\end{array}$$

While this is a clock formula, this need not be the case. To see this, consider the propositional real-time module with two clocks, x and y, and the single guarded delay $\mathit{true} \to \mathbf{wait}$. The transition predicate of this module is

$$(\exists \delta \geq 0.\, x' = x + \delta \,\wedge\, y' = y + \delta)$$

or equivalently, after quantifier elimination,

$$x' \geq x \,\wedge\, x' - x = y' - y.$$

The synchronization of the two clocks introduces a constraint between clock differences.

Exercise 5.4 {T3} [Real-time transition predicates] (1) Given a real-time module P, define the transition predicate of P. (2) A *clock difference formula* is like a clock formula, only that atomic subformulas may have the form $x-y=z-u$, for clock variables x, y, z, and u. Prove that for every propositional real-time module, the transition predicate is equivalent to a clock difference formula (use existential-quantifier elimination for δ). (3) How expensive is it to construct the clock difference formula that is equivalent to the transition predicate of P? ■

Consider a propositional real-time module P. Suppose you are given a clock formula p (over the unprimed variables X, which contain both propositions and clocks) which defines a region σ, and a clock difference formula q (over the unprimed and primed variables $X \cup X'$) which is equivalent to the transition predicate of P. Then the region $\mathit{pre}(\sigma)$ is defined by the formula $r = (\exists X'.p[X := X'] \,\wedge\, q)$ (and the region $\mathit{post}(\sigma)$ is defined by the formula $r' = (\exists X.p \,\wedge\, q)[X' := X]$). By Theorem 5.1, the formula r is again a clock formula. (What about r'?) This formula can be found by existential-quantifier elimination.

Exercise 5.5 {T3} [Reals with addition] The *first-order theory of the reals with addition* contains all atomic formulas that are either boolean variables or have the form $t \sim c$, where t is a sum of clock variables, $\sim$ is either $\leq$ or $\geq$, and c is an integer constant. In particular, all clock difference formulas are quantifier-free formulas of the first-order theory of the reals with addition. (1) Show that this theory permits quantifier elimination; that is, for every formula there is an equivalent quantifier-free one. What is the complexity of your quantifier-elimination procedure? (2) Show that this theory has a decidable satisfiability problem. What is the complexity class of the satisfiability problem? ■

Exercise 5.6 {`P3`} [Real-time reachability analysis] Give a symbolic forward reachability algorithm for propositional real-time modules. Represent all regions that are computed by your algorithms using clock formulas. Step your forward algorithm through a proof that the train *RealTimeTrain* is never in the gate when the gate *RealTimeGate* is closed. Here "stepping through a proof" means to list the clock formulas for σ^I, $post(\sigma^I)$, $post^2(\sigma^I)$, etc., for the real-time module *RailRoadXing*. ■

Exercise 5.7 {`T3`} [Forward reachability analysis] Symbolic backward reachability analysis is guaranteed to terminate for propositional real-time modules, because every region constructed by the algorithm is a block of the region equivalence (which has only finitely many blocks). The same cannot be said for forward analysis. (1) Give a simple propositional real-time module for which symbolic forward reachability analysis does not terminate. (2) Modify the forward algorithm so that it is guaranteed to terminate on all propositional real-time modules. ■

Exercise 5.8 {`P3`} [Real-time mutual exclusion] If clocks are available, mutual exclusion can be guaranteed in a quite simple way. Formalize the following protocol as propositional real-time modules, and step a forward-reachability algorithm through a proof that the protocol ensures mutual exclusion. In your protocol, assume that each assignment requires 2 time units. The two competing processes share a variable k whose value is initially 0, and always either 0, 1, or 2. When the first (second) process wants to enter its critical section, it waits until $k = 0$, then sets k to 1 (resp. 2), then waits for 3 time units, then enters its critical section if k is still 1 (resp. 2); if the value of k is no longer 1, the process repeats the sequence starting from waiting until $k = 0$. Upon leaving its critical section, the process resets k to 0. (Since k is a write-shared variable, you must model it as a separate module.) ■

5.2.3 Difference-bound Matrices

For representing the regions of a propositional real-time module which can be defined by clock formulas, a more efficient symbolic representation is based on difference-bound matrices. For a region R, let $timepre(R)$ the set of all states from which a state in R can be reached by a time step; that is, $timepre(R) = \{s \mid \exists t \in R.\, \exists \delta \geq 0.\, t = s + \delta\}$. For a clock formula p, let $timepre(p)$ be the formula $(\exists \delta \geq 0.\, p + \delta)$, where $p + \delta$ is obtained from p by replacing each clock variable x with $x + \delta$. Define *timepost* similarly.

Exercise 5.9 {T3} [Clock formulas] (1) Prove that the clock constraints are closed under *timepre*, *timepost*, and existential-quantifier elimination, and have a decidable satisfiability problem. What is the cost of each operation? (2) Repeat the exercise for clock formulas. ■

Suppose you are given a clock formula p (over the unprimed variables X, which contain both propositions and clocks) which defines a region σ, a clock formula formula q (over the unprimed and primed variables $X \cup X'$) which defines the discrete transitions of P, and a clock formula formula r (over the unprimed variables X) which defines the time invariant of P (i.e., the disjunctions of all guards of guarded delays). Then the region $\mathit{pre}(\sigma)$ is defined by the clock formula $r = (\exists X'.p[X := X'] \wedge q) \vee (\mathit{timepre}(p) \wedge r)$, and the region $\mathit{post}(\sigma)$ is defined by the clock formula $r = (\exists X.p \wedge q)[X' := X] \vee (\mathit{timepost}(p) \wedge r)$. This, together with the previous exercise, gives us symbolic backward and forward reachability algorithms which do not rely on arbitrary formulas of the first-order theory of the reals with addition, but only on clock formulas. And for clock formulas, there is an efficient symbolic representation.

Exercise 5.10 {T3} [Difference-bound matrices (DBMs)] Represent clock constraints with n clocks by integer square matrices with $n+1$ rows and $n+1$ columns. For two clocks x and y, the (x, y) entry contains a tight upper bound on the clock difference $x - y$ (or ∞, if there is no such upper bound). The $(x, n{+}1)$ entry contains a tight upper bound on the value of x, and the $(n{+}1, x)$ entry contains a tight upper bound on $-x$. Every matrix entry also contains a "bit" indicating if the bound is strict ($<$) or weak ($\leq$). (1) Is this representation canonical? If not, how would you make it canonical? (2) Give algorithms for computing conjunction, equivalence checking, satisfiability checking, renaming, existential-quantifier elimination (for clock variables), *timepre* and *timepost* on the matrix representation of clock constraints. What is the cost of each operation? ■

Exercise 5.11 {T3} [Combining boolean state and DBMs] Devise a "semi-symbolic" representation for clock formulas. Clock formulas result from clock constraint by adding both disjunction and propositions. Every clock formula can be thought of as a set of boolean states (valuations for the propositions), and for each boolean state, a set (disjunction) of clock constraints represented by DBMs. This representation is called semi-enumerative, because boolean state is represented enumeratively and only clock state is represented symbolically (by DBMs). Give algorithms for the boolean operations, satisfiability checking, implication and equivalence checking, renaming, existential-quantifier elimination (for both propositions and clocks), *timepre* and *timepost* on your representation of clock formulas. You may use the algorithms from the previous exercise as black-box subroutines. What is the cost of each operation? ■

Exercise 5.12 {`T4`} [Combining BDDs and DBMs] Devise a "fully-symbolic" representation for clock formulas. Represent the boolean part of a clock formula as a BDD such that each leave does not point to 0 or 1, but to a set (disjunction) of BDMs. As in the previous exercise, give algorithms for the boolean operations, satisfiability checking, implication and equivalence checking, renaming, existential-quantifier elimination (for both propositions and clocks), *timepre* and *timepost* on your representation of clock formulas. What is the cost of each operation? ■

Contents

Chapter 6

Temporal Safety Requirements

6.1 Logical Requirements of Reactive Modules

The invariant-verification problem allows us to check if all reachable states of a reactive module satisfy an observation predicate. Not all module requirements can be formulated as invariants. For instance, for the mutual-exclusion protocols of Chapter 2, we may wish to check the first-request-first-in requirement, that the first process to request admission to the critical section is the first process allowed to enter the critical section. As was discussed in Chapter 2, the first-request-first-in requirement can be checked by composing the mutual-exclusion protocol with a monitor and verifying an invariant of the compound module. The introduction of monitors has the disadvantage of increasing the state space of a module. Here we discuss the alternative of enriching the specification language so that module requirements such as first-request-first-in can be formulated without the use of monitors. For this purpose, we present a class of formal languages called *state logics.* While invariants refer to static observation snapshots of reactive modules, the formulas of state logics refer also to dynamic observation sequences. The observation sequences of a module are captured by an *observation structure.*

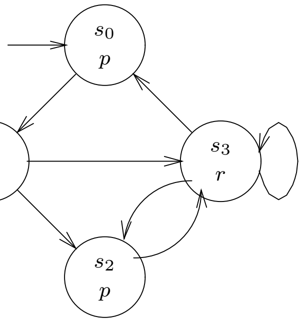


Figure 6.1: Observation structure

6.1.1 Observation Structures

An observation structure is a transition graph whose states are labeled with observations. It is required that there are only finitely many initial states with a given observation, and every state has only finitely many successors with a given observation.

OBSERVATION STRUCTURE

An *observation structure* K consists of (1) a transition graph $(\Sigma, \sigma^I, \rightarrow)$, (2) [the *observation alphabet*] a set A of observations, and (3) [the *observation function*] a function $\langle\langle\cdot\rangle\rangle : \Sigma \rightarrow A$ that maps each state s to an observation $\langle\langle s\rangle\rangle$ such that for every observation $a \in A$, (i) the set $\{s \in \sigma^I \mid \langle\langle s\rangle\rangle = a\}$ is finite, and (ii) for every state $s \in \Sigma$, the set $\{t \in post_G(s) \mid \langle\langle t\rangle\rangle = a\}$ is finite.

Example 6.1 [Observation structure] Figure 6.1 shows an observation structure with the state space $\{s_0, s_1, s_2, s_3\}$ and the observation alphabet $\{p, q, r\}$. ■

We freely attribute properties and derivatives of transition graphs to observation structures. For example, an observation structure inherits the trajectories of the underlying transition graph, and the reachable subgraph of the underlying transition graph gives the reachable substructure of an observation structure.

The observation structure of a module

Every reactive module P defines the transition graph G_P. An observation of P is a valuation to the set $\mathsf{obs}X_P = \mathsf{intf}X_P \cup \mathsf{extl}X_P$ of observable variables of P. The observation alphabet is, then, the set of all observations. The observation function maps a module state s to the projection $s[\mathsf{obs}X_P]$ to the observable variables. There are only finitely many ways to initialize the controlled state of a module: for every valuation t of the external variables $\mathsf{extl}X_P$, there are only finitely many initial states s such that $s[\mathsf{extl}X_P] = t$. It follows that $\sigma_P^I[\mathsf{obs}X_P]$ is finite. Similarly, there are only finitely many ways to update the controlled state of a module: for a state s and a valuation t of the external variables $\mathsf{extl}X_P$, there are only finitely many states u such that $s \rightarrow_P u$ and $u[\mathsf{extl}X_P] = t$. It follows that $post_P(s)[\mathsf{obs}X_P]$ is finite for every state s of the module.

Observation structure of a module

The reactive module P defines the observation structure $K_P = (G_P, \Sigma_{\mathsf{obs}X_P}, \cdot[\mathsf{obs}X_P])$.

Example 6.2 [Observation structure of *Pete*] Recall Peterson's mutual exclusion protocol from Chapter 2. The observable variables for the module *Pete* are the location variables pc_1 and pc_2. The observation alphabet of the observation structure K_{Pete} is

$$\Sigma_{\{pc_1, pc_2\}} = \{inC, reqC, outC\} \times \{inC, reqC, outC\}.$$

The reachable substructure of K_{Pete} has the transition graph of Figure 3.4, and each state is labeled with the value of pc_1 and pc_2. Note that the transition graph of the module $P_1 \| P_2$ is identical to the transition graph of the module $Pete = \textbf{hide } x_1, x_2 \textbf{ in } P_1 \| P_2$, but the observation structures $K_{P_1 \| P_2}$ and K_{Pete} have different observation alphabets, and thus, are nonisomorphic. ■

6.1.2 State Logics

The formulas of state logics are called *state formulas*, because they are interpreted over the states of observation structures; that is, a state formula is true or false in a given state of a given observation structure. Before we study specific state logics, let us define concepts generic to all state logics. A state logic is defined by specifying the rules to build formulas of the logic, and the rules to interpret formulas at states of an observation structure. A state logic Φ consists of two components:

Syntax A set $\mathcal{P}_\Phi$ of formulas. Formulas are defined inductively from atomic formulas using boolean and temporal connectives. The formulas in $\mathcal{P}_\Phi$ are called Φ-*formulas*.

Semantics Given a Φ-formula ϕ, an observation structure K is said to be a ϕ-structure if each observation of K is a valuation for a superset of the variables appearing in the atomic formulas of ϕ. Every Φ-formula ϕ is interpreted over the states of the ϕ-structures. The satisfaction relation $s \models_K \phi$ specifies when the state s of the ϕ-structure K *satisfies* the Φ-formula ϕ.

The satisfaction relation specifies the truth of a formula at a state of an observation structure. An observation structure satisfies a formula if all its initial states do.

SATISFIABILITY

Let ϕ be a state formula, and let K be a ϕ-structure with the state space Σ and the initial region σ^I. The *characteristic ϕ-region* of K is the region

$$[\![\phi]\!]_K \;=\; \{s \in \Sigma \mid s \models_K \phi\}$$

of states in Σ that satisfy the formula ϕ. The observation structure K satisfies the formula ϕ, written $K \models \phi$, if all initial states in σ^I satisfy ϕ; that is, $\sigma^I \subseteq [\![\phi]\!]_K$. The state formula ϕ is *satisfiable* if there is a ϕ-structure that satisfies ϕ. The state formula ϕ is *valid* if all ϕ-structures satisfy ϕ.

The satisfiability problem for a state logic is to decide whether a given formula is satisfiable, and the validity problem is to decide whether a given formula is valid. Let ϕ and ψ be two state formulas. If the observation structure K is both a ϕ-structure and a ψ-structure, we say that K is a (ϕ, ψ)-structure. The state formula ϕ *implies* the state formula ψ if for every (ϕ, ψ)-structure K, the region $[\![\phi]\!]_K$ is a subset of the region $[\![\psi]\!]_K$. The two state formulas ϕ and ψ are *equivalent* if for every (ϕ, ψ)-structure K, the regions $[\![\phi]\!]_K$ and $[\![\psi]\!]_K$ are equal.

Exercise 6.1 {`T2`} [Weak equivalence of state formulas] Let ϕ and ψ be two state formulas. The two state formulas ϕ and ψ are *weakly equivalent* if for every (ϕ, ψ)-structure K, the formula ϕ is K-valid iff the formula ψ is K-valid. Prove that ϕ and ψ are equivalent iff they are weakly equivalent. ■

The model-checking problem

The model-checking problem for Φ asks if ϕ is satisfied by a given ϕ-structure.

MODEL-CHECKING PROBLEM FOR STATE LOGICS

An instance (K, ϕ) of the *model-checking problem* for the state logic Φ consists of (1) a Φ-formula ϕ and (2) a ϕ-structure K. The answer to the model-checking problem (K, ϕ) is YES if K satisfies ϕ, and otherwise NO.

The verification problem for a state logic is to check whether the observation structure of a module satisfies a given formula. It can be reduced, at an exponential cost, to the model-checking problem, similar to the way in which the invariant-verification problem can be reduced to the reachability problem.

Verification problem for state logics

An instance (P, ϕ) of the *verification problem* for the state logic Φ consists of (1) a reactive module P with the set $\mathsf{obs}X_P$ of observable variables, and (2) a Φ-formula ϕ such that the set of variables appearing in the atomic formulas of ϕ is a subset of $\mathsf{obs}X_P$. The answer to the verification problem (P, ϕ) is the answer to the model-checking problem (K_P, ϕ).

6.2 Safe Temporal Logic

Temporal logics extend observation logic with connectives that refer to observation sequences of reactive modules.

6.2.1 Syntax and Semantics of Stl

Safe temporal logic has two temporal connectives: the unary connective *possibly-next*, written $\exists\bigcirc$, and the binary connective *possibly-until*, written $\exists\mathcal{U}$.

Safe temporal logic: Syntax

Safe temporal logic (Stl) is the state logic whose formulas are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \neg\phi \mid \exists\bigcirc\, \phi \mid \phi\exists\mathcal{U}\phi,$$

for atomic formulas p.

The satisfaction of an atomic formula p at a state s depends on the observation of s. The meaning of the boolean connectives is the usual one. Consider two observation predicates p and q for a reactive module P. The state s of the observation structure K_P satisfies the formula $\exists\bigcirc\, p$ if some successor of s satisfies p. The state s satisfies the formula $q\exists\mathcal{U}p$ if there is a source-s trajectory whose states satisfy p or q, and whose sink satisfies p. In other words, the formula $\exists\bigcirc\, p$ asserts that it is possible to execute a single round of the module P so that the observation predicate p becomes true. The formula $q\exists\mathcal{U}p$ asserts that it is possible to execute finitely many rounds of P so that p becomes true, and throughout the execution the observation predicate $p \vee q$ is true.

Safe temporal logic: Semantics

The satisfaction relation for Stl is defined inductively by the following clauses:

$$\begin{array}{lll} s \models_K p & \text{iff} & \langle\!\langle s \rangle\!\rangle \models p; \\ s \models_K \phi \vee \psi & \text{iff} & s \models_K \phi \text{ or } s \models_K \psi; \\ s \models_K \neg\phi & \text{iff} & s \not\models_K \phi; \\ s \models_K \exists\bigcirc\,\phi & \text{iff} & \text{there is a state } t \in \mathit{post}_K(s) \text{ such that } t \models_K \phi; \\ s \models_K \psi \exists\mathcal{U}\phi & \text{iff} & \text{there is a source-}s\text{ trajectory } \overline{s}_{0..m} \text{ of } K \text{ such that} \\ & & (1)\ s_m \models_K \phi \text{ and} \\ & & (2) \text{ for all } 0 \leq i \leq m,\ s_i \models_K \phi \vee \psi. \end{array}$$

where p is an atomic formula, ϕ and ψ are Stl-formulas, K is a (p, ϕ, ψ)-structure, and s is a state of K.

The propositional connectives $\wedge$ (conjunction), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence) can be defined using the connectives $\vee$ (disjunction) and $\neg$ (negation) of state logics, and we freely use the defined connectives as abbreviations.

Remark 6.1 [Until connective] Equivalently, $s \models_K \psi\exists\mathcal{U}\phi$ iff there is a source-s trajectory $\overline{s}_{0..m}$ of K such that (1) $s_m \models_K \phi$ and (2) for all $0 \leq i < m$, $s_i \models_K \psi$. In particular, if $s \models_K \phi$, then $s \models_K \psi\exists\mathcal{U}\phi$. ■

Example 6.3 [Safe temporal logic] Recall the observation structure shown in Figure 6.1. There, $s_0 \models \exists\bigcirc\, q$ and $s_0 \models (p \vee q)\exists\mathcal{U}r$. ■

When all the atomic formulas are propositional formulas, that is, boolean expressions over boolean variables, the temporal logic Stl is called *propositional* Stl.

Defined temporal connectives

Using the connectives $\exists\bigcirc$ and $\exists\mathcal{U}$, we can define additional temporal connectives in Stl:

Inevitably-next	$\forall\bigcirc\,\phi$	for	$\neg\,\exists\bigcirc\,\neg\phi$;
Possibly-eventually	$\exists\Diamond\phi$	for	$\mathit{true}\,\exists\mathcal{U}\phi$;
Inevitably-always	$\forall\Box\phi$	for	$\neg\,\exists\Diamond\neg\phi$;
Inevitably-waiting-for	$\phi\forall\mathcal{W}\psi$	for	$\neg((\neg\psi)\exists\mathcal{U}\neg(\phi \vee \psi))$.

Consider two observation predicates p and q for a reactive module P. The state s of the observation structure K_P satisfies the formula $\forall\bigcirc\, p$ if every successor of s satisfies p. The state s satisfies the formula $\exists\Diamond p$ if some state in the sink region of s satisfies p, and s satisfies $\forall\Box p$ if every state in the sink region of s satisfies p. In other words, the formula $\forall\bigcirc\, p$ asserts that after executing a

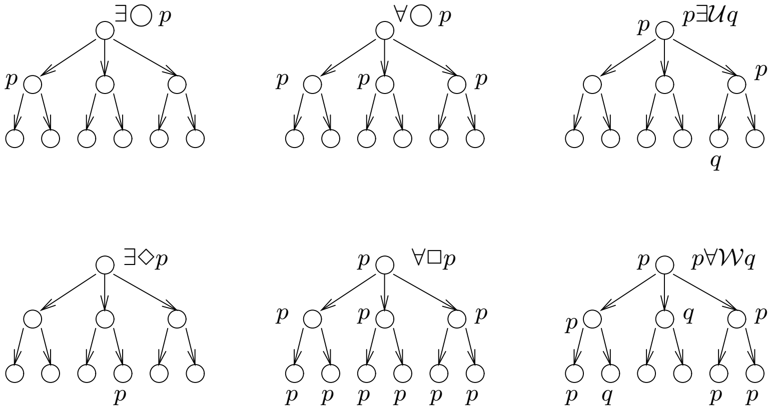


Figure 6.2: The temporal connectives of Stl

single round of the module P the observation predicate p becomes true. The formula $\exists\Diamond p$ asserts that it is possible to execute finitely many rounds of P so that p becomes true, and the formula $\forall\Box p$ asserts that p is an invariant of P. It follows that the invariant-verification problem is a special case of the verification problem for Stl: the invariant-verification problem (P, p) and the Stl-verification problem $(P, \forall\Box p)$ have the same answer.

Exercise 6.2 {T2} [Waiting-for connective] Let ϕ and ψ be two Stl formulas, let K be a (ϕ, ψ)-structure, and let s be a state of K. Prove that $s \models_K \phi\forall\mathcal{W}\psi$ iff for all source-s trajectories $\overline{s}_{0..m}$ of K either for all $0 \leq i \leq m$, $s_i \models_K \phi$; or there is a natural number j with $0 \leq j \leq m$ such that (1) $s_j \models_K \psi$ and (2) for all $0 \leq i < j$, $s_i \models_K \phi$. That is, in every source-s trajectory of K, a state that violates ϕ coincides with or is preceded by a state that satisfies ψ. ■

Figure 6.2 graphically illustrates the requirements that are imposed by the temporal connectives of Stl. When writing Stl formulas, we freely use the defined temporal connectives as abbreviations. We suppress parentheses, assuming that the binary connectives $\exists\mathcal{U}$ and $\forall\mathcal{W}$ associate to the right; that is, we write $\phi\exists\mathcal{U}\psi\exists\mathcal{U}\chi$ for $\phi\exists\mathcal{U}(\psi\exists\mathcal{U}\chi)$.

Exercise 6.3 {T2} [Nested until connectives] Let ϕ, ψ, and χ be three Stl formulas. Let K be a (ϕ, ψ, χ)-structure, and let s be a state of K. Prove that $s \models_K \phi\exists\mathcal{U}(\psi\exists\mathcal{U}\chi)$ iff there is a source-s trajectory $\overline{s}_{0..m}$ of K and a natural number i with $0 \leq i \leq m$ such that (1) $s_m \models_K \chi$, (2) for all $i \leq j < m$, $s_j \models_K \psi$, and (3) for all $0 \leq j < i$, $s_j \models_K \phi$.

Prove or disprove that $(\phi\exists\mathcal{U}\psi)\exists\mathcal{U}\chi$ implies $\phi\exists\mathcal{U}(\psi\exists\mathcal{U}\chi)$, and vice versa. What about $(\phi\forall\mathcal{W}\psi)\forall\mathcal{W}\chi$ vs. $\phi\forall\mathcal{W}(\psi\forall\mathcal{W}\chi)$? ■

Exercise 6.4 {T2} [Dual version of STL] Suppose that the $\text{STL}^{\forall}$ formulas are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \neg\phi \mid \forall\bigcirc\,\phi \mid \phi\forall\mathcal{W}\phi.$$

Define the temporal connectives $\exists\bigcirc$ and $\exists\mathcal{U}$ in this logic; that is, for every STL formula give an equivalent $\text{STL}^{\forall}$ formula. ■

6.2.2 Specifying requirements using STL

Example 6.4 [Mutual exclusion] Recall Peterson's mutual-exclusion protocol from Chapter 2. The mutual-exclusion requirement is specified by the STL formula

$$\forall\Box\,\neg\,(pc_1 = inC \ \wedge \ pc_2 = inC), \qquad (\phi_{mutex})$$

which is equivalent to the STL formula

$$\neg\,\exists\Diamond\,(pc_1 = inC \ \wedge \ pc_2 = inC).$$

The STL formula ϕ_{fifo} specifies the first-request-first-in requirement that if process P_1 attempts to enter the critical section when process P_2 is in its noncritical section , then P_2 cannot overtake P_1 to enter the critical section:

$$\forall\Box((pc_1 = reqC \ \wedge \ pc_2 = outC) \ \rightarrow \ (pc_2 \neq inC)\forall\mathcal{W}(pc_1 = inC)).$$

The STL formula ϕ_{dl_free} specifies the *deadlock-freedom requirement* that if process P_1 attempts to enter the critical section, then there is a trajectory that leads P_1 into its critical section:

$$\forall\Box(pc_1 = reqC \ \rightarrow \ \exists\Diamond(pc_1 = inC)). \qquad (\phi_{dl\_free})$$

Symmetric first-request-first-in and deadlock-freedom requirements can be asserted for process P_2. ■

Exercise 6.5 {P1} [Equal opportunity] Recall Peterson's mutual-exclusion protocol from Chapter 2. Write an STL formula ϕ_{bd_ot} that specifies the equal-opportunity requirement that if process P_1 attempts to enter the critical section when process P_2 is in its noncritical section, then P_2 may enter its critical section at most once before P_1 is allowed to enter its critical section. ■

Example 6.5 [Railroad controller] Recall the module *RailroadSystem* from Chapter 3. The STL-formula

$$\forall\Box\,\neg(\,pc_W = bridge \ \wedge \ pc_E = bridge\,)$$

specifies the safety requirement that both trains should never be simultaneously on the bridge. The STL-formula

$$\forall\Box\,\exists\Diamond\,(\,pc_W \neq bridge \ \wedge \ pc_E \neq bridge\,)$$

specifies the requirement that from every reachable state there exists a trajectory leading to a state in which none of the two trains are on the bride. ■

Event Stl

The logic Stl specifies requirements of trajectories using atomic formulas that are interpreted at states. When a module uses events for communication, it is convenient to use formulas that refer to transitions or pairs of states. For this purpose, we define the state logic Estl. Unlike Stl, it has two sorts of formulas: state formulas that are interpreted with respect to states, and transition formulas that are interpreted with respect to transitions.

Event Stl (Estl) is the state logic whose state formulas are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \neg\phi \mid \varphi\exists\mathcal{U}\varphi.$$

for atomic formulas p and transition formulas φ. The *transition formulas* are generated by the grammar

$$\varphi \ ::= \ \phi \mid \bigcirc\phi \mid \varphi \vee \varphi \mid \neg\varphi$$

for state formulas ϕ. The semantics of the state formulas of Estl is defined as follows:

$$\begin{array}{lll}
s \models_K p & \text{iff} & \langle\!\langle s \rangle\!\rangle \models p; \\
s \models_K \phi \vee \psi & \text{iff} & s \models_K \phi \text{ or } s \models_K \psi; \\
s \models_K \neg\phi & \text{iff} & s \not\models_K \phi; \\
s \models_K \rho\exists\mathcal{U}\varphi & \text{iff} & \text{there is a source-}s\text{ trajectory } \overline{s}_{0..m} \text{ of } K \\
 & & \text{with } m > 0 \text{ such that} \\
 & & (1)\ (s_{m-1}, s_m) \models_K \varphi \text{ and} \\
 & & (2) \text{ for all } 0 < i \leq m,\ (s_{i-1}, s_i) \models_K \varphi \vee \rho.
\end{array}$$

The semantics of the transition formulas of Estl is defined as follows:

$$\begin{array}{lll}
(s,t) \models_K \phi & \text{iff} & s \models_K \phi; \\
(s,t) \models_K \bigcirc\phi & \text{iff} & t \models_K \phi; \\
(s,t) \models_K \varphi \vee \rho & \text{iff} & (s,t) \models_K \varphi \text{ or } (s,t) \models_K \rho; \\
(s,t) \models_K \neg\varphi & \text{iff} & (s,t) \not\models_K \varphi.
\end{array}$$

Thus, the transition formulas can refer to the updated values of variables by using the *next* operator. If x is a boolean variable of the event type, then we use $x?$ as an abbreviation for the transition formula $(x \not\leftrightarrow \bigcirc x)$. Suppose that x and y are two event variables. The following Estl formula asserts that no event x is followed by an event y:

$$\forall\Box(x? \ \rightarrow \ \bigcirc\forall\Box\neg y?).$$

Example 6.6 [Synchronous 3-bit Counter] Recall the synchronous module SCountThree of Example 2.19 that models a 3-bit counter. The desired specification of the counter is that in every update round, if the start command is present ($start' = 1$), then the counter should be reset to 0, and if the increment command is present, then the counter should be incremented by 1 (module 8), and otherwise, the counter should stay unchanged. The following Estl formula expresses the desired update of the bit out_0:

$$\forall\Box \left(\begin{array}{lcl} \bigcirc start & \rightarrow & \neg \bigcirc out \\ \neg \bigcirc start \wedge \bigcirc inc & \rightarrow & out \leftrightarrow \neg \bigcirc out \\ \neg \bigcirc start \wedge \neg \bigcirc inc & \rightarrow & out \leftrightarrow \bigcirc out \end{array} \right).$$

The desired update of the remaining two bits can be specified similarly in Estl. ■

Open modules

Checking existential requirements of an open module is not very meaningful. Existential requirements over external variables are trivially satisfied, while existential requirements over interface variables are not preserved under parallel composition.

Remark 6.2 [Open modules] Let P be a module, and let p and q be boolean expressions over the external variables of P. Then, for every state s of P, $s \models \exists\bigcirc\, p$ and $s \models p\exists\mathcal{U}q$. ■

Exercise 6.6 {T2} [Open modules] Give an example of a module $P \| Q$ and an Stl-formula ϕ such that the answer to the verification problem (P, ϕ) is Yes, while the answer to $(P \| Q, \phi)$ is No. ■

If we restrict ourselves only to the universal formulas, then the compositionality principle holds. Let ∀Stl be the fragment of Stl generated by the grammar

$$\phi \ ::= \ p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall\bigcirc\, \phi \mid \phi\forall\mathcal{W}\phi$$

The logic ∀Stl is not closed under negation.

The parallel composition operation on modules ensures that the projection of a trajectory of a compound module onto the variables of a component is a trajectory of the component. This implies that the compositionality principle holds for ∀Stl.

Proposition 6.1 [Compositionality for ∀Stl] *If the module P satisfies the ∀Stl-formula ϕ, then for every module Q that is compatible with P, the compound module $P \| Q$ satisfies ϕ.*

Proof. Let P and Q be two compatible modules, and let ϕ be a formula ∀Stl. Let $R = P \| Q$. We prove that, for every subformula ψ of ϕ, for all states s of P, if $s \models_P \psi$ then for all states t of R, if $t[X_P] = s$ then $t \models_R \psi$. The proof is by

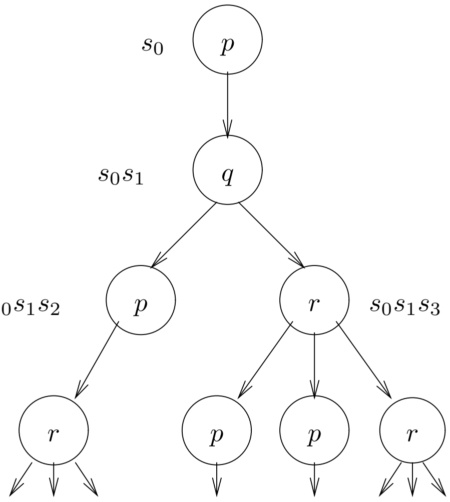


Figure 6.3: Tree Semantics of Stl

induction on the structure of ψ. Consider a state s of P such that $s \models_P \psi$, and let t be a state of R with $t[X_P] = s$. The interesting case is when $\psi = \chi_1 \forall\mathcal{W} \chi_2$.

Consider a source-t trajectory $t_0 \dots t_m$ of R. Then, from the properties of the parallel composition operation, there exists a trajectory $s_0 \dots s_m$ of P such that $s_i = t_i[X_P]$ for all $0 \leq i \leq m$. Since $\overline{s}_{0\dots m}$ is a source-s trajectory of P, and $s \models_P \chi_1 \forall\mathcal{W}\chi_2$, we have either for all $0 \leq i \leq m$, $s_i \models_P \chi_1$; or there is a natural number j with $0 \leq j \leq m$ such that $s_j \models_P \chi_2$ and for all $0 \leq i < j$, $s_i \models_P \chi_1$. From induction hypothesis, it follows that either for all $0 \leq i \leq m$, $t_i \models_R \chi_1$; or there is a natural number j with $0 \leq j \leq m$ such that $t_j \models_R \chi_2$ and for all $0 \leq i < j$, $t_i \models_R \chi_1$. ■

Tree Semantics

The semantics of Stl can, alternatively, be defined using trees. For an observation structure K and a state s of K, the s-rooted tree is obtained by unfolding the source-s trajectories of K into a tree. Figure 6.3 shows the s_0-rooted tree for the transition structure of Figure 6.1.

Formally, for a state s of the observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ the s-rooted K-tree is another transition structure $T_K(s)$:

- The states of $T_K(s)$ are the source-s trajectories of K.
- The only initial state $T_K(s)$ is its root s.

- There is a transition from $\overline{s}_{0\ldots m}$ to $\overline{t}_{0\ldots n}$ if $n = m + 1$ and $s_i = t_i$ for $0 \leq i \leq m$.
- The observation alphabet is A.
- The observation function maps $\overline{s}_{0\ldots m}$ to $\langle\!\langle s_m \rangle\!\rangle$.

Verify that the structure $T_K(s)$ is a tree, that is, every state, except the root s, has a unique predecessor. The formulas of STL can be interpreted over the tree $T_K(s)$ instead the structure K:

Proposition 6.2 [Tree property of STL] *Let ϕ be a* STL*-formula, let K be a ϕ-structure, and let s be a state of K. For every state $\overline{s}_{0\ldots m}$ of the tree structure $T_K(s)$,*

$$\overline{s}_{0\ldots m} \models_{T_K(s)} \phi \text{ iff } s_m \models_K \phi.$$

Exercise 6.7 {T2} [Tree property] Prove Proposition 6.2. ■

Proposition 6.2 implies that the satisfaction nof STL-formulas at a state s of an observation structure depends only upon the substructure of K that is reachable from s, and is insensitive to the unwinding of the structure.

6.3 Model Checking

We are given an STL formula ϕ and a ϕ-structure K, and we are asked to check if all initial states of K satisfy ϕ. For this purpose, we find the characteristic region $[\![\phi]\!]_K$, that is, all states of K that satisfy ϕ. We proceed inductively on the structure of the formula ϕ, by first finding the characteristic regions for the subformulas of ϕ.

Subformulas

The set $Sub(\phi)$ of *subformulas* of the STL formula ϕ is defined inductively:

$Sub(p) = \{p\}$ for an atomic formula p;
$Sub(\psi \vee \chi) = \{\psi \vee \chi\} \cup Sub(\psi) \cup Sub(\chi)$;
$Sub(\neg\psi) = \{\neg\psi\} \cup Sub(\psi)$;
$Sub(\exists\bigcirc\psi) = \{\exists\bigcirc\psi\} \cup Sub(\psi)$;
$Sub(\psi\exists\mathcal{U}\chi) = \{\psi\exists\mathcal{U}\chi\} \cup Sub(\psi) \cup Sub(\chi)$.

Remark 6.3 [Number of subformulas] The STL formula ϕ has at most $|\phi|$ subformulas. ■

Given the input formula ϕ, the model-checking algorithm for STL calls a function $\mathit{OrderedSub}(\phi)$, which returns a queue with the subformulas of ϕ such that a formula appears only after all its subformulas. Assuming a type **form** for formulas:

> **function** *OrderedSub*: **queue of form**
>
> Input: an STL formula ϕ.
> Output: a queue of the formulas in $\mathit{Sub}(\phi)$ such that if $\psi \in \mathit{Sub}(\phi)$ and $\chi \in \mathit{Sub}(\psi)$, then χ precedes ψ in $\mathit{OrderedSub}(\phi)$.

Example 6.7 [Subformula ordering] For example, the function call $\mathit{OrderedSub}((p \wedge q)\exists\mathcal{U}(\neg\exists\bigcirc r))$ may return the queue

$$(p, q, r, p \wedge q, \exists\bigcirc r, \neg\exists\bigcirc r, ((p \wedge q)\exists\mathcal{U}(\neg\exists\bigcirc r))$$

of formulas. ■

Exercise 6.8 {T2} [Computing subformulas] Give an algorithm that, given an STL-formula ϕ with ℓ symbols, computes the function $\mathit{OrderedSub}(\phi)$ in $O(\ell)$ time. ■

6.3.1 Enumerative STL model checking

An enumerative model-checking algorithm computes an enumerative representation of the characteristic region $[\![\phi]\!]_K$ from enumerative representations of the characteristic regions for the subformulas of ϕ.

Assume that the given observation structure K is finite. An enumerative STL model-checking algorithm computes, for each state s of K, the set $\lambda(s) \subseteq \mathit{Sub}(\phi)$ of subformulas of ϕ that are satisfied by s. Initially, $\lambda(s)$ is empty for each state s. Then, all subformulas of ϕ are considered in the order given by the function call $\mathit{OrderedSub}(\phi)$. Consider a subformula ψ of ϕ. For each state s of K, we must decide whether s satisfies ψ, and update $\lambda(s)$ accordingly. Inductively, we know that for every subformula χ of ψ and for each state s, the formula χ belongs to $\lambda(s)$ iff $s \models_K \chi$. The form of ψ leads to various cases. The interesting case occurs when ψ has the form $\chi_1\exists\mathcal{U}\chi_2$. In this case, we define a finite transition graph H:

> The vertices of H are the states Σ of K. A state $s \in \Sigma$ is an initial state of H iff the formula χ_2 belongs to $\lambda(s)$. The graph H has an edge from $s \in \Sigma$ to $t \in \Sigma$ iff (1) K has a transition from t to s and (2) the formula χ_1 belongs to $\lambda(t)$.

The semantics of the possibly-until connective implies that $s \models_K \chi_1\exists\mathcal{U}\chi_2$ iff the vertex s is reachable in the graph H. Consequently, the set of states that satisfy ψ can be computed by a depth-first search in H.

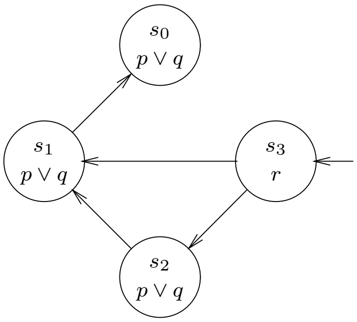


Figure 6.4: Reachability analysis for checking possibly-until

Example 6.8 [Model checking] Consider the observation structure of Figure 6.1, and the possibly-until formula $\phi = (p \vee q)\exists\mathcal{U}r$. Let $\mathit{OrderedSub}(\phi) = \{p, q, r, p \vee q, \phi\}$. First, the formula p is added to the sets $\lambda(s_0)$ and $\lambda(s_2)$. Then, the formula q is added to the set $\lambda(s_1)$. Then, the formula r is added to the set $\lambda(s_3)$. Then, the formula $p \vee q$ is added to the sets $\lambda(s_0)$, $\lambda(s_1)$, and $\lambda(s_2)$. To evaluate the truth of ϕ, consider the transition graph of Figure 6.4 with the initial state s_3. All the states are reachable, implying that the formula ϕ is satisfied in all the states, and hence, for every state s, ϕ is added to the set $\lambda(s)$. ■

The Stl model-checking algorithm shown in Figure 6.5 considers only the reachable substructure of the input structure K. For this purpose, the algorithm calls the function *Reach*(K), which returns a queue with the reachable states of K. The function *Reach* can be implemented using the techniques from Chapter 3. The abstract type for the input structure K supports also the operations *InitQueue*, *PreQueue*, and *PostQueue* (see Chapter 3). Given a state s of K, each function call $\lambda(s)$ returns a set of formulas. The abstract type **set** for the formula sets $\lambda(s)$: **set of form** and the state set σ : **set of state** supports the operations *EmptySet*, *Insert*, and *IsMember*. The satisfaction of atomic formulas is checked by the function *AtomicCheck*. For an observation structure K, a state s of K, and an atomic formula p, *AtomicCheck*(s, p, K) returns *true* iff $s \models_K p$. Checking of the possibly-until formulas employs a depth-first search using the stack τ and the set σ that stores the states visited by the search.

Lemma 6.1 [Correctness of enumerative model checking] *Let ϕ be an* Stl *formula, and let K be a ϕ-structure with a finite reachable substructure. Upon termination of Algorithm 6.1, for every subformula ψ of ϕ and each state s of K, $\psi \in \lambda(s)$ iff $s \models_K \psi$.*

Exercise 6.9 {P2} [Mutual exclusion] Consider the module *Pete* from Chapter 2, and the Stl-formula ϕ_{fifo} of Example 6.4. Execute Algorithm 6.1 on the

Algorithm 6.1 [Enumerative Stl Model Checking]

Input: an Stl formula ϕ, and a ϕ-structure K whose reachable substructure is finite.
Output: the answer to the model-checking problem (K, ϕ).

```
$\sigma^R := Reach(K)$;
foreach $s$ in $\sigma^R$ do $\lambda(s) := EmptySet$ od;
foreach $\psi$ in $OrderedSub(\phi)$ do
  case $\psi = p$ for an atomic formula $p$:
    foreach $s$ in $\sigma^R$ do
      if $AtomicCheck(s, p, K)$ then $\lambda(s) := Insert(p, \lambda(s))$ fi
      od
  case $\psi = \chi_1 \lor \chi_2$:
    foreach $s$ in $\sigma^R$ do
      if $IsMember(\chi_1, \lambda(s))$ or $IsMember(\chi_2, \lambda(s))$ then
        $\lambda(s) := Insert(\psi, \lambda(s))$ fi
      od
  case $\psi = \neg\chi$:
    foreach $s$ in $\sigma^R$ do
      if not $IsMember(\chi, \lambda(s))$ then $\lambda(s) := Insert(\psi, \lambda(s))$ fi
      od
  case $\psi = \exists\bigcirc \chi$:
    foreach $s$ in $\sigma^R$ do
      foreach $t$ in $PostQueue(s, K)$ do
        if $IsMember(\chi, \lambda(t))$ then $\lambda(s) := Insert(\psi, \lambda(s))$ fi
        od
      od
  case $\psi = \chi_1 \exists\mathcal{U} \chi_2$:
    $\sigma := EmptySet$; $\tau := EmptyStack$;
    foreach $s$ in $\sigma^R$ do
      if $IsMember(\chi_2, \lambda(s))$ and not $IsMember(s, \sigma)$ then
        $\tau := Push(s, \tau)$; $\sigma := Insert(s, \sigma)$ fi;
      while not $EmptySet(\tau)$ do
        $t := Top(\tau)$; $\tau := Pop(\tau)$; $\lambda(t) := Insert(\psi, \lambda(t))$;
        foreach $u$ in $PreQueue(t, K)$ do
          if $IsMember(\chi_1, \lambda(u))$ and not $IsMember(u, \sigma)$ then
            $\tau := Push(u, \tau)$; $\sigma := Insert(u, \sigma)$ fi
          od
        od
      od
    end case
  od;
foreach $s$ in $InitQueue(K)$ do
  if not $IsMember(\phi, \lambda(s))$ then return No fi
  od;
return Yes.
```

Figure 6.5: Enumerative Stl model checking

input (K_{Pete}, ϕ_{fifo}), and establish that the answer to the verification problem $(Pete, \phi_{fifo})$ is YES. ■

If the observation structure K is finite, then K can be represented by a record $\{K\}_e$ with two components of type **enumgraph** and **array**[**state**] **of obs**, where **obs** is the type of observations. The second component is redundant if the observation structure K is defined by a reactive module. In this case, the observation of each state can be obtained from the state itself by ignoring the values of the private variables. The *enumerative* structure representation $\{K\}_e$ supports the operations *InitQueue*, *PreQueue*, and *PostQueue* in constant time, and *Reach* can be implemented in time proportional to the number of transitions of K (see Chapter 3). The function λ can be implemented in constant time using an array λ: **array**[**state**] **of set of form**, where **form** ranges over the subformulas of ϕ. If the abstract type **set of** is represented by a boolean array **array**[ ] **of** $\mathbb{B}$, then the operations *Insert* and *IsMember* require constant time, and the operation *EmptySet* requires time proportional to the number of elements in . This representation leads to linear running time of Algorithm 6.1.

Theorem 6.1 [STL model checking] *Let ϕ be an* STL *formula with ℓ symbols, and let K be a finite ϕ-structure with n states and m transitions. Suppose that every call to the function AtomicCheck requires constant time. Given the input ϕ and $\{K\}_e$, Algorithm 6.1 solves the model-checking problem (K, ϕ) in $O(\ell \cdot (n + m))$ time.*

Remark 6.4 [Space complexity of STL model checking] Let ϕ be an STL formula with ℓ symbols, and let K be a finite ϕ-structure with n states. Algorithm 6.1 requires $O(\ell \cdot n)$ space. It is possible to solve the model checking problem in recursively top-down manner to save space. In particular, there is a nondeterministic algorithm that requires space $O(\ell \cdot \log\ n)$ space. If the STL-formula ϕ is small, that is, bounded by a constant, then the complexity class of the model checking problem (K, ϕ) is NLOGSPACE. ■

The STL-verification problem (P, ϕ), for a finite module P and an STL formula ϕ, can be solved by first constructing the enumerative structure representation $\{K_P\}_e$, and then applying Algorithm 6.1. Since the number of states of K_P may be exponentially larger than the description of P, the resulting cost for STL verification is exponential. This cost is unavoidable, because already the propositional STL-verification problem is PSPACE-hard (the propositional invariant-verification problem (P, p) and the STL-verification problem $(P, \forall\Box p)$ have the same answer, and the former was shown to be PSPACE-hard in Chapter 3). In Section 3.2.4, we considered two space optimizations for invariant verification, using on-the-fly representations and using only the latched variables. Both these techniques are useful for improving efficiency of STL model checking.

Exercise 6.10 {T3} [Stl verification in PSPACE] Prove that the Stl-verification problem is in PSPACE. ■

Exercise 6.11 {T3} [Reduced observation structure] Recall the definition of the reduced transition graph of a module. Define the reduced observation structure of a module and use it for an improved solution of the Stl-verification problem, along the lines of Theorem 3.4. ■

6.3.2 Symbolic Stl model checking

A symbolic model-checking algorithm computes a symbolic representation of the characteristic region $[\![\phi]\!]_K$ from symbolic representations of the characteristic regions for the subformulas of ϕ. The symbolic Stl model-checking algorithm shown in Figure 6.6 assumes that the symbolic structure representation supports the operations *InitReg* and *PreReg*, and the symbolic region representation supports, in addition to the operations $\cup$, $\cap$, and $\subseteq$, also the set difference operation $\backslash$. An Stl-verification problem (P, ϕ) can be solved by first constructing a symbolic representation of the observation structure K_P, and then applying Algorithm 6.2.

Consider the propositional Stl-verification problem (P, ϕ). The symbolic representation of the observation structure K_P consists of (1) the symbolic representation of the transition graph G_P (which consists of the initial predicate q^I over X_P and the transition predicate q^T over $X_P \cup X'_P$), and (2) the set $\mathsf{obs}X_P$ of observable variables. Binary decision diagrams are suitable for solving the propositional Stl-verification problem. All the heuristics considered in Section 5.2.4 to improve the efficiency of BDD-based representations are useful in Stl-verification.

Exercise 6.12 {T3} [Symbolic region difference] Write an algorithm that computes, given the BDD representation of two propositional formulas p and q, the BDD representation of the difference $[\![p]\!]\backslash[\![q]\!]$. What is time complexity of the algorithm? ■

Exercise 6.13 {T3} [Event Stl] Write a symbolic model-checking algorithm for Estl that computes characteristic regions only over the latched variables. ■

6.4 The Distinguishing Power of Stl

The partition-refinement algorithms presented in Chapter 4 can be used to reduce the size of an observation structure, while retaining the ability of computing the characteristic regions for Stl formulas. For this purpose, we need to understand when two states of an observation structure satisfy the same Stl formulas.

Algorithm 6.2 [Symbolic STL model checking]

Input: an STL formula ϕ, and a ϕ-structure K.
Output: the answer to the model-checking problem (K, ϕ).

```
foreach $\psi$ in $\mathit{OrderedSub}(\phi)$ do
  case $\psi = p$ for an atomic formula $p$: $[\![\psi]\!] = [\![p]\!]$
  case $\psi = \chi_1 \vee \chi_2$: $[\![\psi]\!] = [\![\chi_1]\!] \cup [\![\chi_2]\!]$
  case $\psi = \neg\chi$: $[\![\psi]\!] = [\![\mathit{true}]\!] \backslash [\![\chi]\!]$
  case $\psi = \exists\bigcirc\chi$: $[\![\psi]\!] = \mathit{PreReg}([\![\chi]\!], K)$
  case $\psi = \chi_1 \exists\mathcal{U} \chi_2$:
    $\sigma := [\![\mathit{false}]\!]$;
    $\tau := [\![\chi_2]\!]$;
    while $\tau \not\subseteq \sigma$ do
      $\sigma := \sigma \cup \tau$;
      $\tau := \mathit{PreReg}(\sigma, K) \cap [\![\chi_1]\!]$
      od
  end case
  od;
if $\mathit{InitReg}(K) \subseteq [\![\phi]\!]$ then return YES else return NO.
```

Figure 6.6: Symbolic STL model checking

6.4.1 State Equivalences

A *state equivalence* $\simeq$ is a family of relations which contains for each observation structure K an equivalence $\simeq_K$ on the state space of K (that is, a partition of the state-space of K). Here are three examples of state equivalences:

1. Within each observation structure K, *state equality* $=$ distinguishes any two different states: for every state s of K, $s/_{=_K}$ is the singleton set $\{s\}$.

2. Within each observation structure K, *observational equivalence* $\approx$ distinguishes any two states with different observations: if $\langle\!\langle \cdot \rangle\!\rangle$ is the observation function of K, then two states s and t of K are *observationally equivalent*, denoted $s \approx_K t$, if $\langle\!\langle s \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle$.

3. Within each observation structure K, *universal equivalence* $\simeq^U$ does not distinguish any two states: if Σ is the state space of K, then for every state s of K, $s/_{\simeq^U}$ equals Σ.

The refinement relation on equivalences induces a preorder on state equivalences. Let $\simeq^1$ and $\simeq^2$ be two state equivalences. The state equivalence $\simeq^1$ is *as distinguishing as* the state equivalence $\simeq^2$, written $\simeq^2 \sqsubseteq \simeq^1$, if for all observation structures K, the equivalence $\simeq^1_K$ refines the equivalence $\simeq^2_K$. The state equivalence $\simeq^1$ is *more distinguishing than* $\simeq^2$, if $\simeq^2 \sqsubseteq \simeq^1$ and $\simeq^1 \not\sqsubseteq \simeq^2$. The two state equivalences $\simeq^1$ and $\simeq^2$ are *equally distinguishing*, if $\simeq^1 \sqsubseteq \simeq^2$ and $\simeq^2 \sqsubseteq \simeq^1$. The two state equivalences $\simeq^1$ and $\simeq^2$ are *incomparable* if $\simeq^1 \not\sqsubseteq \simeq^2$ and $\simeq^2 \not\sqsubseteq \simeq^1$.

Remark 6.5 [State equality and universal equivalence] Let $\simeq$ be a state equivalence. The state equivalence $\simeq$ is as distinguishing as universal equivalence, and state equality is as distinguishing as $\simeq$. In other words, the preorder $\sqsubseteq$ has a bottom, the universal equivalence $\simeq^U$, and a top, the state equality $=$. ■

6.4.2 Bisimilarity

Since observational equivalence refers to static observation snapshots for distinguishing two states, more distinctions can be made by referring also to dynamic observation sequences. Bisimilarity is such a state equivalence which, while less distinguishing than state equality, is more distinguishing than observational equivalence.

Bisimilarity

Let K be an observation structure. The coarsest stable refinement $\simeq^B_K = \min_K(\approx_K)$ of observational equivalence and the induced state equivalence $\simeq^B$ are called *bisimilarity*.

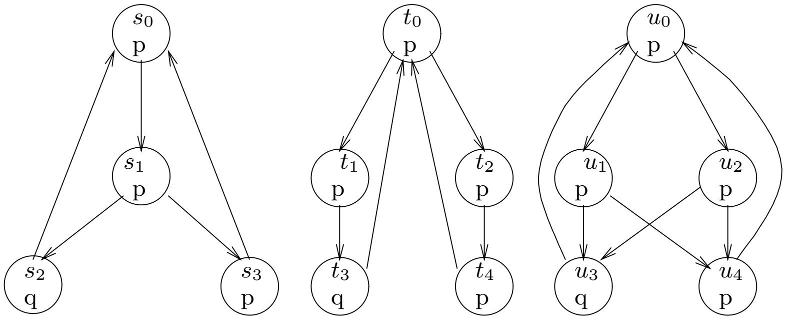


Figure 6.7: Bisimilarity game

Remark 6.6 [Alternative definition of bisimilarity] Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. The equivalence $\cong$ on the states of K is a *bisimulation* of K if (1) the partition $\Sigma/_{\cong}$ is a stable partition of K and (2) $\cong$ refines the observational equivalence $\approx_K$. Thus, for all states s and t of K, if $s \cong t$ then

(1) $\langle\!\langle s\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle$;
(2) if $s \rightarrow s'$, then there is a state t' such that $t \rightarrow t'$ and $s' \cong t'$;
(3) if $t \rightarrow t'$, then there is a state s' such that $s \rightarrow s'$ and $s' \cong t'$.

Two states s and t of K are bisimilar iff there is a bisimulation $\cong$ of K such that $s \cong t$. ■

It follows that bisimilarity can be characterized game-theoretically. Consider the following two-player game on the graph of the observation structure K. Player I, the *protagonist*, attempts to show that two given states s and t are bisimilar, while Player II, the *adversary*, tries to distinguish the two states. If the two given states have different observations, then the adversary wins immediately. Throughout the game, there are two active states; initially s and t are active. Each move of the game consists of two parts —a move by the adversary followed by a move of the protagonist. The adversary picks one of the two active states and replaces it by one of its successors, say s'; the protagonist, then, must match the move of the adversary by replacing the other active state with one of its successors t' such that $\langle\!\langle s'\rangle\!\rangle = \langle\!\langle t'\rangle\!\rangle$. If the protagonist cannot match a move of the adversary, then the adversary wins the game. The two initial states s and t are bisimilar iff the adversary does not have a winning strategy; that is, all of possible moves of the adversary can perpetually be matched by the protagonist.

Example 6.9 [Bisimilarity game] Consider the observation structure shown in Figure 6.7. The two states s_0 and t_0 are not bisimilar. To see this using the bisimilarity game, consider the following strategy for the adversary. The adversary chooses s_0 and moves to s_1. If, in response, the protagonist decides to move from t_0 to t_1, then the adversary moves from s_1 to s_3, and the protagonist cannot match this move (because no transition from t_1 leads to a state with observation p). Similarly, if the protagonist decides to move from t_0 to t_2, then the adversary moves from s_1 to s_2, and the protagonist cannot match this move either. So the adversary has a winning strategy in the bisimilarity game, which implies that the two states s_0 and t_0 are not bisimilar. By contrast, it is easy to check that the two states s_0 and u_0 are bisimilar. ■

Exercise 6.14 {T3} [Fixpoint view of bisimilarity] Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. Given a binary relation $\cong \subseteq \Sigma^2$, we define the binary relation $f(\cong) \subseteq \Sigma^2$ such that for all states s and t of K, $(s,t) \in f(\cong)$ iff

(1) $s \cong t$;
(2) if $s \rightarrow s'$, then there is a state t' such that $t \rightarrow t'$ and $s' \cong t'$;
(3) if $t \rightarrow t'$, then there is a state s' such that $s \rightarrow s'$ and $s' \cong t'$.

First, prove that f is a monotonic function on the complete partial order of the equivalences on Σ under refinement (i.e., if $\cong_1 \preceq \cong_2$ for two equivalences $\cong_1$ and $\cong_2$ on Σ, then $f(\cong_1) \preceq f(\cong_2)$). Second, prove the binary relation $\cong \subseteq \Sigma^2$ is a bisimulation of K iff (1) $\cong$ is an equivalence that refines the observational equivalence $\approx$ and (2) $\cong$ is a fixpoint of f. Third, conclude that bisimilarity $\simeq^B_K$ is the greatest fixpoint of f. ■

An infinite hierarchy of state equivalences

Bisimilarity —a family of coarsest stable refinements— is defined to be the least distinguishing state equivalence in the set of state equivalences whose constituents are stable refinements of observational equivalence. Hence, bisimilarity is defined "from below," in terms of more distinguishing state equivalences. Alternatively, bisimilarity can be defined "from above," as the limit of a sequence of less distinguishing state equivalences. Intuitively, two states are *i-step bisimilar*, written $\approx^i$ for a natural number i, if in the bisimilarity game the adversary has no winning strategy that requires at most i moves. Then, two states are bisimilar if they are i-step bisimilar for all natural numbers i.

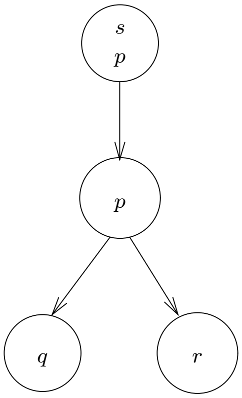


Figure 6.8: The observation structure K_1

i-STEP BISIMILARITY

The state equivalences $\approx^i$, called *i-step bisimilarity* for each natural number i, are defined inductively. The state equivalence $\approx^0$ coincides with observational equivalence; that is, $\approx^0 = \approx$. For each natural number i, for every observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\langle\cdot\rangle\rangle)$, and for all states s and t of K, let $s \approx_K^{i+1} t$ iff

(1) $\langle\langle s\rangle\rangle = \langle\langle t\rangle\rangle$;
(2) if $s \rightarrow s'$, then there is a state t' such that $t \rightarrow t'$ and $s' \approx_K^i t'$;
(3) if $t \rightarrow t'$, then there is a state s' such that $s \rightarrow s'$ and $s' \approx_K^i t'$.

Remark 6.7 [Weak hierarchy] For each natural number i, the state equivalence $\approx^{i+1}$ is as distinguishing as $\approx^i$, and bisimilarity is as distinguishing as $\approx^i$. ■

Exercise 6.15 {T3} [Computation of i-step bisimilarity] Write an algorithm that, given an observation structure K and a natural number i, computes the partition $K/_{\approx_K^i}$. What is the asymptotic running time of your algorithm for finite input structures? ■

The hierarchy of state equivalences given by i-step bisimilarity is strict and converges towards bisimilarity.

Proposition 6.3 [Strict hierarchy] *For each natural number i, the state equivalence $\approx^{i+1}$ is more distinguishing than the state equivalence $\approx^i$.*

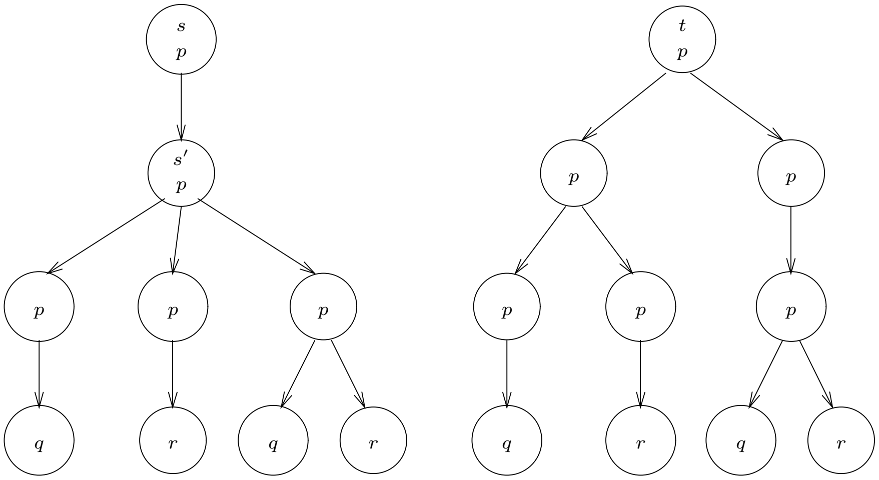


Figure 6.9: The observation structure K_2

Proof. The proof is by induction on i. We only give the base case and indicate the idea behind the inductive step.

For the base case, consider the observation structure K_1 shown in Figure 6.8. To distinguish the two states s and t in the bisimilarity game, the adversary needs a single move (the adversary chooses the successor of s). It follows that $s \approx^0_{K_1} t$ and $s \not\approx^1_{K_1} t$; that is, $\approx^1$ is more distinguishing than $\approx^0$.

For the idea behind the inductive step, consider the observation structure K_2 shown in Figure 6.9. To distinguish the two states s and t, the adversary needs two moves (with its first move, the adversary chooses the left successor of s; with its second move, it chooses the rightmost successor of s'). It follows that $s \approx^1_{K_2} t$ and $s \not\approx^2_{K_2} t$; that is, $\approx^2$ is more distinguishing than $\approx^1$. ■

Exercise 6.16 {T3} [Strict hierarchy] Give a complete proof of Proposition 6.3. Use only finite observation structures to distinguish $\approx^{i+1}$ from $\approx^i$. ■

The next proposition gives establishes that bisimilarity coincides with the inductive definition.

Proposition 6.4 [Alternative definition of bisimilarity] *Bisimilarity* $\simeq^B$ *equals the intersection* $\bigcup i \in \ . \approx^i$ *of the* i*-step bisimilarity equivalences.*

Proof. We show that the function f from Exercise 6.14 is $\bigcap$-continuous; that is, given an observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ and a sequence $\cong_0, \cong_1$ $, \cong_2, \ldots$ of equivalences on Σ, if $\approx\succeq\cong_0\succeq\cong_1\succeq\cong_2\succeq\cdots$, then $f(\cap i \in \ . \cong_i)$ equals $\cap i \in \ . f(\cong_i)$. The proposition follows from the Kleene fixpoint theorem.

By the monotonicity of f (Exercise 6.14), $f(\cap i \in \ \ . \cong_i) \preceq (\cap i \in \ \ . f(\cong_i))$. Conversely, consider two states s and t of K such that for all natural numbers i, $(s,t) \in f(\cong_i)$. Let $s \to s'$. Then for all natural numbers i, there is a state t_i with $t \to t_i$ such that $s' \cong_i t_i$. Since all the equivalences $\cong_i$ refine the propositional equivalence $\approx$, $s' \approx t_i$ for all i. That is, all the state t_i have identical observations. Since $\langle\langle post(s) \rangle\rangle$ is finite, there is a state t' with $t \to t'$ such that $s' \cong_i t'$ for infinitely many natural numbers i. Since $i \leq j$ implies $\cong_i \succeq \cong_j$, $s' \cong_i t'$ for all natural numbers i. Therefore, $(s,t) \in f(\cap i \in \ \ . \cong_i)$. ■

Remark 6.8 [Finite branching] The requirement that every state has only finitely many successors per observation is essential for the validity of Proposition 6.4. Otherwise the function f from Exercise 6.14 is not necessarily $\bigcap$-continuous, and $\simeq_K^B$ properly refines $\bigcup i \in \ \ . \approx^i$. ■

6.4.3 Requirement-preserving Equivalences

Every state logic induces a state equivalence, namely, the state equivalence that distinguishes any two states iff there is a state formula that is satisfied by one state but not by the other state.

Φ-EQUIVALENCE

Let Φ be a state logic and let K be an observation structure. The two states s and t of K are Φ-*equivalent*, denoted $s \simeq_K^\Phi t$, if for all Φ-formulas ϕ such that K is a ϕ-structure, $s \models_K \phi$ iff $t \models_K \phi$. The equivalence $\simeq_K^\Phi$ and the induced state equivalence $\simeq^\Phi$ are called Φ-*equivalence*.

The state equivalences induced by state logics allow us to compare the distinguishing power of two state logics. Let Φ and Ψ be two state logics. The state logic Φ is *as distinguishing as* the state logic Ψ if $\simeq^\Psi \ \sqsubseteq \ \simeq^\Phi$, etc. In other words, the state logic Φ is as distinguishing as the state logic Ψ, if whenever some Ψ-formula distinguishes between two states, there exists some Φ-formula that distinguishes between those two states: for every observation structure K, and for every two states s and t of K, if there exists a Ψ-formula ϕ such that $s \models_K \phi$ but $t \not\models_K \phi$, then there there exists a Φ-formula ψ such that $s \models_K \psi$ but $t \not\models_K \psi$.

Abstraction

Consider the model checking problem (K, ϕ) for a state logic Φ. The notion of abstraction defines the conditions under which computing the characteristic region of K for ϕ can be reduced to computing the characteristic region of a quotient structure of K.

Abstraction

The state logic Φ *admits abstraction* if for every state equivalence $\simeq$ as distinguishing as $\simeq^{\Phi}$, for every Φ-formula ϕ, and for every ϕ-structure K, the characteristic region $\llbracket\phi\rrbracket_K$ is $\bigcup\llbracket\phi\rrbracket_{K/_{\simeq}}$. If Φ admits abstraction and $\simeq^{\Phi}$ $\sqsubseteq$ $\simeq$, then $\simeq$ is called an *abstract semantics* for Φ; if Φ admits abstraction, then $\simeq^{\Phi}$ is the *fully abstract semantics* for Φ.

Let Φ be a state logic, let ϕ be a Φ-formula, and let K be a ϕ-structure. Suppose $\simeq$ is an abstract semantics for Φ. Then, any two states that are $\simeq$-equivalent satisfy the same set of Φ-formulas. Hence, instead of performing model checking on the structure K, we can perform model checking on the quotient structure $K/_{\simeq_K}$. Since the logic Φ admits abstraction, we know that a state s of K satisfies a Φ-formula ϕ iff the $\simeq$-equivalence class containing s satisfies ϕ in the quotient structure. Thus, the model-checking problems (K, ϕ) and $(K/_{\simeq_K}, \phi)$ have the same answer. The latter problem may be much simpler, because the state space of the quotient structure $K/_{\simeq_K}$ may be much smaller than the state space of K.

All the state logics that we study, including the logic Stl, admit abstraction. However, it is possible to define operators whose truth is not preserved by quotients.

Proposition 6.5 [Stl abstraction] Stl *admits abstraction.*

Exercise 6.17 {`T2`} [Stl abstraction] Prove Proposition 6.5. ■

Example 6.10 [Abstraction] Consider the state logic Φ^{na} whose state formulas are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \neg\phi \mid [\mathit{even}]\phi.$$

The semantics of the operator $[\mathit{even}]$ is defined by the clause

$$s \models_K [\mathit{even}]\phi \quad \text{iff} \quad \text{the characteristic region } \llbracket\phi\rrbracket_K \text{ has even cardinality.}$$

For instance, $[\mathit{even}]p$ holds in a state of K iff even number of states of K satisfy p. The logic Φ^{na} does not admit abstraction. Verify that the observational equivalence $\approx$ is the Φ^{na}-equivalence, i.e. no Φ^{na}-formula distinguishes between two states with identical observations. Consider an observation structure K with states Σ and observations A. The characteristic region $\llbracket[\mathit{even}]\mathit{true}\rrbracket_K$ equals Σ if Σ has even cardinality and $\emptyset$ otherwise. The quotient $K/_{\approx}$ has one state per observation of K. Consequently, $\bigcup\llbracket[\mathit{even}]\mathit{true}\rrbracket_{K/_{\approx}}$ equals Σ if A has even cardinality and $\emptyset$ otherwise. Since $|\Sigma|$ may be even while $|A|$ is not, Φ^{na} does not admit abstraction. Consequently, we cannot use quotients to solve the model checking problem for Φ^{na}. ■

Stl **equivalence**

To use Proposition 6.5 for Stl model checking, we need to determine abstract semantics for Stl. The next proposition asserts that no Stl-formula can distinguish between two bisimilar states.

Proposition 6.6 [Stl abstraction] *Bisimilarity is an abstract semantics for* Stl.

Proof. Consider an observation structure K. The bisimilarity partition $\simeq^B_K$ is the coarsest stable refinement $\min(\approx_K)$ of the observational equivalence. We need to prove that two bisimilar states satisfy the same set of Stl-formulas. Let ϕ be a formula of Stl. The proof is by induction on the structure of ϕ. Consider two states s and t such that $s \simeq^B t$. We want to prove that $s \models \phi$ iff $t \models \phi$.

The base case is when ϕ is an observation predicate. Since $\simeq^B$ refines the observational equivalence, we know that s and t have identical observations. Hence, s and t satisfy the same set of observation predicates. The inductive case corresponding to logical connectives is straightforward.

Consider the case $\phi = \psi \exists\mathcal{U} \chi$. By inductive hypothesis, bisimilar states agree on the truth of ψ and χ. Suppose $s \models \phi$. Then, there is a source-s trajectory $\overline{s}_{0..m}$ such that $s_m \models \chi$, and $s_i \models \psi$ for $0 \leq i < m$. Since the partition $\simeq^B$ is stable, starting with state $t_0 = t$, we can find states $t_1, t_2, \ldots, t_m$ such that each $t_i \simeq^B s_i$ for $0 \leq i \leq m$, and $t_0 t_1 \ldots t_m$ is a trajectory of K. From the inductive hypothesis, $t_m \models \chi$ and $t_i \models \psi$ for $0 \leq i < m$. This implies that $t \models \phi$. From symmetry, $s \models \phi$ iff $t \models \phi$.

The remaining case $\phi = \exists\bigcirc\, \psi$ is left for the reader to verify. ■

This suggests that for Stl model checking, it suffices to construct quotients with respect to bisimilarity. Given an observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$, we first consider the observational equivalence $\approx_K$ over the states Σ. The next step is to construct the coarsest stable refinement $\min(\approx_K)$ using one of the algorithms from Chapter 4. This yields the bisimilarity equivalence $\simeq^B_K$ and the minimal quotient $K/_{\simeq^B}$. Then, Stl specifications for the observation structure K can be checked by model checking over the quotient structure $K/_{\simeq^B}$.

We proceed to establish that bisimilarity is a fully abstract semantics of Stl. In fact, if bisimilarity distinguishes two states of an observation structure, then the two states can be distinguished by a Stl formula that employs only the next connective. For instance, in Figure 6.7, the nonbisimilar states s and t can be distinguished by the Stl formula $\exists\bigcirc\, (\exists\bigcirc\, q \wedge \exists\bigcirc\, r)$. The fragment Stl$^{\bigcirc}$ of Stl contains those formulas of Stl that do not contain the until connective $\exists\mathcal{U}$. The fragment Stl$^{\mathcal{U}}$ of Stl contains those formulas of Stl that do not contain the next connective $\exists\bigcirc$. Then Stl is as distinguishing as Stl$^{\bigcirc}$, and also as distinguishing as Stl$^{\mathcal{U}}$.

Proposition 6.7 [STL full abstraction] *The equivalence induced by* STL$^{\bigcirc}$ *coincides with the bisimilarity* $\simeq^B$.

Proof. Consider an observation structure K. We wish to prove that whenever two states s and t of K belong to different equivalence classes of $\simeq^B_K$, there exists a formula ϕ of STL$^{\bigcirc}$ such that $s \models_K \phi$ and $t \not\models_K \phi$. We prove that, for every natural number i, for every equivalence class σ of the i-step bisimilarity $\approx^i_K$, there exists a formula ϕ_σ of STL$^{\bigcirc}$ such that $[\![\phi_\sigma]\!]_K = \sigma$.

Base case $i = 0$: for an equivalence class σ of the propositional equivalence $\approx_K$, the formula ϕ_σ is the observation of σ.

Inductive case $i = k+1$: Let σ be an equivalence class of $\approx^{k+1}_K$. There are only finitely many equivalence classes τ of the partition $\approx^k_K$ such that $s \rightarrow t$ for some $s \in \sigma$ and $t \in \tau$. Then, choose

$$\phi_\sigma \;=\; \bigwedge_{\{\tau \in \approx^k \mid \sigma \rightarrow \tau\}} \exists\bigcirc\, \phi_\tau \;\wedge\; \forall\bigcirc \bigvee_{\{\tau \in \approx^k \mid \sigma \rightarrow \tau\}} \phi_\tau .$$

The reader should verify that the characteristic region $[\![\phi_\sigma]\!]_K$ equals σ. ■

Exercise 6.18 {T3} [Event STL] Does ESTL admit abstraction? What is the state equivalence induced by ESTL? Prove your answers. ■

6.4.4 Stutter-insensitive Equivalences

A reactive module *stutters* when its observable state stays unchanged. An asynchronous module may stutter in every update round. If the number of rounds for which a module stutters before updating its observation is irrelevant, then many such rounds can be combined into a single transition. This suggests defining a closure operation on observation structures that adds a transition from the state s to the state t whenever there is a trajectory from s to t along which the observation stays unchanged.

STUTTER CLOSURE

Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. For two states s and t of K, let $s \rightarrow^S t$ if there is an source-s K-trajectory $\overline{s}_{0..m}$ such that (1) for all $0 \leq i < m$, $\langle\!\langle s_i \rangle\!\rangle = \langle\!\langle s \rangle\!\rangle$, and (2) $s_m = t$. The relation $\rightarrow^S$ is called the *stutter-closed transition relation* of K. The *stutter closure* K^S is the observation structure $(\Sigma, \sigma^I, \rightarrow^S, A, \langle\!\langle\cdot\rangle\!\rangle)$.

Remark 6.9 [Stutter closure] The stutter-closed transition relation is reflexive. The reachable region of the stutter closure of the observation structure K coincides with the reachable region of K. ■

Exercise 6.19 {T2} [Stutter closure] Let K be a finite observation structure with n states. Give an $O(n^3)$ algorithm that computes the stutter closure K^S. ■

Stutter closure operation extends to state equivalences also. For instance, two check whether states s and t of an observation structure K are equivalent according to the stutter-closure of bisimilarity, we first compute the stutter-closure of K and then check if the two states s and t are bisimilar.

> STUTTER CLOSURE OF STATE EQUIVALENCES
>
> Let $\simeq$ be a state equivalence, and let K be an observation structure. For two states s and t of K, $s \cong_K t$, for the stutter closure $\cong$ of $\simeq$, if $s \simeq_{K^S} t$. The induced state equivalence $\cong$ is called the *stutter closure* of $\simeq$. The state equivalence $\simeq$ is *stutter-insensitive* if $\simeq = \cong$.

Thus, the equivalence $\cong_K$ is same as the equivalence $\simeq_{K^S}$. For instance, for the structure $K_{\bigcirc}$ of Figure 6.11 and the bisimilarity partition $\Sigma/_{\simeq^B}$ containing the three singleton regions $\{s\}$, $\{t\}$, and $\{u\}$, the stutter closure $\Sigma/_{\cong^B}$ contains the two regions $\{s, t\}$ and $\{u\}$.

Remark 6.10 [Stutter insensitivity] Observational equivalence is stutter-insensitive, and bisimilarity is not. ■

Proposition 6.8 [Stutter closure] *Bisimilarity $\simeq^B$ is more distinguishing than its stutter closure $\cong^B$.*

Exercise 6.20 {T2} [Stutter-insensitive bisimilarity] Prove Proposition 6.8. ■

Thus, the number of equivalence classes of $\cong^B$ may be much smaller than the number of equivalence classes of $\simeq^B$, and thus, employing $\cong^B$ for reduction may improve the efficiency of verification. Observe that the stutter-closed bisimilarity partition $\cong^B_K$ can be computed by first constructing the stutter closure K^S, and then employing the partition refinement algorithms using the propositional equivalence as the initial partition.

Example 6.11 [Stuttering equivalence of message passing protocols] Recall the modules *SyncMsg* of Figure 2.20 and *AsyncMsg* of Figure 2.24. Note that the two modules have identical observations, namely, the produced message msg_P and the consumed message msg_C. The two modules are not bisimilar. This is because the number of rounds it takes for the produced message to appear as a consumed message are different in the two modules. However, verify the two modules are stutter-closed bisimilar. ■

For the observation structure $K_{\bigcirc}$ of Figure 6.11, we have $s \cong^B t$, while the STL formula $\exists \bigcirc p$ is satisfied only by the state t. Intuitively, the next operator allows us to count the number of rounds, while stutter-closure does not care about the number of update rounds required to change the observation. Thus, stutter-closed bisimilarity is not an abstract semantics for STL. The next proposition asserts that STL$^{\mathcal{U}}$ formulas cannot distinguish among $\cong^B$-equivalent states. This implies that the logic STL$^{\mathcal{U}}$ is stutter-insensitive, and thus, for model checking of STL$^{\mathcal{U}}$-formulas, we may use stutter-closed bisimilarity for reduction.

Proposition 6.9 [STL$^{\mathcal{U}}$ abstraction] *Stutter-closed bisimilarity $\cong^B$ is an abstract semantics for* STL$^{\mathcal{U}}$.

Exercise 6.21 {T3} [STL$^{\mathcal{U}}$] (1) Prove Proposition 6.9. (2) Show that stutter-closed bisimilarity is not fully abstract for STL$^{\mathcal{U}}$. ■

6.5 The Expressive Power of STL

While the distinguishing powers of the logics STL and STL$^{\bigcirc}$ are identical, they have different expressive powers.

Let Φ and Ψ be two state logics. The logic Φ is *as expressive as* the logic Ψ if for every formula ϕ of Ψ, there exists a formula ψ of Φ such that for every observation structure K, the characteristic regions $[\![\phi]\!]_K$ and $[\![\psi]\!]_K$ are identical. The logic Φ is *more expressive than* the logic Ψ if Φ is as expressive as Ψ, but Ψ is not as expressive as Φ. The two logics Φ and Ψ are *equally expressive* if Φ is as expressive as Ψ, and Ψ is as expressive as Φ. The expressive powers of the two logics Φ and Ψ are *incomparable* if Φ is not as expressive as Ψ, and Ψ is not as expressive as Φ.

Exercise 6.22 {T2} [Distinguishing vs. expressive power] Let Φ and Ψ be two state logics. Prove that if Φ is as expressive as Ψ, then Φ is as distinguishing as Ψ. What other relationships between the distinguishing and expressive powers of state logics can you think of? ■

The following proposition establishes that the *until*-connective cannot be expressed by combining the *next*-connectives.

Proposition 6.10 [Expressiveness of Until] *The logic* STL$^{\bigcirc}$ *is not as expressive as the logic* STL$^{\mathcal{U}}$.

Proof. Consider the formula $\exists\Diamond p$ of STL$^{\mathcal{U}}$. To prove that the formula $\exists\Diamond p$ is not expressible in STL$^{\bigcirc}$, we need to show that, for every formula ϕ of STL$^{\bigcirc}$, there exists an observation structure K such that $[\![\phi]\!]_K$ differs from $[\![\exists\Diamond p]\!]_K$.

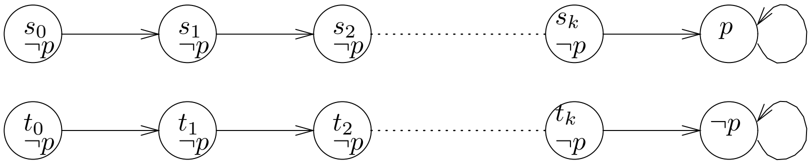


Figure 6.10: Expressive power of the eventually operator

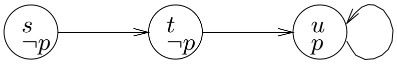


Figure 6.11: Expressive power of the next operator

Consider a formula ϕ of $\text{STL}^{\bigcirc}$, and let k be the number of occurrences of the operator $\exists\bigcirc$ in ϕ. Consider the structure shown in Figure 6.10. We have $s_0 \models \exists\Diamond p$ and $t_0 \not\models \exists\Diamond p$. We claim that for every $0 \leq i \leq k$, if ψ is a formula of $\text{STL}^{\bigcirc}$ with at most i occurrences of the temporal operator $\exists\bigcirc$, then either both s_{k-i} and t_{k-i} satisfy ψ, or both do not satisfy ψ. The proof is by induction on i.

For $i = 0$, we need to consider formulas with no occurrences of $\exists\bigcirc$, that is, propositional formulas. Since the states s_k and t_k have same observations, they agree on the truth of propositional formulas.

Now consider the case $i = j + 1$ for $0 \leq j < k$. We need to prove that the states s_{k-i} and t_{k-i} agree on the truth of formulas with at most i occurrences of $\exists\bigcirc$ assuming that the states s_{k-j} and t_{k-j} agree on the truth of formulas with at most j occurrences of $\exists\bigcirc$. The proof is straightforward using induction on the structure of ψ.

In conclusion, since ϕ has only k occurrences of $\exists\bigcirc$, the states s_0 and t_0 agree on the truth of ϕ, and hence, $[\![\phi]\!] \neq [\![\exists\Diamond p]\!]$. ■

Conversely, the *next*-connective cannot be expressed by combining the *until*-connectives.

Proposition 6.11 [Expressiveness of Next] *The logic* $\text{STL}^{\mathcal{U}}$ *is not as expressive as the logic* $\text{STL}^{\bigcirc}$.

Proof. Consider the formula $\exists\bigcirc p$ of $\text{STL}^{\bigcirc}$, and the observation structure $K_{\bigcirc}$ shown in Figure 6.11. We know that $t \models \exists\bigcirc p$ and $s \not\models \exists\bigcirc p$. We claim that the formula $\exists\bigcirc p$ is not expressible in the fragment $\text{STL}^{\mathcal{U}}$. It suffices to show that, for the observation structure $K_{\bigcirc}$, for every formula ϕ of $\text{STL}^{\mathcal{U}}$, either both s and t satisfy ϕ or both do not satisfy ϕ. The proof is by induction on the structure of ϕ. Since the states s and t have the same observations, they satisfy the same set of predicates. The inductive cases for logical connectives follow immediately. Let us consider the formula $\phi = \phi_1 \exists\mathcal{U} \phi_2$. From inductive hypothesis, the states s and t agree on the truth of the subformulas ϕ_1 and ϕ_2. This implies that both s and t satisfy ϕ precisely under the same conditions, namely, when they satisfy ϕ_2, or when they satisfy ϕ_1 and u satisfies ϕ_2. ■

Exercise 6.23 {T3} [STL without until] Prove that there is no formula ϕ of STL such that (1) ϕ uses only propositions, logical connectives, and the temporal operators $\exists\bigcirc$ and $\exists\Diamond$, and (2) for every observation structure K, $[\![\phi]\!]_K = [\![p\exists\mathcal{U}q]\!]_K$. This implies that the until-operator is not expressible using the next and eventually operators. ■

Exercise 6.24 {T3} [Event STL] Is ESTL more expressive than STL? Prove your answer. ■

Exercise 6.25 {T3} [Strict Until] The logic STL$^+$ has the syntax

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \neg\phi \mid \exists\bigcirc\, \phi \mid \phi\exists\mathcal{U}^+\phi,$$

where the semantics of the *strict-until* operator is defined by the clause

$$\begin{array}{lll} s \models_K \psi\exists\mathcal{U}^+\phi & \text{iff} & \text{there is a source-}s\text{ trajectory } \overline{s}_{0..m} \text{ of } K \text{ such that (1) } m > 0 \\ & & \text{(2) } s_m \models_K \phi \text{ and} \\ & & \text{(3) for all } 0 \le i \le m,\ s_i \models_K \phi \vee \psi. \end{array}$$

Thus, while $p\exists\mathcal{U}q$ can be satisfied in a state s by satisfying q in s, satisfaction of $p\exists\mathcal{U}^+q$ in a state s requires a source-s trajectory that is of at least length 2 and leads to a state satisfying q. Is the logic STL as expressive as the logic STL$^+$? Is the logic STL$^+$ as expressive as the logic STL? ■

Contents

Chapter 7

Automata-theoretic Safety Verification

Automata provide an alternative to temporal logic for specifying requirements of reactive modules. In the automata-theoretic approach to verification, a module is viewed as a generator of a formal language over the set of observations. The requirement is specified by an automaton that accepts only the desired behaviors. The verification problem, then, reduces to a language-inclusion problem: whether every sequence of observations generated by the module is accepted by the requirements automaton.

7.1 Automata

Languages over observations

The execution of a transition graph G results in a trajectory of G, which is a finite sequence of states. The execution of an observation structure K results in a trajectory of the underlying transition graph, and each state of the trajectory has an associated observation. The resulting finite sequence of observations is called a *trace* of K. The set of traces corresponding to initialized trajectories is a language over the set A of observations, and is the language generated by K.

Traces

Let $K = (G, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. A *trace* of K is a nonempty word $\overline{a}$ over the alphabet A of observations such that there is a trajectory $\overline{s}$ of G with $\overline{a} = \langle\!\langle\overline{s}\rangle\!\rangle$. The word $\overline{a}$ is a *source-s trace* of K, for a state s of K, if there is a source-s trajectory $\overline{s}$ of G with $\overline{a} = \langle\!\langle\overline{s}\rangle\!\rangle$. The word $\overline{a}$ is an *initialized trace* of K if there is an initialized trajectory $\overline{s}$ of G with $\overline{a} = \langle\!\langle\overline{s}\rangle\!\rangle$. We write $L_K(s)$ for the set of source-s traces of K, and L_K for the set of initialized traces of K. The set L_K is called the *language* of K. For a module P, the set L_{K_P} is called the language of P.

Example 7.1 [Traces] A possible initialized trace of the observation structure of Figure 6.1 is the word *pqprrrpr*. The language of initialized traces is a regular language specified by the expression $p + (pq((\epsilon + p)r)^*)^+(\epsilon + p)$.

The language of the module *Pete* of Chapter 1 is a regular language over the alphabet $\{outC, reqC, inC\} \times \{outC, reqC, inC\}$. A possible initialized trace of the module *Pete* is the word

$$\begin{array}{llll}(outC, outC), & (outC, reqC), & (reqC, inC), & (reqC, inC),\\ (reqC, outC), & (reqC, reqC), & (inC, reqC), & (outC, reqC),\\ (outC, inC), & (reqC, outC), & (inC, outC), & (outC, outC)\end{array}$$

corresponding to the initialized trajectory of Figure 2.1. ■

Remark 7.1 [Closure properties of trace languages] For every observation structure K, the language L_K is prefix-closed, but not necessarily fusion-closed. For example, for the observation structure shown in Figure 6.1, both *pqrpq* and *pqp* are initialized traces, but *pqpq* is not an initialized trace. This simply says that an observation may be caused by many different states and, by itself, does not necessarily determine the future behavior of the structure. ■

Exercise 7.1 {`T2`} [Prefix-closure] If K is an observation structure with the observation alphabet A, then the set L_K of initialized traces is a prefix-closed language over the alphabet A. Conversely, let A be an alphabet, and let L be a prefix-closed language over A. Show that there is an observation structure K such that $L = L_K$. ■

From observation structures to automata

Since languages of observation structures are prefix-closed, observation structures are not closed under complementation. To define languages that are not prefix-closed, we add acceptance conditions to observation structures.

Automaton

An *automaton* M consists of (1) an observation structure K and (2) [the *accepting region*] a region σ^A of K. An initialized trajectory $\overline{s}_{0..m}$ of K is *accepted* by the automaton M if $s_m \in \sigma^A$. An initialized trace $\overline{a}$ of K is *accepted* by M if $\overline{a} = \langle\langle \overline{s} \rangle\rangle$ for some initialized trajectory $\overline{s}$ of K that is accepted by M. The *language* L_M of the automaton M is the set of initialized accepted traces of M.

The language of an automaton is a subset of the language of the underlying observation structure. Let K be an observation structure with the state space Σ. Declaring every state of K as accepting, we obtain the automaton $M_K = (K, \Sigma)$. For every observation structure K, $L_{M_K} = L_K$. Thus, every observation structure can be considered as an automaton, and sometimes we will not make the distinction between the observation structure K and the automaton M_K.

Remark 7.2 [Automaton definition] Our definition of an automaton is similar to the common definitions found in the textbooks on formal languages. In particular, if the automaton M has finitely many states, then the language L_M is regular. In the more common definition of an automaton, the transitions of the automaton are labeled with alphabet symbols. This is dual to our definition in which the states are labeled with observations. As a consequence of our definition, the empty word ϵ does not belong to the language of any automaton. For every regular language L over a finite alphabet A, there exists a finite automaton M such that $L_M = L \backslash \{\epsilon\}$. ■

The language-inclusion problem

The language-inclusion problem asks whether every initialized accepted trace of one automaton is also an initialized accepted trace of another automaton.

The language-inclusion problem

An instance (M_1, M_2) of the *language-inclusion problem* consists of two automata M_1 and M_2 over the same observation alphabet A. The answer to the language-inclusion problem (M_1, M_2) is Yes if $L_{M_1} \subseteq L_{M_2}$, and otherwise No.

7.2 Safe Automaton Logic

7.2.1 Syntax and Semantics

Automata can be used for specifying requirements of reactive modules. The observations of the requirements automaton are boolean expressions over the observable variables of modules. We define the state logic Sal whose formulas are boolean combinations of such automata.

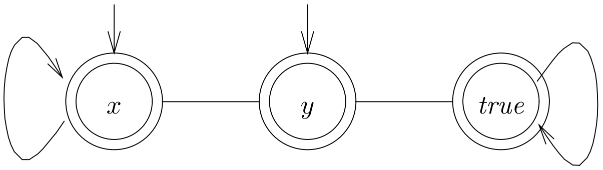


Figure 7.1: The automaton $M_{\mathcal{W}}$

SAFE AUTOMATON LOGIC

The formulas of the state logic *Safe automaton logic* (SAL) are generated by the grammar

$$\phi \ ::= \ \forall M \mid \neg\phi \mid \phi \vee \phi$$

where M is an automaton whose observations are boolean expressions.

Given a formula ϕ of SAL, an observation structure K is a ϕ-structure if each observation of K is a valuation for a superset of the variables appearing in the observations of all automata occurring in ϕ.

The satisfaction relation for SAL is defined by:

$$\begin{array}{lll} s \models_K \forall M & \text{iff} & \text{for every source-}s\text{ trajectory } \overline{s}_{0..m} \text{ of } K \\ & & \text{there is a trace } \overline{a}_{0..m} \in L_M \text{ such that} \\ & & \text{for all } 0 \leq i \leq m,\ s_i \models a_i \\ s \models_K \phi \vee \psi & \text{iff} & s \models_K \phi \text{ or } s \models_K \psi; \\ s \models_K \neg\phi & \text{iff} & s \not\models_K \phi, \end{array}$$

where M is an automaton, ϕ and ψ are SAL formulas, and K is a (M, ϕ, ψ)-structure.

In other words, a state s of K satisfies the requirement specified by the formula $\forall M$ if for every source-s trace $\overline{a}$ of K, we can find an initialized accepting trace $\overline{b}$ of M such that every observation in $\overline{a}$ is consistent with the corresponding expression in $\overline{b}$. The characteristic region of an SAL formula, the satisfaction of an SAL formula by an observation structure, the model-checking problem for SAL, and the verification problem for SAL are defined as in case of other state logics such as STL.

Example 7.2 [Specifying $x\forall\mathcal{W}y$ in SAL] The SAL formula $\forall M_{\mathcal{W}}$ for the automaton shown in Figure 7.1 asserts that, given a state s, along every source-s trajectory a state that violates x coincides with or is preceded by a state that satisfies y. The formula $\forall M_{\mathcal{W}}$ can be interpreted at states of an observation structure whose observations assign values to x and y. It follows that the SAL formula $\forall M_{\mathcal{W}}$ is equivalent to the STL formula $x\forall\mathcal{W}y$. ■

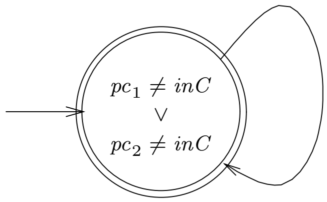
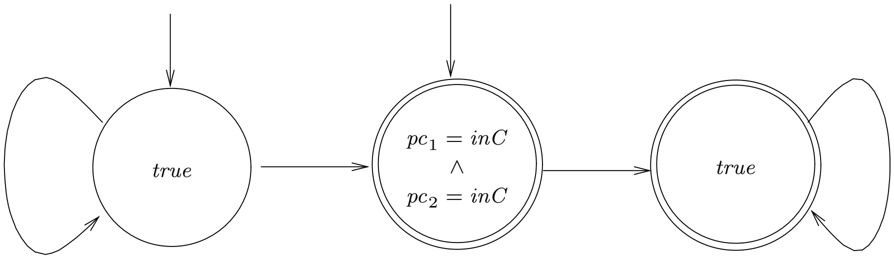


Figure 7.2: The mutual-exclusion requirement in Sal



Automaton $M_{\neg mutex}$

Figure 7.3: The violation of mutual-exclusion requirement in Sal

Example 7.3 [Mutual exclusion] Recall mutual-exclusion protocols from Chapter 1. The mutual-exclusion requirement can be expressed in Sal by the automaton M_{mutex} shown in Figure 7.2. That is, the Sal formula $\forall M_{mutex}$ is equivalent to the Stl formula $\forall\Box(pc_1 \neq inC \vee pc_2 \neq inC)$. Notice that allowing the predicate $(pc_1 \neq inC \vee pc_2 \neq inC)$ as an observation in the automaton M_{mutex} permits a compact description of the property. ■

Remark 7.3 [Final states] Notice that both automata $M_{\mathcal{W}}$ and M_{mutex} are really observation structures, because all their states are accepting. This is no accident. Given an automaton M, let M^+ be the automaton that accepts the maximal prefix-closed subset of L_M; that is, $\overline{a} \in L_{M^+}$ if all prefixes of $\overline{a}$ are in L_M (how would you construct M^+?). Clearly, L_{M^+} can be accepted by an automaton all of whose states are accepting (remove the states that cannot reach an accepting state and make all remaining states accepting). If we specify a requirement of a state s of structure K by the Sal formula $\forall M$, then $s \models_K M$ iff $s \models_K M^+$. So we may specify the same requirement as $\forall M^+$.

We use automata (with accepting states) for specifying existential properties. Given an automaton M, let $-M$ be the complementary automaton (complementation requires accepting states; see below). In state s, the Sal formula $\neg\forall - M$ specifies that *some* trace from s is a trace of M. Therefore, we define

$\exists M$ as $\neg\forall - M$. For example, mutual exclusion can be specified by the SAL formula $\neg\exists M_{\neg mutex}$, where the automaton $M_{\neg mutex}$ of Figure 7.3 accepts all traces that cause violation of the mutual exclusion requirement. ■

Exercise 7.2 {`T2`} [Mutual exclusion] Express in SAL the first-request-first-in requirement and the equal-opportunity requirement for mutual-exclusion protocols (see Chapter 6 for the requirements). ■

Exercise 7.3 {`T2`} [Specifying $\forall\Box(x\forall\mathcal{W}y)$ in SAL] The automaton of Figure 7.1 expresses a requirement that is equivalent to the STL formula $x\,\forall\mathcal{W}\,y$. Write an automaton M such that $\forall M$ is equivalent to the STL formula $\forall\Box(x\,\forall\mathcal{W}\,y)$. ■

SAL **model checking**

The model-checking problem for SAL can be reduced to the language-inclusion problem. First, since every SAL formula is a boolean combination of automata, it is sufficient to consider the problem of checking whether an observation structure satisfies an automaton specification. For this purpose, we expand each automaton M of SAL to a larger automaton EM whose observations are valuations to the variables appearing in the observations of M.

EXPANSION OF A SAL AUTOMATON

For a SAL automaton M, its *expansion* EM is another automaton with the following components. (1) [Observation alphabet] Observations of EM are the valuations for the variables appearing in the observations of M. (2) [State space] For every state s of M, EM has a state $\langle s,t\rangle$ for each observation t of EM such that t satisfies the observation of s. (3) [Initial region] The state $\langle s,t\rangle$ is initial in EM if the state s is initial in M. (4) [Transition relation] There is a transition from $\langle s,t\rangle$ to $\langle s',t'\rangle$ in EM if there is a transition from s to s' in M. (5) [Observation function] The observation of $\langle s,t\rangle$ is t. (6) [Accepting region] The state $\langle s,t\rangle$ is accepting in EM if the state s is accepting in M.

Example 7.4 [SAL model checking] Figure 7.4 shows the observation structure $EM_{\mathcal{W}}$ for the SAL automaton $M_{\mathcal{W}}$ from Figure 7.1 (all states are accepting). ■

Thus, the observations of the automaton EM completely specify the values of the variables mentioned in the expressions labeling the original automaton M. Consider a M-structure K, and a trajectory $\overline{s}_{0...m}$ of K. There exists a (initialized accepting) trace $\overline{a}_{0...m}$ of M such that $\langle\!\langle s_i\rangle\!\rangle \models a_i$ for $0 \leq i \leq m$, iff $\langle\!\langle \overline{s}\rangle\!\rangle$ is an (initialized accepting) trace of EM. It follows that checking whether an observation structure satisfies an SAL automaton M is equivalent to checking whether K satsifies the expanded automaton EM, which in turn corresponds to checking whether the language of K is contained in the language of EM.

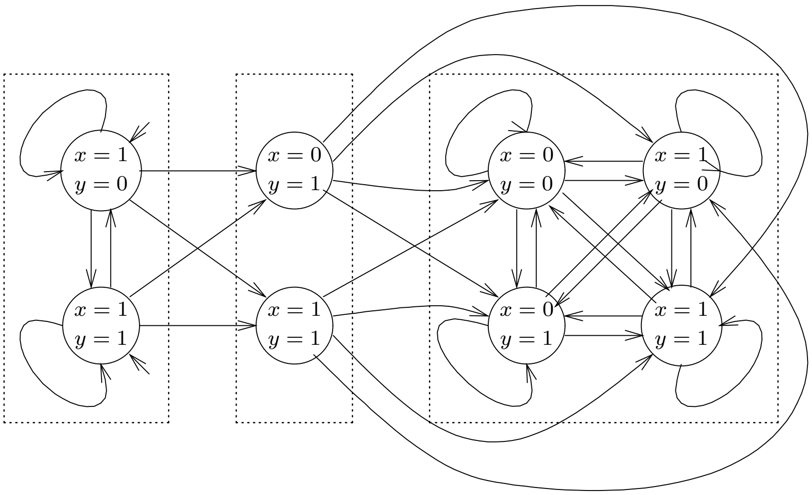


Figure 7.4: The automaton $EM_{\mathcal{W}}$

Proposition 7.1 [SAL model checking] *The* SAL *model-checking problem* (K, M) *and the language-inclusion problem* (K, EM) *have the same answer.*

Exercise 7.4 {T3} [SAL verification vs. invariant-verification with monitors] Can a given SAL-verification problem always be reduced to an invariant-verification problem after introducing a monitor (see Chapter 2 for monitors)? What about the converse? ■

7.2.2 The Distinguishing Power of SAL

The set of traces associated with a state of an observation structure leads to a natural way of equating two states: two states s and t of an observation structure are *trace equivalent* iff every source-s trace is also a source-t trace, and vice versa.

TRACE EQUIVALENCE

Two states s and t of an observation structure K are *trace equivalent*, denoted $s \simeq^L_K t$, if $L_K(s) = L_K(t)$. The induced state equivalence $\simeq^L$ is also called *trace equivalence.*

Example 7.5 [Trace equivalence versus bisimilarity] Consider the observation structure shown in Figure 7.5. The two states s and t are bisimilar, but not trace equivalent. This is because, trace equivalence, unlike bisimilarity, disregards the branching within an observation structure. ■

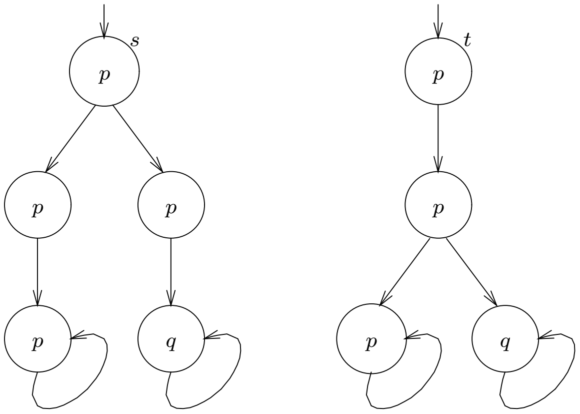


Figure 7.5: Bisimilarity vs. trace equivalence

The next proposition establishes that trace equivalence is less distinguishing than bisimilarity. This means that two bisimilar states are guaranteed to be trace equivalent, but not vice versa. Since bisimilarity is sensitive to the branching nature of the structure, while trace equivalence depends only on the set of trace generated, bisimilarity is called a *branching-time* equivalence, and trace equivalence is called a *linear-time* equivalence.

Proposition 7.2 [Distinguishing power of trace equivalence] *Trace equivalence is less distinguishing than bisimilarity.*

Proof. Let K be an observation structure, and let s and t be two states of K. If s and t are bisimilar, then for every source-s trajectory $\overline{s}_{0\ldots m}$, there exists a source-t trajectory $\overline{t}_{0\ldots m}$ such that $s_i \simeq^B t_i$ for all $0 \leq i \leq m$. Since bisimilar states have identical observations, it follows that every source-s trace is also a source-t trace. Hence, $s \simeq^L t$. This establishes that bisimilarity is as distinguishing as trace equivalence. Example 7.5 establishes that trace equivalence is not as distinguishing as bisimilarity. ■

Like Stl, for model checking of Sal formulas it is fine to consider a quotient structure obtained by collapsing states that satisfy the same set of Sal formulas.

Proposition 7.3 [Sal abstraction] Sal *admits abstraction.*

Recall that bisimilarity is a fully abstract semantics for Stl: two bisimilar states satisfy the same set of Stl formulas, and two non-bisimilar states disagree on the satisfaction of some Stl formula. The fully abstract semantics for Sal is trace equivalence: no Sal formula can distinguish between two states that are trace equivalent, and for every two states that are not trace equivalent, there exists a Sal formula that is satisfied by only one of the two states.

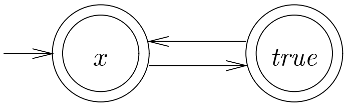


Figure 7.6: The automaton M_{even}

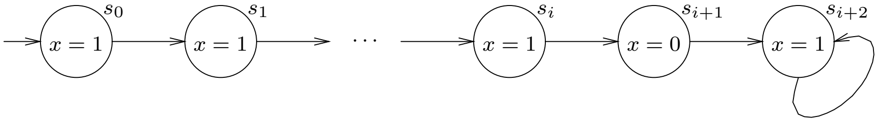


Figure 7.7: STL cannot express $\forall M_{even}$

Theorem 7.1 [Distinguishing power of SAL] *Trace equivalence is a fully abstract semantics for* SAL*.*

It follows that SAL is less distinguishing than STL.

Corollary 7.1 [Distinguishing power of SAL vs. STL] SAL *is less distinguishing than* STL*.*

It follows that to solve an instance (K, ϕ) of the model checking problem for SAL, we can construct the minimal quotient $K/_{\simeq^B}$ using one of the algorithms of Chapter 4, and solve $(K/_{\simeq^B}, \phi)$.

7.2.3 The Expressive Power of SAL

If a state logic Φ is more distinguishing than a state logic Ψ, then the logic Ψ cannot be as expressive as Φ.

Example 7.6 [SAL is not as expressive as STL] Consider Figure 7.5. The states s and t are trace equivalent, and hence, satisfy the same set of SAL formulas. On the other hand, for the STL formula $\phi = \forall\bigcirc\,\exists\bigcirc\, p$, $s \not\models \phi$ and $t \models \phi$. It follows that no formula of SAL is equivalent to the STL formula ϕ. ■

Even though STL is more distinguishing than SAL, there are requirements that are expressible in SAL, but not in STL.

Theorem 7.2 [Expressive power of SAL vs. STL] *The expressive powers of* SAL *and* STL *are incomparable.*

Proof. We need to establish that STL is not as expressive as SAL. Consider the SAL formula $\phi_{even} = \forall M_{even}$ shown in Figure 7.6, where x is a boolean variable. The SAL formula ϕ_{even} asserts that, given a state s, along every source-s trajectory x is satisfied in every other state. That is, a state s satisfies ϕ_{even} if for every source-s trajectory $\overline{s}_{0 \ldots m}$, for all $0 \leq i \leq m$, if the index i is even, $x[s_i] = 1$.

We wish to establish that no STL formula is equivalent to ϕ_{even}. Assume, to the contrary, that there exists an STL formula ψ such that for every observation structure K, $[\![\psi]\!]_K = [\![\phi_{even}]\!]_K$. Suppose ψ has i subformulas. Consider the observation structure of Figure 7.7. Observe that only one of the two states s_0 and s_1 satisfies ϕ_{even}. We will establish that either both of them satisfy ψ or none of them satisfies ψ.

We prove that for every subformula χ of ψ, if χ has n subformulas, then the states $s_0, \ldots s_{i-n+1}$ agree on the truth of χ. The proof is by induction on n.

Base case $n = 1$: χ is an atomic formula. The states $s_0, \ldots s_i$ have identical observations, and hence, agree on the truth of χ.

Inductive case $n > 1$: We consider the case $\chi = \chi_1 \exists \mathcal{U} \chi_2$, and leave the simpler cases for the reader to verify. Consider two states s_j and s_k with $0 \leq j, k \leq (i - n + 1)$. It suffices to prove that if $s_j \models \chi$ then $s_k \models \chi$. Assume that $s_j \models \chi$. We consider the case when $j > k$, and the case $j < k$ is left to the reader.

Suppose $s_j \models \chi_2$. Then, by induction hypothesis, $s_k \models \chi_2$, and hence, $s_k \models \chi$.

Suppose $s_j \not\models \chi_2$. Then there exists $j' > j$ such that $s_{j'} \models \chi_2$ and $s_{j''} \models \chi_1$ for all $j \leq j'' < j'$. Since $s_j \models \chi_1$, by induction hypothesis, $s_{j''} \models \chi_1$ for all $k \leq j'' \leq j$. Hence, $s_k \models \chi$. ■

Exercise 7.5 {T2} [Attempting to specify even requirement in STL] Why is the STL formula

$$x \wedge \forall\Box(x \rightarrow \forall\bigcirc \neg x) \wedge \forall\Box(\neg x \rightarrow \forall\bigcirc x)$$

not equivalent to $\forall M_{even}$? ■

Exercise 7.6 {T3} [Expressive power of SAL vs. STL$^{\mathcal{U}}$] Prove that the expressive powers of SAL and STL$^{\mathcal{U}}$ are incomparable. ■

Exercise 7.7 {T4} [Stutter sensitivity] (1) Show that SAL is stutter-sensitive. (2) Define a stutter-insensitive version of SAL. What is the equivalence induced by stutter-insensitive SAL? How do the distinguishing power and the expressive power of stutter-insensitive SAL compare with STL$^{\mathcal{U}}$? ■

7.3 Operations on Automata

The determinization and product construction for automata are building blocks for solving the language-inclusion problem.

7.3.1 Determinization

In deterministic observation structures, there is at most one initial state per observation, and each state has at most one successor per observation.

Deterministic observation structure

Let $K = (G, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. The observation structure K is *deterministic* if (1) [deterministic initialization] for each observation a of K, there is at most one initial state s with $\langle\!\langle s\rangle\!\rangle = a$, and (2) [deterministic update] for each state s and each observation a of K, there is at most one successor t of s with $\langle\!\langle t\rangle\!\rangle = a$.

An automaton is deterministic if its observation structure is deterministic. The module P is deterministic if the observation structure K_P is deterministic. The module P is deterministic if (1) it is closed, and (2) the initial commands of all its atoms are deterministic, and (3) the update commands of all its atoms are deterministic.

Example 7.7 [Deterministic structures] The observation structure shown in Figure 6.1 is not deterministic. The Sal formulas of Figure 7.1 and Figure 7.2 are deterministic, while the observation structure of Figure 7.4 is not. ■

Remark 7.4 [Traces define trajectories in deterministic structures] If K is a deterministic observation structure, and $\overline{t}$ and $\overline{u}$ are different trajectories of K with the same source, then $\langle\!\langle\overline{t}\rangle\!\rangle \neq \langle\!\langle\overline{u}\rangle\!\rangle$. It follows that for each state s of a deterministic observation structure K, there is a one-to-one correspondence between the source-s trajectories and the source-s traces. ■

The region σ of the observation structure K is *consistent* if for all states in σ have the same observation: for all s and t in σ, $s \approx_K t$. In other words, a consistent region is a subset of an $\approx_K$-equivalence class of the propositional equivalence. Every observation structure, and every automaton, can be determinized by replacing the states with the consistent regions. The determinization procedure is usually referred to as the *subset construction*. A state of ΔK is a set of states of K. Intuitively, the transition relation of ΔK is defined so that the sink-state of ΔK corresponding to a trace $\overline{a}$ contains all the sink-states of initialized trajectories of K corresponding to the trace $\overline{a}$. Since our observation structures have finite nonobservable nondeterminism, during determinization, it suffices to consider only finite consistent regions.

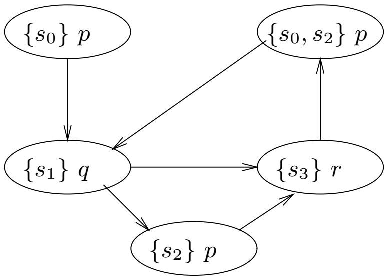


Figure 7.8: Determinized structure

DETERMINIZATION

Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. The *determinized structure* ΔK is the observation structure $(\Sigma_\Delta, \sigma^I_\Delta, \rightarrow_\Delta, A, \langle\!\langle\cdot\rangle\!\rangle_\Delta)$, where (1) [state space] Σ_Δ is the set of nonempty finite consistent regions of M; (2) [initial region] $\sigma \in \sigma^I_\Delta$ iff there is an observation $a \in A$ such that $\sigma = \{s \in \sigma^I \mid \langle\!\langle s\rangle\!\rangle = a\}$; (3) [transition relation] $\sigma \rightarrow_\Delta \tau$ iff there is an observation $a \in A$ such that $\tau = \{s \in \mathit{post}(\sigma) \mid \langle\!\langle s\rangle\!\rangle = a\}$; (4) [observation function] $s \in \sigma$ implies $\langle\!\langle \sigma\rangle\!\rangle_\Delta = \langle\!\langle s\rangle\!\rangle$.

For an automaton $M = (K, \sigma^A)$, the *determinized automaton* ΔM is the automaton $(\Delta K, \sigma^A_\Delta)$, where $\sigma \in \sigma^A_\Delta$ iff $\sigma \cap \sigma^A$ is nonempty.

Example 7.8 [Determinization] Figure 7.8 shows the result of determinizing the observation structure from Figure 6.1. ■

Proposition 7.4 [Determinization] *For every observation structure K, the observation structure ΔK is deterministic and $L_{\Delta K} = L_K$; for every automaton M, the automaton ΔM is deterministic and $L_{\Delta M} = L_M$.*

Exercise 7.8 {T2} [Properties of determinization] Which of the following properties of an observation structure K are inherited by the determinized structure ΔK: serial; finite; finitely-branching; stutter-closed? ■

Observe that the determinized observation structure has exponentially many more states than the original structure, and thus, determinization is a computationally expensive procedure.

Exercise 7.9 {T2} [Cost of determinization] Let K be a finite observation structure with n states and m transitions. Give a tight bound on the number of states and transitions of the determinized structure ΔK. ■

Exercise 7.10 {T3} [Nondeterminism and exponential succinctness] Consider the observation alphabet $A = \{0, 1\}$. For every natural number m, define the language

$$L_m = \{\overline{a}_{0\ldots 2m} \mid \ a_i = a_{i+m} \text{ for some } 0 \leq i < m\}$$

(1) Show that for every natural number m, there is a nondeterministic automaton M with $3m$ states such that $L_M = L_m$. (2) Show that if M is a deterministic automaton with $L_M = L_m$ then M has at least $\mathbf{2}^m$ states. ■

7.3.2 Boolean Operations

Disjoint union

Two observation structures are *disjoint* if their state spaces are disjoint.

Disjoint union

Let $K_1 = (\Sigma_1, \sigma_1^I \rightarrow_1, A_1, \langle\langle\cdot\rangle\rangle_1)$ and $K_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2, A_2, \langle\langle\cdot\rangle\rangle_2)$ be two disjoint observation structures. The *union* $K_1 + K_2$ is the observation structure $(\Sigma_1 \cup \Sigma_2, \sigma_1^I \cup \sigma_2^I, \rightarrow_1 \cup \rightarrow_2, A_1 \cup A_2, \langle\langle\cdot\rangle\rangle)$, where $\langle\langle s\rangle\rangle = \langle\langle s\rangle\rangle_1$ if $s \in \Sigma_1$, and otherwise $\langle\langle s\rangle\rangle = \langle\langle s\rangle\rangle_2$.

Let $M_1 = (K_1, \sigma_1^A)$ and $M_2 = (K_2, \sigma_2^A)$ be two disjoint automata. The *union* $M_1 + M_2$ is the automaton $(K_1 + K_2, \sigma_1^A \cup \sigma_2^A)$.

Proposition 7.5 [Disjoint union] *If K_1 and K_2 be two disjoint observation structures then $L_{K_1+K_2} = L_{K_1} \cup L_{K_2}$; if M_1 and M_2 are two disjoint automata, then $L_{M_1+M_2} = L_{M_1} \cup L_{M_2}$.*

Remark 7.5 [Cost of union] Let K_1 be a finite structure with n_1 states and m_1 transitions. Let K_2 be a finite structure with n_2 states and m_2 transitions. Assume that K_1 and K_2 are disjoint. Then, the union $M_1 + M_2$ has $n_1 + n_2$ states and $m_1 + m_2$ transitions. Thus, the cost of union is only additive. ■

Product

To obtain intersection of the languages of two observation structures or two automata, we take the cartesian product of the underlying state-spaces.

Product

Let $K_1 = (\Sigma_1, \sigma_1^I, \rightarrow_1, A, \langle\!\langle\cdot\rangle\!\rangle_1)$ and $K_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2, A, \langle\!\langle\cdot\rangle\!\rangle_2)$ be two observation structures. The *product* $K_1 \times K_2$ is the observation structure $(\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$:

- $\Sigma = \{(s_1, s_2) \mid s_1 \in \Sigma_1 \text{ and } s_2 \in \Sigma_2 \text{ and } \langle\!\langle s_1\rangle\!\rangle_1 = \langle\!\langle s_2\rangle\!\rangle_2\}$;
- $(s_1, s_2) \in \sigma^I$ iff $s_1 \in \sigma_1^I$ and $s_2 \in \sigma_2^I$;
- $(s_1, s_2) \rightarrow (t_1, t_2)$ iff $s_1 \rightarrow_1 t_1$ and $s_2 \rightarrow_2 t_2$;
- $\langle\!\langle (s_1, s_2)\rangle\!\rangle = \langle\!\langle s_1\rangle\!\rangle_1 = \langle\!\langle s_2\rangle\!\rangle_2$.

Let $M_1 = (K_1, \sigma_1^A)$ and $M_2 = (K_2, \sigma_2^A)$ be two automata. The *product* $M_1 \times M_2$ is the automaton $(K_1 \times K_2, \sigma^A)$, where $(s_1, s_2) \in \sigma^A$ iff $s_1 \in \sigma_1^A$ and $s_2 \in \sigma_2^A$.

Proposition 7.6 [Product] *If K_1 and K_2 are two observation structures, then $L_{K_1 \times K_2} = L_{K_1} \cap L_{K_2}$. If M_1 and M_2 are two automata, then $L_{M_1 \times M_2} = L_{M_1} \cap L_{M_2}$.*

Remark 7.6 [Cost of product] Let K_1 be a finite structure with n_1 states and m_1 transitions. Let K_2 be a finite structure with n_2 states and m_2 transitions. Then, the product $K_1 \times K_2$ has at most $n_1 \cdot n_2$ states and at most $m_1 \cdot m_2$ transitions. Thus, the cost of product is multiplicative. ■

Exercise 7.1 shows that the class of languages of observation structures is precisely the class of prefix-closed languages. Since the languages of observation structures are closed under union and intersection, the union and the intersection of two prefix-closed languages are prefix-closed.

Exercise 7.11 {T2} [Operations on automata] Which of the following properties of two observation structures K_1 and K_2 are inherited by the disjoint union $K_1 + K_2$ and the product $K_1 \times K_2$: reflexive; finite; finitely-branching; stutter-closed; deterministic? Which laws govern the interplay of the operations stutter closure $\cdot^S$, determinization Δ, union $+$, and product $\times$ on observation structures? ■

Symbolic representation

Let P be a module. Then, the corresponding observation structure K_P is symbolically represeted by (1) the set X_P of variables, (2) the set $\mathsf{obs}X_P$ of observable variables, (3) the initial predicate q_P^I, and (4) the transition predicate q_P^T. If P and Q are two modules, then the union $K_P + K_Q$ has the initial predicate $q_P^I \vee q_Q^I$ and the transition predicate $q_P^T \vee q_Q^T$, and the product $K_P \times K_Q$ has the initial predicate $q_P^I \wedge q_Q^I$ and the transition predicate $q_P^T \wedge q_Q^T$. Thus, the union and product operations are easily implemented on the symbolic representations.

Exercise 7.12 {`P2`} [Symbolic determinization] Write an algorithm that constructs, given the symbolic representation of an observation structure K, the symbolic representation of the determinized structure ΔK. ■

7.3.3 Complementation

Let M_1 and M_2 be two automata with the same set A of observations. The automaton M_2 is a *complement* of M_1 if $L_{M_2} = A^+ \backslash L_{M_1}$. Complementing a deterministic automaton involves two steps: completion and inversion.

In a deterministic automaton, there is at most one initial state per observation, and each state has at most one successor per observation. In a complete automaton, on the other hand, there is at least one initial state per observation, and each state has at least one successor per observation.

Complete automaton

Let $M = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle, \sigma^A)$ be an automaton. The automaton M is *complete* if (1) $\langle\!\langle \sigma^I \rangle\!\rangle = A$ and (2) for each state s, $\langle\!\langle post_M(s) \rangle\!\rangle = A$.

An incomplete automaton can be completed, without changing its language, by adding a dummy nonaccepting state for each observation.

Completion

Let $M = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle, \sigma^A)$ be an automaton. The *completion* ΓM is the automaton over the alphabet A with the following components

- every state of M is a state of ΓM, and in addition, for every observation $a \in A$, ΓM has the state s_a;
- every initial state of M is an initial state of ΓM, and in addition, for every observation $a \in A$ such that $a \notin \langle\!\langle \sigma^I \rangle\!\rangle$, the state s_a is an initial state of ΓM.
- every transition of M is a transition of ΓM, for all observations $a, b \in A$, ΓM has a transition from the state s_a to the state s_b, and for every state s of M and an observation $a \in A$ such that $a \notin \langle\!\langle post_M(s) \rangle\!\rangle$, ΓM has a transition from the state s to the state s_a;
- for $s \in \Sigma$, the observation of s in ΓM is $\langle\!\langle s \rangle\!\rangle$, and for an observation $a \in A$, the observation of s_a in ΓM is a.
- the accepting region of ΓM equals σ^A.

Proposition 7.7 [Automaton completion] *For every automaton M, the completion ΓM is complete and $L_{\Gamma M} = L_M$.*

If an automaton M is deterministic, then so is the automaton ΓM. The second step of complementation corresponds to inverting the accepting region.

INVERSION

Let $M = (K, \sigma^A)$ be an automaton with the state space Σ. The *inversion* $-M$ is the automaton $(K, \Sigma \backslash \sigma^A)$.

Inversion corresponds to complementation for complete deterministic automata.

Proposition 7.8 [Automaton inversion] *Let M be a deterministic and complete automaton. Then the inversion $-M$ is a complement of M.*

Consequently, determinization, followed by completion, and then by inversion results in complementation.

Corollary 7.2 [Automaton complementation] *For an automaton M, the automaton $-\Gamma\Delta M$ is a complement of M.*

Exercise 7.13 {`P1`} [Automaton complementation] Apply the completion and inversion operations to the determinized structure of Figure 7.8, and verify that the language of the resulting automaton is the complement of the language of the observation structure of Figure 6.1. ■

Exercise 7.14 {`T2`} [Properties of complementation] Which of the following properties of a deterministic automaton M are inherited by the complement $-\Gamma M$: serial; finite; finitely-branching; stutter-closed? ■

As mentioned earlier, the class of languages defined by observation structures is not closed under complementation, and thus, complementing an observation structure results in an automaton.

7.4 Model Checking

Now we are ready to address the language-inclusion problem for automata, and consequently, the model checking question for SAL. To check whether the language one automaton M_1 is contained in that of another automaton M_2, it suffices to check whether the language of M_1 has an empty intersection with the complement of the language of M_2.

Proposition 7.9 [Automaton language inclusion] *Let M_1 and M_2 be two automata. Then $L_{M_1} \subseteq L_{M_2}$ iff the language $L_{M_1 \times -\Gamma\Delta M_2}$ is empty.*

Since we already know how to construct the automaton $M_1 \times -\Gamma\Delta M_2$ from the automata M_1 and M_2, we have reduced the language-inclusion problem to that of checking language emptiness. Checking emptiness of the language of an automaton is a reachability problem.

Proposition 7.10 [Automaton language emptiness] *Let M be an automaton with the transition graph G and the accepting region σ^A. Then L_M is empty iff the answer to the reachability problem (G, σ^A) is* No.

This immediately suggests a decision procedure for the language-inclusion problem. Algorithm 7.1 solves the language-inclusion problem using a depth-first-search on the state space of the product $M_1 \times \Delta M_2$. The completion and inversion steps are applied only implicitly. The abstract data type for automata supports, in addition to the operations *InitQueue* and *PostQueue*, the operation *Accept*, that, given an automaton M returns the accepting region of M, and the operation *Obs*, that, given an automaton M and a state s of M returns the observation of s.

Theorem 7.3 [Language inclusion] *Let M_1 be a finite automaton with n_1 reachable states and m_1 reachable transitions, and let M_2 be a finite automaton with n_2 states. Algorithm 7.1 solves the language-inclusion problem (M_1, M_2) in time $O((n_1 + m_1) \cdot \mathbf{2}^{n_2})$.*

Remark 7.7 [Space complexity of language inclusion] Solving the language-inclusion problem (M_1, M_2) requires searching the state-space of $M_1 \times -\Gamma\Delta M_2$, which can be performed in space logarithmic in the number of states of $M_1 \times -\Gamma\Delta M_2$. Consequently, the language-inclusion problem (M_1, M_2) can be solved in space $O(n_2 \cdot \log\ n_1)$. Thus the space complexity of the language-inclusion problem is Pspace. The problem is Pspace-hard in the number of states of K_2. The language-inclusion problem (K_1, K_2) for observation structures is Pspace-complete in the number of states of K_2. ■

Exercise 7.15 {P2} [Witness reporting in language inclusion] If the answer to the language inclusion problem (M_1, M_2) is No, then a witness trace is a trace in $L_{M_1} \backslash L_{M_2}$. Modify Algorithm 7.1 so that when it answers No, it also returns a witness trace. ■

Sal **model checking**

Consider the model checking problem $(K, \forall M)$, where M is a specification automaton. As discussed earlier, the model checking problem $(K, \forall M)$ can be solved by solving the language-inclusion problem (K, EM). Suppose M has n states and the observations of M are boolean expressions over a set X of propositional variables. According to the definition of EM, a state of EM consists of

Algorithm 7.1 [Language inclusion]

```
Input: two automata $M_1$ and $M_2$.
Output: the answer to the language-inclusion problem $(M_1, M_2)$.
$s, t$: state;  $\sigma, \tau$: region;
$\sigma^R$: set of state $\times$ region;
$F$: stack of state $\times$ region;
$F := \mathit{EmptyStack}$;
$\sigma^R := \mathit{EmptySet}$;
foreach $s$ in $\mathit{InitQueue}(M_1)$ do
  $\sigma := \mathit{EmptySet}$;
  foreach $t$ in $\mathit{InitQueue}(M_2)$ do
    if $\mathit{Obs}(s, M_1) = \mathit{Obs}(t, M_2)$ then $\sigma := \mathit{Insert}(t, \sigma)$ fi
    od;
  if $\mathit{IsMember}(s, \mathit{Accept}(M_1)$ and $\mathit{IsEmpty}(\sigma \cap \mathit{Accept}(M_2))$
    then return No fi;
  if not $\mathit{IsMember}((s, \sigma), \sigma^R)$ then
    $F := \mathit{Push}((s, \sigma), F)$;
    $\sigma^R := \mathit{Insert}((s, \sigma), \sigma^R)$
    fi
  od;
while not $\mathit{IsEmpty}(F)$ do
  $(s, \sigma) := \mathit{Top}(F)$;
  $F := \mathit{Pop}(F)$;
  foreach $t$ in $\mathit{PostQueue}(s, M_1)$ do
    $\tau := \mathit{EmptySet}$;
    foreach $u$ in $\mathit{PostQueue}(\sigma, M_2)$ do
      if $\mathit{Obs}(t, M_1) = \mathit{Obs}(u, M_2)$ then $\tau := \mathit{Insert}(t, \tau)$ fi
      od;
    if $\mathit{IsMember}(t, \mathit{Accept}(M_1))$ and $\mathit{IsEmpty}(\tau \cap \mathit{Accept}(M_2))$
      then return No fi;
    if not $\mathit{IsMember}((t, \tau), \sigma^R)$ then
      $\sigma^R := \mathit{Insert}((t, \tau), \sigma^R)$;
      $F := \mathit{Push}((t, \tau), F)$
      fi
    od
  od;
return Yes.
```

Figure 7.9: Checking language inclusion

a state of M and a valuation for the variables in X. Thus, the automaton EM has $n \cdot \mathbf{2}^k$ states if X has k propositional variables. However, there is no need to construct EM explicitly. As usual, we will use on-the-fly representation, that is, implement the functions *InitQueue* and *PostQueue*. Verify that during the execution of Algorithm 7.1, a state (s, t) of EM will be visited only if t is an observation of some reachable state of K. Secondly, during determinization of the automaton EM, a consistent region of EM contains at most n states, and the automaton ΔEM has at most $\mathbf{2}^n \cdot \mathbf{2}^k$ states. Thus, the number of consistent subsets is exponential in the size of the specification automaton M, rather than exponential in the size of the expansion EM. These two observations lead to the following bound on the Sal model checking.

Theorem 7.4 [Sal model checking] *Let K be a finite observation structure with n reachable states and m reachable transitions, and let M be a finite* Sal *specification automaton with k states. Algorithm 7.1 solves the model-checking problem $(K, \forall M)$ in time $O((n + m) \cdot \mathbf{2}^k)$.*

Remark 7.8 [Space complexity of Sal model checking] The Sal model checking problem (K, ϕ) is Pspace-complete in the size of ϕ. ■

While the cost of Sal model checking is high compared to Stl model checking, this is so only in terms of the size of the specification. Since, in practice, the size of the observation structure is a computational bottleneck, while specifications are small, it is fruitful to note the *model complexity* of the model checking problem, namely, the parametric complexity in terms of the size of the structure assuming constant-size specifications.

Remark 7.9 [Model complexity of Sal model checking] Algorithm 7.1 yields a solution to the Sal model checking problem (K, ϕ) with linear-time model complexity. This coincides with the model complexity of the enumerative Stl model checking algorithms of Chapter 5. The model complexity of Sal model checking, as well as Stl model checking, is Nlogspace. ■

Sal **verification**

The Sal verification problem $(P, \forall M)$ is propositional if P is a propositional module, M is a finite automaton, and the observations of M are propositional formulas. Algorithm 7.1 can be used to solve the Sal verification problem $(P, \forall M)$ for an automaton specification M whose observations are boolean expressions over $\mathsf{obs}X_P$. First, we construct on-the-fly representations of the observation structures K_P and EM, and then, use Algorithm 7.1 to test language inclusion. The complexity is exponential in both the number of variables of P and the number of states of M. As usual, the verification problem is exponentially harder than the corresponding model checking problem.

Remark 7.10 [Space complexity of SAL verification] The SAL verification problem $(P, \forall\phi)$ is PSPACE-complete both in the size of the description of P and in the number of states of ϕ. ■

Deterministic specifications

The exponential complexity of the test for language-inclusion (M_1, M_2) disappears if M_2 is deterministic. When M_2 is deterministic, then Algorithm 7.1 solves the language-inclusion problem (M_1, M_2) in time $O((n_1 + m_1) \cdot n_2)$. Consequently, the SAL model checking problem $(K, \forall M)$ is computationally easier when the expansion EM is deterministic. The expansion EM is guaranteed to be deterministic if

1. the observations of every pair of initial states of M are mutually exclusive (i.e. if s and t are initial states, the observation of s is the expression p, and the observation of t is the expression q, then the conjunction $p \wedge q$ is an unsatisfiable formula), and
2. for every state s of M, the observations of every pair of successor states of s are mutually exclusive.

In such a case, the complexity of SAL model checking $(K, \forall M)$ is $O(|K| \cdot |M|)$.

There is another advantage of restricting to deterministic specification automata: the language-inclusion problem (M_1, M_2) can be solved symbolically if the automaton M_2 is deterministic.

Exercise 7.16 {P3} [Symbolic language inclusion] Consider the language-inclusion problem (K_1, K_2) when the observation structure K_1 is represented symbolically, and the observation structure K_2 is represented enumeratively. Write an algorithm for solving the language-inclusion problem that combines symbolic reachability analysis of K_1 with determinization of K_2. ■

Complemented specifications

Example 7.3 shows two ways of writing a requirement in SAL. A mutual-exclusion protocol satisfies the exclusion requirement if it satisfies the SAL formula $\forall M_{mutex}$, or equivalently, the formula $\neg\exists M_{\neg mutex}$. Checking whether an observation structure satisfies $\forall M_{mutex}$ requires complementing M_{mutex}, and hence, determinizing it. On the other hand, an observation structure satisfies $\neg\exists M_{\neg mutex}$ if the product structure $K \times EM_{\neg mutex}$ has an empty language, and thus, can be checked without determinization. This holds whenever the SAL specification is given as a negation of an automaton that accepts the undesirable behaviors.

Proposition 7.11 [Checking negated automata] *The answer to the* SAL *model checking problem* $(K, \neg\exists M)$ *is* YES *iff the language* $L_{K \times EM}$ *is nonempty.*

Consequently, the Sal model checking problem $(K, \neg\exists M)$ can be solved in time $O(|K| \cdot |M|)$ by exploring the product of K and EM on-the-fly. Thus, requiring the user to specify the automaton accepting the undesirable behaviors, rather than the automaton accepting desirable behaviors, simplifies the model checking task.

Contents

Chapter 8

Hierarchical Verification

In hierarchical design, we construct various models of a system at different levels of detail. The key verification issue, then, is to check that a detailed model P of the system conforms with a more abstract model Q. If both P and Q are reactive modules, then our notion of conformance requires that every finite sequence of observations $\overline{a}$ that may result from executing the detailed module P may also result from executing the more abstract module Q. In this case, we say that the module P *implements* the module Q.

8.1 Implementation of Reactive Modules

Given a reactive module P, an *initialized trace* of P is an initialized trace of the observation structure K_P, and the *language* L_P of the module P is the language L_{K_P} of the corresponding observation structure. The interaction of a module P with its environment is completely determined by the set $\mathsf{intf}X_P$ of interface variables, the set $\mathsf{extl}X_P$ of external variables, the awaits dependencies among the observable variables, and the set L_P of traces. Two modules that agree on these components will interact with other modules in the same way irrespective of their branching structures. This is illustrated by the following proposition which asserts that the traces of a compound module are completely determined by the traces of its components. In particular, if P and Q have identical observations then $L_{P\,\|\,Q}$ equals $L_P \cap L_Q$.

Proposition 8.1 [Traces of compound modules] *Let P and Q be compatible modules, and let $\overline{a}$ be a word over the observations of the compound module $P \,\|\, Q$. Then, $\overline{a}$ belongs to the language $L_{P\,\|\,Q}$ iff the projection $\mathsf{obs}X_P[\overline{a}]$ belongs to L_P and the projection $\mathsf{obs}X_Q[\overline{a}]$ belongs to L_Q.*

Exercise 8.1 {`t2`} [Traces of compound modules] Prove Proposition 8.1. ■

This leads to a natural way of comparing two modules.

Implementation

The reactive module P *implements* the reactive module Q, denoted $P \preceq^L Q$, if

1. every interface variable of Q is an interface variable of P: $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$,
2. every external variable of Q is an observable variable of P: $\mathsf{extl}X_Q \subseteq \mathsf{obs}X_P$,
3. for all variables x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and
4. if $\overline{a}$ is an initialized trace of P then the projection $\overline{a}[\mathsf{obs}X_Q]$ of $\overline{a}$ onto the observable variables of Q is an initialized trace of Q.

The two modules P and Q are *trace equivalent*, denoted $P \simeq^L Q$, if $P \preceq^L Q$ and $Q \preceq^L P$.

Intuitively, if $P \preceq^L Q$ then the module P is *as complex as* the module Q: P has possibly more interface and external variables than Q, P has more await dependencies among its observable variables, and has less traces than Q, and thus, more constraints on its execution. The superscript L in the implementation relation $\preceq^L$ indicates that this relation is based on the languages of the modules.

Remark 8.1 [Implementation preorder] The implementation relation $\preceq^L$ on modules is reflexive and transitive. ■

Example 8.1 [Synchronous versus asynchronous mutual exclusion] The module *SyncMutex* of Figure 1.22 gives the synchronous solution to the mutual exclusion problem, and the module *Pete* of Figure 1.23 gives the asynchronous solution. Both the modules have identical interface variables, no external variables, and no await dependencies. Verify that every trace of *SyncMutex* is a trace of *Pete*, and thus, *SyncMutex* $\preceq^L$ *Pete*. However, the two modules are not trace equivalent. The word

$$(pc_1 = pc_2 = outC), (pc_1 = reqC, pc_2 = outC), (pc_1 = pc_2 = reqC)$$

is a trace of *Pete*, but is not a trace of *SyncMutex*. Intuitively, the asynchronous solution is more abstract, and admits more traces. ■

Example 8.2 [Nondeterministic versus deterministic scheduling] Recall the module *Scheduler* from Figure 1.4 consisting of atoms $A3$, $A4$, and $A5$. Consider the atom $A6$

```
$A6$: atom controls $proc$ reads $task_1, task_2$
  update
    $[\!]$ $task_1 = 0 \wedge task_2 = 0 \rightarrow proc' := 0$
    $[\!]$ $task_2 > 0$                     $\rightarrow proc' := 2$
    $[\!]$ $task_1 > 0$                     $\rightarrow proc' := 1$
```

When both the tasks are pending the atom $A6$ assigns the processor to one of them in a nondeterministic fashion. The module *NonDetScheduler* is like *Scheduler* with the atom $A5$ replaced by the atom $A6$ (*NonDetScheduler* no longer needs the private variable *prior*). The two modules *Scheduler* and *NonDetScheduler* have identical interface and external variables, and identical awaits dependencies among their observable variables. Verify that every trace of *Scheduler* is a trace of *NonDetScheduler*, but not vice versa. We have, *Scheduler* $\preceq^L$ *NonDetScheduler*. The module *Scheduler* is an implementation of the specification *NonDetScheduler*. The specification only requires that if one of the tasks is pending, then the processor should be assigned to a pending task, but does not specify a policy to resolve the contention when both tasks are pending. The implementation refines the specification by implementing a deterministic policy using the variable *prior*. ■

Example 8.3 [Binary counter specification] Recall the example of the sequential circuit for three-bit binary counter from Example 1.16. The counter takes two boolean inputs, represented by the external variables *start* and *inc*, for starting and incrementing the counter. The counter value ranges from 0 to 7, and is represented by three interface binary variables out_0, out_1, and out_2. The *specification* of the counter is the module *Sync3BitCounterSpec* of Figure 8.1. The module *Sync3BitCounter* of Figure 1.19 is a possible implementation. The correctness of the design *Sync3BitCounter* with respect to the specification *Sync3BitCounterSpec* is expressed by the fact that the two modules are trace-equivalent. ■

Exercise 8.2 {P2} [Zero-delay vs. unit-delay vs. buffered vs. lossy squaring] Recall the definitions of the modules *SyncSquare*, *SyncSquare2*, *AsyncSquare*, and *LossyAsyncSquare* (see Example 1.10, and Example 1.11) all of which compute squares of input numbers. Which pairs of modules among these four modules are related by the implementation relation $\preceq^L$? ■

Exercise 8.3 {P2} [Synchronous vs. asynchronous message passing] Consider the module *SyncMsg* of Figure 1.25 for synchronous message passing, and the module *AsyncMsg* of Figure 1.30 for asynchronous message passing. Does *SyncMsg* implement *AsyncMsg*? Are the two modules trace equivalent? ■

```
**module** $\mathit{Sync3BitCounterSpec}$ **is**
  **interface** $out_0, out_1, out_2$
  **external** $start, inc$
  **atom controls** $out_0$ **reads** $out_0$ **awaits** $start, inc$
    **update**
      $[\!]\ start' = 1 \qquad\qquad\qquad \rightarrow out'_0 := 0$
      $[\!]\ start' = 0 \wedge inc' = 1 \rightarrow out'_0 := \neg out_0$
  **atom controls** $out_1$ **reads** $out_0, out_1$ **awaits** $start, inc$
    **update**
      $[\!]\ start' = 1 \qquad\qquad\qquad \rightarrow out'_1 := 0$
      $[\!]\ start' = 0 \wedge inc' = 1 \rightarrow out'_1 := out_0 \oplus out_1$
  **atom controls** $out_2$ **reads** $out_0, out_1, out_2$ **awaits** $start, inc$
    **update**
      $[\!]\ start' = 1 \qquad\qquad\qquad \rightarrow out'_2 := 0$
      $[\!]\ start' = 0 \wedge inc' = 1 \rightarrow out'_2 := (out_0 \wedge out_1) \oplus out_2$
```

Figure 8.1: Specification of three-bit counter

Composing two modules using parallel composition creates a module that is more complex than its components, while hiding a variable creates a simpler module with less number of observable variables.

Proposition 8.2 [Module operations and implementation] *(1) For compatible reactive modules P and Q, $P \,\|\, Q \preceq^L P$. (2) For a variable x and a reactive module P, $P \preceq^L$* **hide** *x* **in** *P.*

Exercise 8.4 {T1} [Module operations and implementation] Prove Proposition 8.2. ■

The implementation problem

The implementation problem asks whether one module implements another module.

> THE IMPLEMENTATION PROBLEM
>
> An instance (P, Q) of the *implementation problem* consists of two reactive modules P and Q. The answer to the implementation problem (P, Q) is YES if P implements Q, and otherwise NO.

Recall that the module P implments the module Q if (1) $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$, (2) $\mathsf{extl}X_Q \subseteq \mathsf{extl}X_P$, (3) for all x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and (4) if $\overline{a}$ is a trace of P then $\overline{a}[\mathsf{obs}X_Q]$ is a trace of Q. Checking first three conditions is easy, and the fourth condition reduces to the language inclusion problem. Thus, the implementation problem relates to the language-inclusion problem in the same way in which the invariant-verification problem relates to the reachability problem, and in which the STL verification problem relates to the STL model-checking problem.

Consider two modules P and Q such that the first three conditions for P to implement Q are met. Let $P' = \textbf{hide}\ (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q)\ \textbf{in}\ P$. Then P' and Q have identical observable variables. The module P implements Q if every trace of P' is also a trace of Q, that is, if the answer to the language-inclusion problem $(K_{P'}, K_Q)$ is YES. The complexity of solving the language-inclusion question is exponential in its second argument. The next theorem concerning checking implementation relation between two propositional modules follows.

Theorem 8.1 [The implementation problem] *Let P be a propositional module with k propositional variables and let Q be a propositional module with ℓ propositional variables. The propositional implementation problem (P, Q) can be solved using the language-inclusion algorithm in time $O(4^k \cdot 2^{2^\ell})$.*

Remark 8.2 [Space complexity of implementation problem] The propositional implementation problem (P, Q) is EXPSPACE hard in its second argument. Checking implementation requires searching the product of the observation structure of P and the determinization of the the observation structure of Q, and can be performed in space $O(k \cdot 2^\ell)$ if P has k variables and Q has ℓ variables. It follows that propositional implementation problem is complete for EXPSPACE. ■

If Q is an observably-deterministic module, then so is the observation structure K_Q. In this case, the language inclusion question $(K_{P'}, K_Q)$ can be solved without determinization.

Theorem 8.2 [Deterministic case of implementation problem] *Let P be a propositional module with k propositional variables and let Q be a propositional observably-deterministic module with ℓ propositional variables. The propositional implementation problem (P, Q) can be solved in time $O(4^{k+\ell})$.*

Remark 8.3 [Deterministic case of space complexity of implementation proble] If Q is observably-deterministic, then the propositional implementation problem (P, Q) is PSPACE-complete. ■

8.1.1 From Bisimilarity to Implementation

Given two observation structures K_1 and K_2, because the language-inclusion problem (K_1, K_2) is hard, in practice, it is important to find sufficient conditions for $L_{K_1} \subseteq L_{K_2}$ that can be checked efficiently. One such sufficient condition is bisimilarity.

State preorders

A *state preorder* $\preceq$ is a family of preorders, one preorder $\preceq_K$ on the of each observation structure K. The *trace preorder* $\preceq^L$ is the following state preorder: for two states s and t of an observation structure K, let $s \preceq^L_K t$ iff $L_K(s) \subseteq L_K(t)$.

A state preorder allows us to compare states. To compare two observation structures with identical observations using a state preorder, we consider the disjoint union of the two structures, and check if every initial state of one is related to some initial state of the other. It also leads to a way of comparing two reactive modules.

Structure and module preorders of a state preorder

Let $\preceq$ be a state preorder. Let K_1 and K_2 be two disjoint observation structures. Let σ^I_1 be the initial region of K_1, and let σ^I_2 be the initial region of K_2. Then, $K_1 \preceq K_2$ if for all states $s \in \sigma^I_1$, there is a state $t \in \sigma^I_2$ such that $s \preceq_{K_1+K_2} t$.

For two modules P and Q, $P \preceq Q$ if (1) every interface variable of Q is an interface variable of P, (2) every external variable of Q is an observable variable of P, (3) for all variables x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and (4) $K_{P'} \preceq K_Q$ for $P' = \textbf{hide } (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q) \textbf{ in } P$.

Observe that the structure preorder relates two structures only when they have identical observations, and hence, when comparing two modules we use hiding before we compare the corresponding observation structures. For the bisimilarity relation, $K_1 \preceq^B K_2$ means that every initial state of K_1 is bisimilar to some initial state of K_2.

Remark 8.4 [Bisimulation preorder] Since bisimilarity is an equivalence relation over states, if $K_1 \preceq^B K_2$ and both structures have unique initial states, then $K_2 \preceq^B K_1$. If the two modules P and Q have identical interface and external variables, and unique initial states, then if $P \preceq^B Q$ then $Q \preceq^B P$. ■

Exercise 8.5 {T1} [Checking bisimilarity preorder] Given two observation structures K_1 and K_2, what is the time complexity of checking $K_1 \preceq^B K_2$? ■

For the trace preorder, the induced preorder over observation structures is language inclusion, and the induced preorder over modules is implementation. Since bisimilar states have identical languages, proving bisimilarity preorder is a sufficient condition for proving implementation. However, it is not a necessary condition, because bisimilarity is more distinguishing than language equivalence.

Proposition 8.3 [Trace and bisimilarity preorders] *For two observation structures K_1 and K_2, if $K_1 \preceq^B K_2$, then $K_1 \preceq^L K_2$. For two reactive modules P and Q, if $P \preceq^B Q$ then $P \preceq^L Q$.*

Example 8.4 [Trace and bisimilarity preorders] In Example 8.1, we noted that *SyncMutex* $\preceq^L$ *Pete*. However, *SyncMutex* $\preceq^B$ *Pete* does not hold.

In Example 8.2, we noted that *Scheduler* $\preceq^L$ *NonDetScheduler*. Hopwever, *Scheduler* $\preceq^B$ *NonDetScheduler* does not hold.

In Example 8.3, we noted that *Sync3BitCounter* and *Sync3BitCounterSpec* are trace-equivalent. Verify that the two modules are equivalent according to the bisimilarity preorder also. ■

8.2 Compositional Reasoning

8.2.1 Compositionality

If we prove that a module P implements another module Q, can we substitute P for Q in all contexts? Compositional proof rules admit such deductions, thereby reducing reasoning about compound modules to reasoning about the component modules.

Compositionality

The preorder $\preceq$ on reactive modules is *compositional* if for all modules P and Q, if $P \preceq Q$ then

1. for every reactive module R that is compatible with P, R is compatible with Q and $P \| R \preceq Q \| R$;
2. for variable x of P, **hide** x **in** $P \preceq$ **hide** x **in** Q;
3. for every variable renaming ρ, $P[\rho] \preceq Q[\rho]$.

A compositional equivalence on modules is called a *module congruence.*

Remark 8.5 [Congruence] If $\preceq$ is a compositional preorder on modules, then the symmetric closure of $\preceq$ is a module congruence. ■

Theorem 8.3 [Compositionality of implementation] *The implementation preorder $\preceq^L$ on modules is compositional.*

Proof. Consider two reactive modules P and Q such that $P \preceq^L Q$. The cases corresponding to the operations of hiding and renaming are straightforward.

We consider only parallel composition. Let R be a reactive module that is compatible with P.

First, we need to establish that Q and R are compatible. Since $\mathsf{intf}X_P \cap \mathsf{intf}X_R$ is empty, and $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$, we conclude that $\mathsf{intf}X_Q \cap \mathsf{intf}X_R$ is empty. Asymmetricity of $(\prec_Q \cup \prec_R)^+$ follows from (1) $(\prec_P \cup \prec_R)^+$ is asymmetric, and (2) for two variables $x, y \in \mathsf{obs}X_Q$, if $x \prec_Q y$ then $x, y \in \mathsf{obs}X_P$ with $x \prec_P y$.

Next, we establish that every initialized trace of $P \parallel R$ is also an initialized trace of $Q \parallel R$. Let $\overline{a}$ be an initialized trace of $P \parallel R$. From Proposition 8.1, $\mathsf{obs}X_P[\overline{a}]$ is an initialized trace of P and $\mathsf{obs}X_R[\overline{a}]$ is an initialized trace of R. Since $P \preceq^L Q$, $\mathsf{obs}X_Q[\overline{a}]$ is an initialized trace of Q. Again, from Proposition 8.1, $\mathsf{obs}X_{Q \parallel R}[\overline{a}]$ is an initialized trace of $Q \parallel R$. ■

Corollary 8.1 [Congruence of trace equivalence] *Trace equivalence is a module congruence.*

Suppose that we wish to prove that a compound module $P_1 \| P_2$ implements the abstraction $Q_1 \| Q_2$, where Q_1 is an abstraction of P_1 and Q_2 is an abstraction of P_2. By Theorem 8.3, it suffices to prove separately that (1) the component module P_1 implements Q_1, and (2) the component module P_2 implements Q_2. Both proof obligations (1) and (2) involve smaller state spaces than the original proof obligation.

Example 8.5 [Compositional proof] Consider the synchronous message-passing protocol

$$\mathit{SyncMsg} \;=\; \textbf{hide } \mathit{ready}, \mathit{transmit}, \mathit{msg}_S \textbf{ in } \mathit{SyncSender} \parallel \mathit{Receiver}$$

and the asynchronous message passing protocol

$$\mathit{AsyncMsg} \;=\; \textbf{hide } \mathit{ready}, \mathit{transmit}, \mathit{msg}_S \textbf{ in } \mathit{AsyncSender} \parallel \mathit{Receiver}.$$

Suppose we wish to establish that the synchronous protocol is an implementation of the asynchronous one:

$$\mathit{SyncMsg} \;\preceq^L\; \mathit{AsyncMsg}.$$

Then, since $\preceq^L$ is compositional, it suffices to establish that

$$\mathit{SyncSender} \;\preceq^L\; \mathit{AsyncSender}.$$

Verify that synchronous sender *SyncSender* indeed is an implementation of the asynchronous sender *AsyncSender*. ■

Exercise 8.6 {T3} [Bisimilarity congruence] Prove that the bisimulation preorder $\preceq^B$ is compositional. ■

It also follows that the state logic Sal is compositional:

Corollary 8.2 [Compositionality of Sal] *If a reactive module P satisfies an* Sal *formula ϕ, then every compound module $P \parallel Q$ also satisfies the* Sal *formula ϕ.*

8.2.2 Assume-guarantee Reasoning

Compositional proof rules, while useful, may not always be applicable. In particular, P_1 may not implement Q_1 for all environments, but only if the environment behaves like P_2, and vice versa. In this case, *assumption-guarantee proof rules* are needed. An assume-guarantee proof rule asserts that in order to prove that $P_1 \| P_2$ implements $Q_1 \| Q_2$, it suffices to prove (1) $P_1 \| Q_2$ implements Q_1, and (2) $Q_1 \| P_2$ implements Q_2. Both proof obligations (1) and (2) typically involve smaller state spaces than the original proof obligation, because the compound module $P_1 \| P_2$ usually has the largest state space involved. Observe the circular nature of the assume-guarantee reasoning. Its correctness depends crucially on the fact that a module does not constrain the behavior of its environment, and thus, interacts with other modules in a non-blocking way.

Theorem 8.4 [Assume-guarantuee reasoning] *Let P_1 and P_2 be two compatible reactive modules, and let Q_1 and Q_2 be two compatible reactive modules. If $P_1 \| Q_2 \preceq^L Q_1$, $Q_1 \| P_2 \preceq^L Q_2$, and every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$, then $P_1 \| P_2 \preceq^L Q_1 \| Q_2$.*

Proof. Consider four modules P_1, P_2, Q_1, and Q_2 such that

1. P_1 and P_2 are compatible,
2. Q_1 and Q_2 are compatible,
3. every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$,
4. $P_1 \| Q_2 \preceq^L Q_1$, and
5. $Q_1 \| P_2 \preceq^L Q_2$.

We wish to establish that $P_1 \| P_2 \preceq^L Q_1 \| Q_2$. The definition of implementation has four requirements. Let us consider these four goals one by one.

Goal 1: To show that every interface variable of $Q_1 \| Q_2$ is an interface variable of $P_1 \| P_2$, let x be an interface variable of $Q_1 \| Q_2$. Due to symmetry, it suffices to consider the case that x is an interface variable of Q_1. By assumption (4), x is an interface variable of $P_1 \| Q_2$. The assumption (2) implies that x is not an interface variable of Q_2. It follows, from the definition of parallel composition, that x is an interface variable of P_1, and hence, of $P_1 \| P_2$.

Goal 2: The second requirement that every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$ is assumption (3).

Goal 3: We wish to show that if $y \prec_{Q_1 \| Q_2} x$ then $y \prec_{P_1 \| P_2} x$. Since $\prec_{Q_1 \| Q_2}$ is the transitive closure of the union $\prec_{Q_1}$ and $\prec_{Q_2}$, and by symmetry, it suffices to prove that if $y \prec_{Q_1} x$ then $y \prec_{P_1 \| P_2} x$.

Consider an interface variable y and an observable variable of x of Q_1 such that $y \prec_{Q_1} x$. From assumption (4), we have $y \prec_{P_1 \| Q_2} x$. We know that $\prec_{P_1 \| Q_2}$ is the transitive closure of the union of $\prec_{P_1}$ and $\prec_{Q_2}$. Hence, whenever $y \prec_{Q_1} x$, there is a finite chain of awaits dependencies such that

$$y \prec_{P_1} y_1 \prec_{Q_2} y_2 \prec_{P_1} \cdots \ x \quad (\dagger)$$

Similarly, if $y \prec_{Q_2} x$, then there exists a chain of awaits dependencies

$$y \prec_{P_2} y_1 \prec_{Q_1} y_2 \prec_{P_2} \cdots \ x \quad (\ddagger)$$

By repeatedly applying (†) and (‡), since the awaits relations are acyclic, and the number of variables is finite, if $y \prec_{Q_1} x$, then there exists a finite chain of awaits dependencies

$$y \prec_{P_1} y_1 \prec_{P_2} y_2 \prec_{P_1} \cdots \ x$$

and thus, $y \prec_{P_1 \| P_2} x$.

Goal 4: We wish to establish that every trace of $P_1 \| P_2$ is also a trace of $Q_1 \| Q_2$. We start by defining some additional concepts. For simplicity, in the following we omit explicit projections. For instance, if X is a superset of $\mathsf{obs}X_P$, and $\overline{s}$ is sequence of valuations for X such that $\overline{s}[\mathsf{obs}X_P]$ is a trace of P, then we consider $\overline{s}$ also to be a trace of P.

Given a module P, a subset $X \subseteq \mathsf{obs}X_P$ of the observable variables is *awaits-closed* if, whenever $y \prec_P x$ and $y \in X$, then $x \in X$. For an awaits-closed set X, the pair $(\overline{s}_{0\ldots m}, t)$ consisting of a trace $\overline{s}_{0\ldots m} \in L_P$ of P and a valuation t for X is said to be a X*-partial-trace* of P if there exists an observation s_{m+1} such that (1) $s_{m+1}[X] = t$, and (2) $\overline{s}_{0\ldots(m+1)} \in L_P$. Thus, partial-traces are obtained by executing only some of the subrounds of the last update round. The following facts about partial-traces follow from the definition of reactive modules.

1. If $P \preceq^L Q$ and X is awaits-closed for P, then X is awaits-closed for Q. If $P \preceq^L Q$, and $(\overline{s}, t)$ is a X-partial-trace of P, then $(\overline{s}, t)$ is a X-partial-trace of Q. Thus, inclusion of traces is equivalent to inclusion of partial traces.

2. The partial-traces of a compound module are determined from the partial-traces of the components: $(\overline{s}, t)$ is a X-partial-trace of $P \| Q$ iff it is a X-partial-trace of both P and Q.

3. If $(\overline{s}, t)$ is a X-partial-trace of P, and u is a valuation for a subset Y of the external variables of P, then $(\overline{s}, t \cup u)$ is a $(X \cup Y)$-partial-trace of P. This property is due the nonblocking nature of reactive modules.

Let $X_1, \ldots X_k$ be a partitioning of $\mathsf{obs}X_{P_1 \| P_2}$ into disjoint subsets such that (1) each X_i either contains only external variables of $P_1 \| P_2$ or contains only interface variables of P_1 or only interface variables of P_2, and (2) $x \prec_{P_1 \| P_2} y$

and $x \in X_i$ then $y \in X_j$ for some $j < i$. Let $Y_0 = \emptyset$, and for $0 \leq i < k$, $Y_{i+1} = Y_i \cup X_i$. Each such set Y_i is awaits-closed. Let $\mathcal{L}$ be the set of pairs $(\overline{s}, t)$ such that $(\overline{s}, t)$ is a X-partial-trace of $P_1 \| P_2$ for $X = Y_j$ for some $0 \leq j \leq k$. Define an ordering $<$ over $\mathcal{L}$: if $(\overline{s}, t)$ is a Y_j-partial-trace with $j < k$, and $(\overline{s}, u)$ is a Y_{j+1}-partial-trace with $u[Y_j] = t$ then $(\overline{s}, t) < (\overline{s}, u)$; and $(\overline{s}, t)$ is a Y_k-partial-trace then $(\overline{s}, t) < (\overline{s} \cdot t, \emptyset)$. Clearly, the ordering $<$ is well founded. By well-founded induction with respect to $<$, we prove that every partial-trace in $\mathcal{L}$ is a partial-trace of $Q_1 \| Q_2$.

Consider $(\overline{s}, \emptyset)$ in $\mathcal{L}$. If $\overline{s}$ is the empty trace, then $(\varepsilon, \emptyset)$ is a partial-trace of all modules. Otherwise, $\overline{s}$ is nonempty: $\overline{s} = \overline{t} \cdot u$. Then $(\overline{t}, u)$ is a Y_k-partial-trace of $P_1 \| P_2$. Since $(\overline{t}, u) < (\overline{s}, \emptyset)$, by induction hypothesis, $(\overline{t}, u)$ is a partial-trace of $Q_1 \| Q_2$, and hence, so is $(\overline{s}, \emptyset)$.

Consider $(\overline{s}, t)$ in $\mathcal{L}$ such that t is a valuation for Y_{j+1} for some $0 \leq j < k$. Let $u = t[Y_j]$. Then, $(\overline{s}, u) < (\overline{s}, t)$. By induction hypothesis, $(\overline{s}, u)$ is a Y_j-partial-trace of $Q_1 \| Q_2$. By the property (2) of partial-traces, $(\overline{s}, u)$ is a Y_j-partial-trace of both Q_1 and Q_2. Consider $Y_{j+1} = Y_j \cup X_j$. We know that X_j contains interface variables of at most one of P_1 and P_2. Without loss of generality, let us assume that X_j contains no interface variables of P_2, and hence, no interface variables of Q_2. By property (3) of partial-traces, the Y_j-partial-trace $(\overline{s}, u)$ of Q_2 can be extended with any valuation for X_j. In particular, $(\overline{s}, t)$ is a Y_{j+1}-partial-trace of Q_2. Hence, $(\overline{s}, t)$ is a Y_{j+1}-partial-trace of $P_1 \| Q_2$. Since $P_1 \| Q_2 \preceq^L Q_1$, and by property (1) of partial-traces, $(\overline{s}, t)$ is a partial-trace of Q_1. Again, by property (2) of partial-traces, $(\overline{s}, t)$ is a partial-trace of $Q_1 \| Q_2$. ■

Example 8.6 [Assume guarantee reasoning] To illustrate the application of assume guarantee reasoning, we consider a simple version of the alternating-bit protocol. The sender process is the module *ABPSender* of Figure 8.2. The private variable x indicates the bit to be sent with the next message. The message is transmitted by issuing the interface event $\mathit{transmit}_S$, and updating the variables *abp* and *msg* to the message contents. The acknowledgements issued by the receiver are stored in the private buffer z. After sending the message, the process removes an acknowledgement from z. If the acknowledgement equals the current value of the alternating-bit x, the sender concludes a correct delivery of the message, and updates the alternating-bit x.

The receiver process *ABPReceiver* is symmetric, and is shown in Figure 8.3 The messages received from the sender are stored in the private buffer z (for simplicity, the message is ignored, and the alternating-bit is stored). The process removes the first message in z in the variable x, which is, then, issued at a later time along with the interface event $\mathit{transmit}_R$.

Consider the module $\mathit{ABP} = \mathit{ABPSender} \parallel \mathit{ABPReceiver}$. The observable behavior of *ABP* is very regular: first the sender issues $\mathit{transmit}_S$ with the bit 0,

```
module $ABPSender$ is
  interface $transmit_S\colon$  ; $abp\colon \mathbb{B}$; $msg\colon$
  external $transmit_R\colon$  ; $ack\colon \mathbb{B}$
  private $consume\colon$  ; $pc\colon \{send, wait\}$; $x\colon \mathbb{B}$; $z\colon$ queue of $\mathbb{B}$
  passive atom
    controls $z$
    reads $consume, transmit_R$
    awaits $consume, transmit_R, ack$
    init
      $\| \ true \to z' := EmptyQueue$
    update
      $\| \ consume? \wedge transmit_R? \quad \to z' := Enqueue(ack', Dequeue(z))$
      $\| \ consume? \wedge \neg transmit_R? \to z' := Dequeue(z)$
      $\| \ \neg consume? \wedge transmit_R? \to z' := Enqueue(ack', z)$
  lazy atom
    controls $consume, x$
    reads $pc, z, x, consume$
    init
      $\| \ true \to x' := 0$
    update
      $\| \ pc = wait \wedge z \neq EmptyQueue \wedge x = Front(z) \to consume!$
      $\| \ pc = wait \wedge z \neq EmptyQueue \wedge x \neq Front(z) \to consume!;\ x' := \neg x$
  passive atom
    controls $pc$
    reads $pc, consume, transmit_S$
    awaits $consume, transmit_S$
    init
      $\| \ true \to pc' := send$
    update
      $\| \ pc = send \wedge transmit_S? \to pc' := wait$
      $\| \ pc = wait \wedge consume? \quad \to pc' := send$
  lazy atom
    controls $transmit_S, msg, abp$
    reads $pc, transmit_S, x$
    update
      $\| \ pc = send \to transmit_S!;\ abp' := x;\ msg' :=$
```

Figure 8.2: Sender process of Alternating-bit Protocol

then the receiver issues $\mathit{transmit}_R$ with the acknowledgement 0, then the sender issues $\mathit{transmit}_S$ with the bit 1, then the receiver issues $\mathit{transmit}_R$ with the acknowledgement 1.

Figure 8.4 shows simpler abstract versions of the sender and receiver. The module *AbstractSender* differs from the module *ABPSender* in two ways. It assumes that (1) it will always recieve the correct acknowledgements, and (2) the acknowledgement events issued by the receiver strictly alternate with the events issued by the sender. Consequently, it does not read the values of the acknowledgements, and it does not buffer the acknowledgements. The process *AbstractReceiver* is a similar simplification of *ABPReceiver*.

Suppose we wish to establish that

$$\mathit{ABPSender} \parallel \mathit{ABPReceiver} \preceq^L \mathit{AbstractSender} \parallel \mathit{AbstractReceiver}. \quad (\dagger)$$

Compositionality cannot simplify this goal, becuase neither *ABPSender* implement *AbstractSender*, nor does *ABPReceiver* implement *AbstractReceiver*. However, verify that both

$$\mathit{ABPSender} \parallel \mathit{AbstractReceiver} \preceq^L \mathit{AbstractSender},$$

and

$$\mathit{AbstractSender} \parallel \mathit{ABPReceiver} \preceq^L \mathit{AbstractReceiver}$$

hold. Then, by assume-guarantee theorem we can conclude the obligation (†). ■

Exercise 8.7 {`T2`} [Side condition in assume-guarantee rule] Show that the assumption that every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$ is essential (i.e. it does not follow from the assumptions (1), (2), (4), and (5) in the proof of the assume-guarantee theorem. ■

Exercise 8.8 {`T3`} [Assume-guarantee for bisimilarity] Does Theorem 8.4 hold for the bisimulation preorder $\preceq^B$? ■

8.3 Simulation Relations

Establishing trace preorder between two observation structures is computationally hard. While bisimilarity of two structures is a sufficient condition to establish trace preorder, bisimilarity over structures in an equivalence relation, and does not admit the implementation to have less traces than the specification. Simulation relations offer a practical alternative: on one hand, computing simulation relations is computationally easier than establishing trace preorder,

```
**module** $ABPReceiver$ **is**
  **external** $transmit_S:\ ;\ abp:\mathbb{B};\ msg:$
  **interface** $transmit_R:\ ;\ ack:\mathbb{B}$
  **private** $consume:\ ;\ pc:\{send, wait\};\ x:\mathbb{B};\ z:$ **queue of** $\mathbb{B}$
  **passive atom**
    **controls** $z$
    **reads** $consume, transmit_S$
    **awaits** $consume, transmit_S, abp$
    **init**
      $\| \ true \rightarrow z' := EmptyQueue$
    **update**
      $\| \ consume? \wedge transmit_S? \quad \rightarrow z' := Enqueue(abp', Dequeue(z))$
      $\| \ consume? \wedge \neg transmit_S? \rightarrow z' := Dequeue(z)$
      $\| \ \neg consume? \wedge transmit_S? \rightarrow z' := Enqueue(abp', z)$
  **lazy atom**
    **controls** $consume, x$
    **reads** $pc, z$
    **update**
      $\| \ pc = wait \wedge z \neq EmptyQueue \rightarrow consume!;\ x' := Front(z)$
  **passive atom**
    **controls** $pc$
    **reads** $pc, consume, transmit_R$
    **awaits** $consume, transmit_R$
    **init**
      $\| \ true \rightarrow pc' := wait$
    **update**
      $\| \ pc = send \wedge transmit_R? \rightarrow pc' := wait$
      $\| \ pc = wait \wedge consume? \quad \rightarrow pc' := send$
  **lazy atom**
    **controls** $transmit_R, ack$
    **reads** $pc, transmit_R, x$
    **update**
      $\| \ pc = send \rightarrow transmit_R!;\ ack' := x$
```

Figure 8.3: Receiver process of Alternating-bit Protocol

```
module $AbstractSender$ is
  interface $transmit_S\colon\ ;\ abp\colon \mathbb{B}$
  external $transmit_R\colon$
  private $pc\colon \{send, wait\};\ x\colon \mathbb{B}$
  passive atom
    controls $pc$
    reads $pc, transmit_S, transmit_R$
    awaits $transmit_R, transmit_S$
    init
      $\| \ true \rightarrow pc' := send$
    update
      $\| \ pc = send \wedge transmit_S? \rightarrow pc' := wait$
      $\| \ pc = wait \wedge transmit_R? \rightarrow pc' := send$
  lazy atom
    controls $x, transmit_S, abp$
    reads $pc, x, transmit_S$
    init
      $\| \ true \rightarrow x' := 0$
    update
      $\| \ pc = send \rightarrow transmit_S!;\ abp' := x;\ x' := \neg x$

module $AbstractReceiver$ is
  external $transmit_S\colon$
  interface $transmit_R\colon\ ;\ ack\colon \mathbb{B}$
  private $pc\colon \{send, wait\};\ x\colon \mathbb{B}$
  passive atom
    controls $pc$
    reads $pc, transmit_S, transmit_R$
    awaits $transmit_R, transmit_S$
    init
      $\| \ true \rightarrow pc' := wait$
    update
      $\| \ pc = send \wedge transmit_R? \rightarrow pc' := wait$
      $\| \ pc = wait \wedge transmit_S? \rightarrow pc' := send$
  lazy atom
    controls $x, transmit_R, ack$
    reads $pc, x, transmit_R$
    init
      $\| \ true \rightarrow x' := 0$
    update
      $\| \ pc = send \rightarrow transmit_R!;\ ack' := x;\ x' := \neg x$
```

Figure 8.4: Abstract Sender and Receiver of Alternating-bit Protocol

while on the other hand, existence of simulation relations is much less stringent requirement compared to bisimilarity.

Simulation

Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. A *simulation* $\preceq$ of K is a binary relation on the state space such that for all states s and t of K, if $s \preceq t$ then (1) $\langle\!\langle s\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle$ and (2) if $s \rightarrow s'$, then there is a state t' such that $t \rightarrow t'$ and $s' \preceq t'$. The state t *simulates* the state s, denoted $s \preceq^S_K t$, if there is a simulation $\preceq$ such that $s \preceq t$.

From the definition of simulation relations, it follows that the union of two simulation relations is also a simulation relation.

Proposition 8.4 [Union-closure of simulation relations] *Let K be an observation structure with state-space Σ. For two simulation relations $\preceq_1$ and $\preceq_2$, their union $\preceq_1 \cup \preceq_2$ is a simulation relation.*

It follows that the set of simulation relations forms a complete partial-order with respect to the subset relation.

Corollary 8.3 [Maximal simulation] *For an observation structure K, the relation $\preceq^S_K$ is a simulation of K, and equals the union of all simulation relations of K.*

The maximal simulation relation $\preceq^S_K$ is reflexive and transitive, and thus, a state preorder. This follows from the fact that reflexive-transitive closure a simulation is also a simulation.

Proposition 8.5 [Simulation preorder] *For an observation structure K, if $\preceq$ is a simulation of K, then so is its reflexive-stransitive closure $\preceq^*$.*

Exercise 8.9 {`T2`} [Simulation preorder] Prove Proposition 8.5. ■

Recall the alternative definitions of bisimilarity from Chapter 6. Similarity relation can be also be explained in various ways. Let us consider the similarity game on the graph of the observation structure K. Player I, the *protagonist*, attempts to show that the state t simulates the state s, while Player II, the *adversary*, tries to establish otherwise. If the two given states have different observations, then the adversary wins immediately. Throughout the game, each player has an active state. Initially, the active state of the adversary is s, and the active state of the protagonist is t. In each move of the game, the adversary replaces its active state by one of its successors, say s'; the protagonist, then, must replace its own active state with one of its successors t' such that s' and t' have identical observations. If the protagonist cannot find such a replacement, then the adversary wins the game. The state t simulates the state s iff the adversary does not have a winning strategy; that is, all of possible moves of the adversary can perpetually be matched by the protagonist. Contrast this game with the bisimilarity game: the bisimilarity game has two active states at each step, the adversary chooses one of the two active states, and replaces it by one of its successors, and the protagonist is required to find a replacement for the other active state. Thus, similarity game is like bisimilarity game in which, the adversary starts playing from s and never switches sides.

Example 8.7 [Simulation game] Let us revisit the bisimilarity game of Example 6.9 (See Figure 6.7). States s_0 and u_0 are bisimilar, and hence, similar. States s_0 and t_0 are not bisimilar. Now let us consider the similarity game. Suppose initially the active state of the adversary is s_0, while the active state of the protagonist is t_0. Verify that the adversary has a winning strategy in this case. Consequently, the state t_0 does *not* simulate the state s_0, and $s_0 \not\preceq^S t_0$. On the other hand, suppose initially the active state of the adversary is t_0 and the active state of the protagonist is s_0. In this case, the protagonist can match every move of the adversary. In fact, $\{(t_0, s_0), (t_1, s_1), (t_2, s_1), (t_3, s_2), (t_4, s_3)\}$ is a simulation relation. Consequently, the state s_0 *does* simulate the state t_0, and $t_0 \preceq^S s_0$. ■

Remark 8.6 [Simulation vs. bisimulation] Recall that an equivalence relation $\cong$ on the states of an observation structure K is a bisimulation iff whenever $s \cong t$, (1) $\langle\langle s\rangle\rangle = \langle\langle t\rangle\rangle$, and (2) if $s \rightarrow s'$, then there is a state t' such that $t \rightarrow t'$ and $s' \cong t'$. It follows that the bisimulations of K are precisely the symmetric simulations of K; that is, the equivalence $\cong$ on the states of K is a bisimulation iff both $\cong$ and $\cong^{-1}$ are simulations of K. ■

To prove that the language of a state s is included in the language of a state t, it suffices to prove that t simulates s.

Theorem 8.5 [Simulation vs. language inclusion] *Let s and t be two states of an observation structure K. If $s \preceq^S_K t$, then $s \preceq^L_K t$.*

Proof. Consider two states s and t of K such that $s \preceq^S_K t$. Consider a source-s trajectory $\overline{s}_{0\ldots m}$. Let $t_0 = t$. For $i = 1, \ldots m$, by induction on i, since $s_{i-1} \preceq^S_K t_{i-1}$ and $s_{i-1} \rightarrow s_i$, there exists a state t_i such that $s_i \preceq^S_K t_i$ and $t_{i-1} \rightarrow t_i$. Thus, $\overline{t}_{0\ldots m}$ is a source-t trajectory of K. Furthermore, for all $0 \leq i \leq m$, $\langle\langle s_i\rangle\rangle = \langle\langle t_i\rangle\rangle$, and hence, $\langle\langle \overline{s}_{0\ldots m}\rangle\rangle$ is also a source-t trace of K. ■

The simulation preorder allows comparing two observations structures: for two observation structures K_1 and K_2 with disjoint state-spaces and identical observations, $K_1 \preceq^S K_2$ if for every initial state s of K_1, there exists an initial state t of K_2 such that $s \preceq^S_{K_1+K_2} t$.

Corollary 8.4 [Simulation preorder vs. trace preorder] *For two observation structures K_1 and K_2, if $K_1 \preceq^S K_2$, then $K_1 \preceq^L K_2$.*

Remark 8.7 [Simulation of reachable states] If $K_1 \preceq^S K_2$ then for every reachable state s of K_1, there exists a reachable state t of K_2 such that t simulates s. ■

It follows that the language-inclusion problem (K_1, K_2) can be solved by exhibiting a simulation $\preceq$ of $K_1 + K_2$ such that for every initial state s of K_1 there is an initial state t of K_2 with $s \preceq t$.

8.3.1 Similarity

Similarity
The state equivalence $\simeq^S$ induced by the simulation preorder $\preceq^S$ is called *similarity.*

Thus, $s \simeq^S t$ for two states s and t of the observation structure K if there exists a simulation $\preceq_1$ of K with $s \preceq_1 t$ and a simulation $\preceq_2$ of K with $t \preceq_2 s$. To observe that the similarity is more distinguishing than trace equivalence, consider states s and t of Figure 6.8 We have $s \simeq^L t$, but t does not simulate s.

Example 8.8 [Similarity vs. bisimilarity] To observe that the bisimilarity is more distinguishing than similarity, consider states s_0 and t_0 of Figure 8.5. The two states are not bisimilar. Observe that the relation

$$\{(s_0, t_0), (s_1, t_2), (s_2, t_4), (s_3, t_5)\}$$

is a simulation relation, and hence, t_0 simulates s_0. The relation

$$\{(t_0, s_0), (t_1, s_1), (t_2, s_1), (t_3, s_2), (t_4, s_2), (t_5, s_3)\}$$

is also a simulation relation, and hence, s_0 simulates t_0. Thus, the two states s_0 and t_0 are similar. ■

The relationship among various state equivalences is summarized in Theorem 8.6.

Theorem 8.6 [Distinguishing power of state equivalences] $\approx\ \sqsubset\ \simeq^L\ \sqsubset\ \simeq^S\ \sqsubset\ \simeq^B\ \sqsubset\ =$.

Exercise 8.10 {`T4`} [*i*-step similarity] (1) Define i-step trace equivalence and i-step similarity. Show that i-step similarity lies strictly between i-step trace equivalence and i-step bisimilarity on one hand, and between $(i-1)$-step similarity and $(i+1)$-step similarity on the other hand. (2) Give a fixpoint characterization of similarity. ■

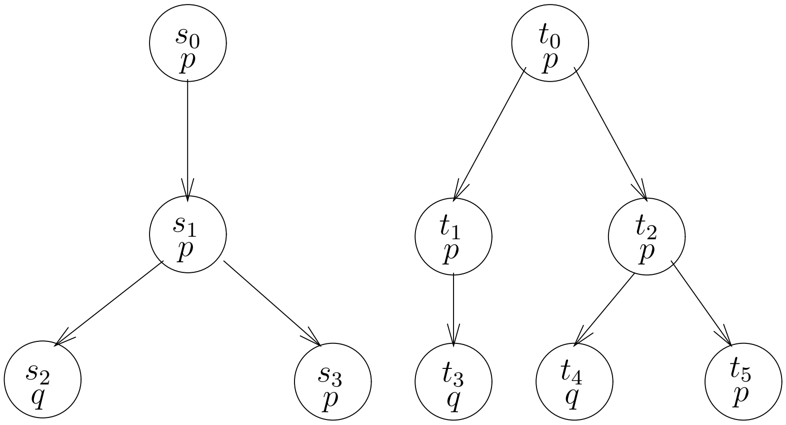


Figure 8.5: Comparing similarity and bisimilarity

Exercise 8.11 {T3} [Backward simulation] Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle)$ be an observation structure. A *backward simulation* $\preceq$ of K is a binary relation on the state space such that for all states s and t of K, if $s \preceq t$ then (1) $\langle\!\langle s \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle$ and (2) if $s' \rightarrow s$, then there is a state t' such that $t' \rightarrow t$ and $s' \preceq t'$. The state t *backward simulates* the state s if there is a backward simulation $\preceq$ such that $s \preceq t$.

Prove that the language-inclusion problem (K_1, K_2) has the answer YES if there is a backward simulation $\preceq$ of $K_1 + K_2$ such that (1) for every state s of K_1 there is a state t of K_2 with $s \preceq t$, and (2) if s is initial and $s \preceq t$, then t is initial.

Prove that similarity and backward similarity are incomparable state equivalences. ■

Exercise 8.12 {T4} [Forward-backward simulation] (1) Prove that simulations and backward simulations are closed under relational composition, and show that the composition of a simulation with a backward simulation may be neither a simulation nor a backward simulation. (2) The composition of a simulation with a backward simulation is called a *forward-backward simulation*, and the composition of a backward simulation with a simulation is a *backward-forward simulation*. In this manner, we can define an infinite family of state equivalences. Prove that all members of this family lie strictly between trace equivalence and bisimilarity in distinguishing power. ■

8.3.2 Universal and existential STL

Universal and existential STL are the fragments of STL whose formulas do not contain quantifier switches. Since quantifier switches correspond to switching sides in the bisimilarity game, universal and existential STL cannot distinguish between similar states.

Let us recall the definition of *universal* STL (∀STL) from Chapter 6. The formulas of ∀STL are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \forall\bigcirc\phi \mid \phi\forall\mathcal{W}\phi.$$

Proposition 6.6 states that STL cannot distinguish between bisimilar states. Now, we establish that ∀STL cannot distinguish between similar states.

Proposition 8.6 [Simulation and Universal STL] *Let s and t be two states of an observation structure K, and let ϕ be a formula of* ∀STL. *Then, if $s \preceq_K^S t$ and $t \models \phi$ then $s \models \phi$.*

Exercise 8.13 {T2} [Simulation and Universal STL] Prove Proposition 8.6. ■

Corollary 8.5 [Similarity and Universal STL] *Similarity is an abstract semantics for* ∀STL.

It follows that it suffices to construct quotients with respect to similarity for model checking of ∀STL requirements. Since similarity is a coarser equivalence than bisimilarity, the quotient with respect to similarity can be smaller than the quotient with respect to bisimilarity.

Recall that bisimilarity is a fully abstract semantics for STL: the equivalence induced by STL coincides with bisimilarity. A similar result holds for ∀STL and similarity: two states of an observation structure K that are not similar can be distinguished by an ∀STL-formula.

Proposition 8.7 [∀STL full abstraction] *Similarity is a fully abstract semantics for* ∀STL.

Exercise 8.14 {T3} [Distinguishing non-similar states with ∀STL] Show that two non-similar states of a finitary observation structure can be distinguished by an ∀STL-formula that uses only the next-time operator $\forall\bigcirc$. Proposition 8.7 follows. ■

Exercise 8.15 {T3} [Existential STL] The formulas of *existential* STL (∃STL) are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists\bigcirc\phi \mid \phi\exists\mathcal{U}\phi.$$

(1) Let s and t be two states of an observation structure K, and let ϕ be a formula of ∃STL. Prove that if $s \preceq_K^S t$ and $s \models \phi$ then $t \models \phi$. It follows that two similar states satisfy the same ∃STL formulas, and similarity is an abstract semantics for ∃STL. (2) Let s and t be two non-similar states of an observation structure K. Prove that there exists an ∃STL-formula that is satisfied by only one of the two states. It follows that the equivalence induced by ∃STL coincides with similarity. ■

State Equivalence	Complexity	Logic
Trace equivalence $\simeq^L$	$O(m \cdot \mathbf{2}^n)$/PSPACE	SAL
Similarity $\simeq^S$	$O(m \cdot n)$	$\forall$STL, $\exists$STL
Bisimilarity $\simeq^B$	$O(m \cdot \log\ n)$	STL

Figure 8.6: Summary of state equivalences

8.4 Computing Similarity

We proceed to study algorithms for deciding whether one structure simulates another. As in case of partition refinement, both enumerative and symbolic algorithms are considered. The complexity of deciding the similarity relation on a finite observation structure is quadratic ($O(m \cdot n)$). Contrast this with $O(m \cdot \log\ n)$ complexity of deciding the bisimilarity relation, which is finer than similarity, and PSPACE complexity of deciding language equivalence, which is coarser than similarity.

The results concerning the three state equivalences, trace equivalence, similarity, and bisimilarity, are summarized in Figure 8.6. The second column shows complexity of deciding equivalence of two states in a structure with n states and m transitions, while the third column list the logic(s) for which the corresponding equivalence is fully abstract.

Let K be an observation structure, and let s be a state of K. Then, the *simulator set* $sim(s)$ of s is the set of states that simulate s.

An instance of the similarity-checking problem consists of a finite observation structure K. The answer to the similarity-checking problem is the set of simulator sets $sim(s)$, for each state s of K.

Remark 8.8 [Simulator sets] Let K be an observation structure. Similar states have identical simulator sets: for two states s and t of K, if $s \simeq^S t$ then $sim(s) = sim(t)$. The simulator set of every state is a block of the partition $\simeq^S$: for every state s, $sim(s)$ is a union of equivalence classes of $\simeq^S$. ■

Once the similarity-checking problem K is solved, then $s \simeq^S t$ iff $s \in sim(t)$ and $t \in sim(s)$. Similarity-checking problem can be used to decide if one observation structure simulates another.

8.4.1 Enumerative Similarity Checking

We develop our enumerative algorithm in three steps.

Algorithm 8.1 [Schematic Similarity]

Input: a finite observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $sim(s)$.

```
foreach $s \in \Sigma$ do $sim(s) := \{t \in \Sigma \mid \langle\!\langle t\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle\}$ od;
while there are three states $t$, $s$, and $u$ such that
        $s \in post(t)$, $u \in sim(t)$, and $post(u) \cap sim(s) = \emptyset$ do
    $sim(t) := Delete(u, sim(t))$
    {I0: assert for all $s, t \in \Sigma$, if $t$ simulates $s$ then $t \in sim(s)$}
    od.
```

Figure 8.7: Enumerative similarity checking

Schematic similarity

We start with the schematic algorithm shown in Figure 8.7. For each state s, the set $sim(s)$ contains states that are candidates for simulating s. Initially, $sim(s)$ contains all states with the observation of s. If $t \rightarrow s$ and $u \in sim(t)$, but there is no $v \in sim(s)$ such that $u \rightarrow v$, then u cannot simulate t and is removed from $sim(t)$, without violating the invariant assertion I0. In this case, we say that $sim(t)$ is *sharpened* with respect to the transition (t, s). It is easy to check that if no transitions allow a sharpening of $sim(t)$ for any state t, then for all s, all states in $sim(s)$ can simulate s.

Theorem 8.7 [Schematic similarity] *Given a finite observation structure K, Algorithm 8.1 correctly solves the similarity-checking problem.*

If the input structure has n states, there can be at most n^2 iterations of the while loop. A naive implementation of the schematic algorithm therefore requires time $O(m^2n^3)$, where $m \geq n$ is the number of transitions of the input structure. We will improve the running time to $O(mn)$.

Refined similarity

The algorithm of Figure 8.8 refines the schema of Algorithm 8.1. The key idea of the refinement is the introduction of a set $prevsim(s)$ for each state s. For each state s, the set $prevsim(s)$ is a superset of $sim(s)$ and contains states that once were considered candidates for simulating s. The crucial invariant I2 of the while loop allows us to sharpen $sim(t)$ with respect to the transition (t, s) by looking only at states in $prevsim(s)$ when checking if a state $u \in sim(t)$ has a successor in $sim(s)$. Moreover, once $v \in prevsim(s) \backslash sim(s)$ is examined once, v is removed from $prevsim(s)$ forever.

Algorithm 8.2 [Refined Similarity]

Input: a finite observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\langle \cdot \rangle\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $sim(s)$.

```
**foreach** $s \in \Sigma$ **do**
   $prevsim(s) := \Sigma$;
   **if** $post(s) = \emptyset$
      **then** $sim(s) := \{t \in \Sigma \mid \langle\langle t \rangle\rangle = \langle\langle s \rangle\rangle\}$
      **else** $sim(s) := \{t \in \Sigma \mid \langle\langle t \rangle\rangle = \langle\langle s \rangle\rangle$ and $post(t) \neq \emptyset\}$
      **fi**
   **od**;
**while** there is a state $s \in \Sigma$ such that $sim(s) \neq prevsim(s)$ **do**
   {I1: **assert** for all $s \in \Sigma$, $sim(s) \subseteq prevsim(s)$}
   {I2: **assert** for all $t, s, u \in \Sigma$, if $t \rightarrow s$ and $u \in sim(t)$,
      then $post(u) \cap prevsim(s) \neq \emptyset$}
   $remove := pre(prevsim(s)) \backslash pre(sim(s))$;
   **foreach** $t \in pre(s)$ **do** $sim(t) := sim(t) \backslash remove$ **od**;
   $prevsim(s) := sim(s)$
   **od**.
```

Figure 8.8: Refined similarity checking

The initial **for** loop of Algorithm 8.2 performs, in addition to the work of the initial **for** loop of Algorithm 8.1, also some of the work of the **while** loop of Algorithm 8.1. For each state s, the set $\mathit{prevsim}(s)$ is initialized to contain all states, and $\mathit{sim}(s)$ is initialized to contain all states with the same observation as that of s, and that have a successor if s does. This initialization establishes the two invariants I1 and I2. In each iteration of the **while** loop, we nondeterministically pick a state s for which $\mathit{sim}(s)$ improves on $\mathit{prevsim}(s)$, and we sharpen $\mathit{sim}(t)$ for all predecessors t of s with respect to the transition (t, s). By I2, all states in $\mathit{sim}(t)$ have successors in $\mathit{prevsim}(s)$, and we can find all states in $\mathit{sim}(t)$ that do not have successors in $\mathit{sim}(s)$ by looking at the predecessor set of $\mathit{prevsim}(s)$. These states are collected in the set *remove* and deleted from $\mathit{sim}(t)$. Once all predecessors of s have been processed in this fashion, we update $\mathit{prevsim}(s)$ to $\mathit{sim}(s)$. If $\mathit{sim}(s) = \mathit{prevsim}(s)$ for all states s, then I2 implies the termination condition of the **while** loop of Algorithm 8.1.

Quadratic similarity checking

The algorithm of Figure 8.9 implements the scheme of Algorithm 8.2 using two data structures. First, instead of recomputing the set *remove* in each iteration of the **while** loop, the algorithm dynamically maintains for each state s a set $\mathit{remove}(s)$ that satisfies the invariant I3. If $\mathit{remove}(s) = \emptyset$ for all states s, then I1 and I3 imply the termination condition of the **while** loop of Algorithm 8.2. Second (not shown in the figure), we maintain a two-dimensional array $\mathit{count}[1..n, 1..n]$ of nonnegative integers such that $\mathit{count}[v, t] = |\mathit{post}(v) \cap \mathit{sim}(t)|$ for all states v and t. The array *count* is initialized in time $O(mn)$. Whenever a state u is removed from $\mathit{sim}(t)$, then the counters $\mathit{count}[v, t]$ are decremented for all predecessors v of u. The cost of these decrements is absorbed in the cost of the innermost **if** statement. With the array *count*, the test $\mathit{post}(v) \cap \mathit{sim}(t) = \emptyset$ of that if statement can be executed in constant time, by checking if $\mathit{count}[v, t] = 0$.

The initialization of $\mathit{sim}(s)$ for all s requires time $O(n \cdot (m + n))$. The initialization of $\mathit{remove}(s)$ for all s requires time $O(mn)$. Given two states s and u, if the test $u \in \mathit{remove}(s)$ is positive in iteration i of the **while** loop, then the test $u \in \mathit{remove}(s)$ is negative in all iterations $j > i$. This is because (1) in all iterations, $u \in \mathit{remove}(s)$ implies that $u \notin \mathit{pre}(\mathit{sim}(s))$, (2) the value of $\mathit{prevsim}(s)$ in all iterations $j > i$ is a subset of the value of $\mathit{sim}(s)$ in iteration i, and (3) invariant I1. It follows that the test $u \in \mathit{sim}(t)$ is executed $\Sigma_s \Sigma_u |\mathit{pre}(s)| = O(mn)$ times. The test $u \in \mathit{sim}(t)$ is positive at most once for every u and t, because after a positive test u is removed from $\mathit{sim}(t)$ and never put back. Therefore the body of the outer **if** statement in the **while** loop contributes time $\Sigma_u \Sigma_t (1 + |\mathit{pre}(u)|) = O(mn)$. This gives a total running time of $O(mn)$.

Algorithm 8.3 [Quadratic Similarity]

Input: a finite observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $sim(s)$.

```
foreach $s \in \Sigma$ do
  {let $prevsim(s) := \Sigma$}
  if $post(s) = \emptyset$
    then $sim(s) := \{t \in \Sigma \mid \langle\!\langle t\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle\}$
    else $sim(s) := \{t \in \Sigma \mid \langle\!\langle t\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle$ and $post(t) \neq \emptyset\}$
    fi;
  $remove(s) := pre(\Sigma) \backslash pre(sim(s))$
  od;
while there is a state $s \in \Sigma$ such that $remove(s) \neq \emptyset$ do
  {I3: assert for all $s \in \Sigma$, $remove(s) = pre(prevsim(s)) \backslash pre(sim(s))$}
  foreach $t \in pre(s)$ do
    foreach $u \in remove(s)$ do
      if $u \in sim(t)$ then
        $sim(t) := Delete(u, Sim(t))$;
        foreach $v \in pre(u)$ do
          if $post(v) \cap sim(t) = \emptyset$ then $remove(t) := Insert(v, remove(t))$ fi
          od
        fi
      od
    od;
    {let $prevsim(s) := sim(s)$}
  $remove(s) := \emptyset$
  od.
```

Figure 8.9: Efficient similarity checking

Theorem 8.8 [Enumerative similarity checking] *Given a finite observation structure with n states and m transitions, Algorithm 8.3 solves the similarity checking problem in time $O(mn)$.*

Corollary 8.6 [Checking similarity of states] *The similarity of two states of a finite observation structure can be decided in time $O(mn)$.*

8.4.2 Symbolic Similarity Checking

Symbolic procedures operate on regions, rather than states. Instead of computing simulator sets for individual states, we compute simulator sets for entire regions. Recall that if two states are similar, then their simulator sets are identical, and the simulator set of every state is a block of the similarity relation $\simeq^S$. This suggests that we should compute simulator sets of equivalence classes of $\simeq^S$, rather than simulator sets of individual states. These two facts lead us to the following definition.

SYMBOLIC SIMULATOR SETS

Given an observation structure K, and a K-partition $\cong$, the simulator function for $\cong$ is the function *Sim* that maps each region σ in $\cong$ to the union $\bigcup_{s\in\sigma} \mathit{sim}(s)$. The *symbolic simulator structure* for K is the pair $(\simeq^S, \mathit{Sim})$ consisting of the similarity partition $\simeq^S$ and the simulator function *Sim* for $\simeq^S$.

Constructing the symbolic simulator structure suffices to answer the similarity checking problem: if $(\simeq^S, \mathit{Sim})$ is the symbolic simulator structure for K, then for a state s of K, $\mathit{sim}(s) = \mathit{Sim}(\sigma)$ where σ is the equivalence class of $\simeq^S$ that contains s.

For an observation a, let $\Sigma_a = \{s \in \Sigma \mid \langle\!\langle s \rangle\!\rangle = a\}$ be the region of states with the observation a. Thus, the collection $\{\Sigma_a \mid a \in A\}$ defines the partition induced by the propositional equivalence $\approx$. We develop our procedure in two steps.

Revised schematic similarity

We start with the schema shown in Figure 8.10, which relaxes the schema of Algorithm 8.1. The initial **for** loops are identical, and establish the two invariants I4 and I5. The invariant I5 asserts that whenever a simulator set $\mathit{sim}(s)$ contains a state u', and u'' simulates u', then $\mathit{sim}(s)$ contains also u''. Assuming I5, if $u \in \mathit{sim}(t)$, $u' \in \mathit{sim}(s)$, and $t \rightarrow u'$, but there is no $u'' \in \mathit{sim}(s)$ such that $u \rightarrow u''$, then u cannot simulate t. This is because in order for u to simulate t, some successor of u would have to simulate u', which is not possible, because by I5 all states that simulate u' are contained in $\mathit{sim}(s)$. We can therefore remove u from $\mathit{sim}(t)$, maintaining both invariants, even if Algorithm 8.1

Algorithm 8.4 [Revised Schematic Similarity]

Input: an observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$.
Output: for each state $s \in \Sigma$, the simulator set $sim(s)$.

```
    foreach $s \in \Sigma$ do $sim(s) := \{t \in \Sigma \mid \langle\!\langle t\rangle\!\rangle = \langle\!\langle s\rangle\!\rangle\}$ od;
    while there are three states $s$, $t$, and $u$ such that
            $post(t) \cap sim(s) \neq \emptyset$, $u \in sim(t)$, and $post(u) \cap sim(s) = \emptyset$ do
      {I4: assert for all $s \in \Sigma$, $s \in sim(s)$}
      {I5: assert for all $s, t, u \in \Sigma$, if $t \preceq^S u$ and $t \in sim(s)$, then $u \in sim(s)$}
      $sim(t) := Delete(u, sim(t))$
      od.
```

Figure 8.10: Revised scheme for similarity checking

would not have allowed us to do so. In this case, we say that $sim(t)$ is *freely sharpened* with respect to the transition (t, u'). If the transition (t, s) allows a sharpening of $sim(t)$, then I4 implies that (t, s) also allows a free sharpening of $sim(t)$. Consequently, if no transitions allow a free sharpening of $sim(t)$ for any state t, then the termination condition of Algorithm 8.1 is satisfied. This implies the partial correctness of the revised scheme.

Theorem 8.9 [Revised schematic similarity] *Given a finite observation structure K, Algorithm 8.4 correctly solves the similarity-checking problem.*

Symbolic algorithm

The symbolic procedure, shown in Figure 8.11, is an instance of the schema of Algorithm 8.4. The symbolic algorithm uses a symbolic representation of regions. The only primitive operations it needs are boolean operations and the *pre*-operation on regions, and emptiness checking of regions. Thus, it is not restricted to finite observation structures, but rather to those structures that support an effective symbolic representation of regions. If the similarity relation $\simeq^S$ of the input structure is finite, then it has only finitely many blocks, and the invariant I7 ensures that Algorithm 8.5 terminates. If $\simeq^S$ is infinite, then the partition $\cong$ needs to be refined infinitely often, and the procedure does not terminate.

In implementing Algorithm 8.5, we can enforce the invariant that for all regions $\sigma \in \cong$, the region $Sim(\sigma)$ is a block of $\cong$, by refining the partition $\cong$ whenever this becomes necessary due to the creation of a new simulator set. Such an implementation maintains a finite partition $\cong$ of the state space Σ together

Algorithm 8.5 [Symbolic Similarity]

Input: an observation structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$.
Output: the symbolic simulator structure $(\simeq^S, \mathit{Sim})$ of K.

```
$\cong\ :=\ \{\Sigma_a \mid a \in A \text{ and } \Sigma_a \neq \emptyset\}$;
**foreach** $\sigma \in \cong$ **do** $\mathit{Sim}(\sigma) := \sigma$ **od**;
**while** there are two regions $\sigma, \tau \in \cong$ such that $\sigma \cap \mathit{pre}(\mathit{Sim}(\tau)) \neq \emptyset$
        and $\mathit{Sim}(\sigma) \backslash \mathit{pre}(\mathit{Sim}(\tau)) \neq \emptyset$ **do**
    {I6: **assert** for all $\sigma \in \cong$ and all $s \in \sigma$, $\mathit{sim}(s) = \mathit{Sim}(\sigma)$}
    {I7: **assert** for all $\sigma \in \cong$, both $\sigma$ and $\mathit{Sim}(\sigma)$ are blocks of $\simeq^S$}
    $\sigma' := \sigma \cap \mathit{pre}(\mathit{Sim}(\tau))$; $\sigma'' := \sigma \backslash \mathit{pre}(\mathit{Sim}(\tau))$;
    $\cong := \mathit{Insert}(\sigma', \mathit{Delete}(\sigma, \cong))$;
    $\mathit{Sim}(\sigma') := \mathit{Sim}(\sigma) \cap \mathit{pre}(\mathit{Sim}(\tau))$;
    **if** $\sigma'' \neq \emptyset$ **then** $\cong := \mathit{Insert}(\sigma'', \cong)$; $\mathit{Sim}(\sigma'') := \mathit{Sim}(\sigma)$ **fi**
    **od**.
```

Figure 8.11: Symbolic similarity checking

with pointers from each region σ in $\cong$ to all regions υ in $\cong$ with $\upsilon \subseteq \mathit{Sim}(\sigma)$, without representing the simulator set $\mathit{Sim}(\sigma)$ explicitly.

Theorem 8.10 [Symbolic similarity checking] *Given an observation structure K with a finite similarity relation, Algorithm 8.5 terminates and computes the symbolic simulator structure for K.*

8.5 Hierarchical Reasoning

Establishing that a reactive module P implements another module Q is computationally hard. In this section, we consider simulation relations as a sufficient condition for establishing the implementation relation between two modules.

8.5.1 Simulation preorder over modules

Every state preorder for observation structures leads to a preorder over modules. Thus, the simulation preorder $\preceq^S$ can be used to compare one module with another. The reactive module Q *simulates* the reactive module P, denoted $P \preceq^S Q$, if (1) every interface variable of Q is an interface variable of P: $\mathsf{intf}X_Q \subseteq \mathsf{intf}X_P$, (2) every external variable of Q is an observable variable of P: $\mathsf{extl}X_Q \subseteq \mathsf{obs}X_P$, (3) for all variables x in $\mathsf{obs}X_Q$ and y in $\mathsf{intf}X_Q$, if $y \prec_Q x$ then $y \prec_P x$, and (4) the observation structure K_Q simulates the observation structure $K_{P'}$ for $P' = \textbf{hide}\ (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q)\ \textbf{in}\ P$.

As in case of the implementation preorder, if $P \preceq^S Q$, then the module P is more constrained than P. The fourth requirement can informally be read as "whatever P does is allowed by Q." Again, we can think of Q as the (more abstract) specification, and P as the (more detailed) implementation. The relationship between simulation preorder and language preorder over states leads to:

Proposition 8.8 [Simulation and implementation] *For two reactive modules P and Q, if $P \preceq^S Q$ then $P \preceq^L Q$.*

Remark 8.9 [Simulation and ∀Stl] Suppose $P \preceq^S Q$, and let ϕ be a formula of ∀Stl. If the answer to the verification problem (Q, ϕ) is Yes, then the answer to the verification problem (P, ϕ) is also Yes. ■

Given two modules P and Q such that $\mathsf{obs}X_Q \subseteq \mathsf{obs}X_P$, a state t of the module P simulates a state s of P, if $s \preceq^S t$ in the observation structure $K_{P'} + K_Q$ for $P' = \textbf{hide}\ (\mathsf{obs}X_P \backslash \mathsf{obs}X_Q)\ \textbf{in}\ P$. If $P \preceq^S Q$ then every reachable state of P is simulated by some reachable state of Q.

Example 8.9 [Nondeterministic versus deterministic scheduling] Recall the modules *Scheduler* and *NonDetScheduler* from Example 8.2. For a state s of *Scheduler* and a state t of *NonDetScheduler*, let $s \preceq t$ if the two states assign the same values to the variables $task_1$, $task_2$, $proc$, new_1, and new_2. Verify that $\preceq$ is a simulation relation: every transition of *Scheduler* is allowed by *NonDetScheduler*. ■

Example 8.10 [Synchronous versus asynchronous mutual exclusion] Let us revisit the two solutions to the mutual exclusion problem, namely, the modules *SyncMutex* and *Pete*. In Example 8.1, we established that *SyncMutex* $\preceq^L$ *Pete*. However, *SyncMutex* $\preceq^S$ *Pete* does not hold. To see this, first note that if a state t of *Pete* simulates a state s of *SyncMutex* then $pc_1[s] = pc_1[t]$ and $pc_2[s] = pc_2[t]$. Consider the following trajectory of *SyncMutex*:

$$\begin{array}{llll} s_0 : (outC, outC) & \rightarrow & s_1 : (outC, reqC) & \rightarrow \\ s_2 : (outC, inC) & \rightarrow & s_0 : (outC, outC) & \rightarrow \\ s_3 : (reqC, reqC) & \rightarrow & s_4 : (inC, reqC) & \end{array}$$

If $s_3 \preceq^S t_3$, then $x_1[t_3] \neq x_2[t_3]$. This is because in *Pete* if both processes are requesting and $x_1 = x_2$, then P_2 enters the critical section first, and hence, cannot match the transition from s_3 to s_4. This implies that if $s_0 \preceq^S t_0$, then $x_1[t_0] = x_2[t_0]$ (since t_0 is required to be a predecessor of t_3). Continuing this line of reasoning, if $s_2 \preceq^S t_2$, then $x_1[t_2] = x_2[t_2]$; if $s_1 \preceq^S t_1$, then $x_1[t_1] = x_2[t_1]$. Now there is no transition between t_0 and t_1, and thus, no such simulation exists. ■

Exercise 8.16 {P2} [Synchronous vs. asynchronous message passing] Recall the modules *SyncMsg* and *AsyncMsg* for synchronous and asynchronous message passing protocols. Does *SyncMsg* $\preceq^S$ *AsyncMsg* hold? ■

8.5.2 Compositional reasoning

Consider the implementation problem of verifying that a module implements its specification. As explained in Section 8.2, this task can be decomposed into subtasks using the compositional and modular properties of the implementation preorder. To verify a particular subtask $P \preceq^L Q$, we can try to prove the stronger goal $P \preceq^S Q$. To establish $P \preceq^S Q$, we can use the symbolic algorithms for similarity checking.

It turns out that the simulation preorder itself is compositional. Thus, if $P \preceq^S Q$ then $P \| R \preceq^S Q \| R$. This helps in decomposing the verification problem for ∀Stl: if $P \preceq^S Q$ then all ∀Stl formulas satisfied by $Q \| R$ are also satisfied by $P \| R$.

Proposition 8.9 [Compositionality of simulation] *The simulation preorder* $\preceq^S$ *on modules is compositional.*

Proof. Consider two reactive modules P and Q such that $P \preceq^S Q$. The cases corresponding to the operations of hiding and renaming are straightforward. We consider only parallel composition. Let R be a reactive module that is compatible with P. For a state s of $P \| R$ and a state t of $Q \| R$, let $s \preceq t$ iff (1) $X_R[s] = X_R[t]$, and (2) $X_Q[t]$ simulates $X_P[s]$.

We first show that $\preceq$ is a simulation relation. Consider $s \preceq t$ and s' be a successor of s in $P \| R$. Then, $X_Q[t]$ simulates $X_P[s]$, and $X_P[s']$ is a successor of $X_P[s]$. Since $P \preceq^S Q$, there exists a state t' of Q such that t' is a successor of $X_Q[t]$ in Q, and t' simulates $X_P[t]$. Let t'' be the state of $Q \| R$ such that $X_Q[t''] = t'$ and $X_R[t''] = X_R[t]$. By definition, $s' \preceq t''$. Since $X_R[s']$ is a successor of $X_R[s]$ in R, it follows that $X_R[t'']$ is a successor of $X_R[t]$ in R. Thus, t'' is a successor of t in $Q \| R$.

Along the same lines, we can establish that for every initial state s of $P \| R$, there is an initial state t of $Q \| R$ such that $s \preceq t$. ■

Exercise 8.17 {T5} [Assume-Guarantee for Simulation] Does the assume-guarantee theorem for implementation preorder (Theorem 8.4) hold for the simulation preorder $\preceq^S$? ■

8.5.3 Refinement mappings

Refinement maps, or homomorphisms, are special types of simulation relations.

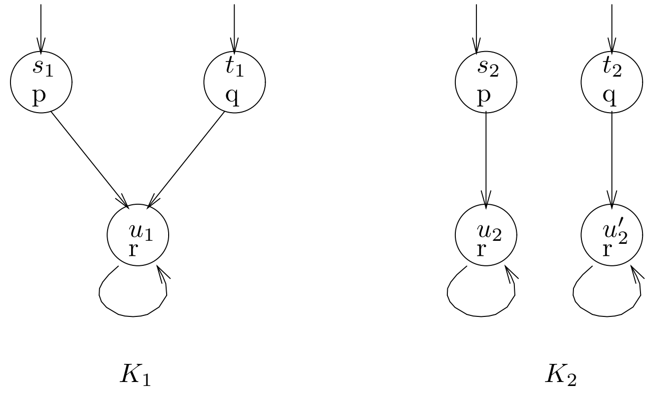


Figure 8.12: Refinement maps versus simulation relations

Refinement maps

Let $K_1 = (\Sigma_1, \sigma_1^I, \rightarrow_1, A, \langle\!\langle\cdot\rangle\!\rangle_1)$ and $K_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2, A, \langle\!\langle\cdot\rangle\!\rangle_2)$ be two observation structures. A *refinement mapping* hom from K_1 to K_2 is a function from the reachable region σ_1^R of K_1 to Σ_2 such that (1) for all $s \in \sigma_1^R$, $\langle\!\langle \mathrm{hom}(s)\rangle\!\rangle_2 = \langle\!\langle s\rangle\!\rangle_1$, (2) for every reachable transition $s \rightarrow_1 t$ of K_1, $\mathrm{hom}(s) \rightarrow_2 \mathrm{hom}(t)$, and (3) for all $s \in \sigma_1^I$, $\mathrm{hom}(s) \in \sigma_2^I$.

If hom is a refinement map from K_1 to K_2 then the set $\{(s, \mathrm{hom}(s)) \mid s \in \sigma_1^R\}$ is simulation relation over the union $K_1 + K_2$.

Proposition 8.10 [Refinement maps and simulations] *If there exists a refinement mapping from the observation structure K_1 to K_2 then $K_1 \preceq^S K_2$.*

For two observation structures K_1 and K_2, if there exists a refinement map from K_1 to K_2, then $K_1 \preceq^S K_2$, and hence, $K_1 \preceq^L K_2$. Thus, we can establish implementation relation between two modules by supplying a refinement map from the states of the detailed module to the states of the abstract module. Existence of simulation relation between two modules, however, does not guarantee existence of refinement maps.

Example 8.11 [Nondeterministic versus deterministic scheduling] Recall the modules *Scheduler* and *NonDetScheduler* from Example 8.9. Given a state s of *Scheduler*, let hom(s) be the state of *NonDetScheduler* obtained by simply discarding the value of the variable *prior*. In this case, this projection map is a refinement map, and establishes that *Scheduler* $\preceq^S$ *NonDetScheduler*. ■

Example 8.12 [Refinement map vs. simulation] Consider the two observation structures K_1 and K_2 shown in Figure 8.12. The relation $\{(s_1, s_2), (t_1, t_2), (u_1, u_2), (u_1, u_2')\}$

is a simulation relation, and thus, $K_1 \preceq^S K_2$. However, there is no refinement map from K_1 to K_2. Observe that there is a refinement map from K_2 to K_1, and $K_2 \preceq^S K_1$. ■

Exercise 8.18 {`T4`} [Completeness of refinement mappings] Let P and Q be two reactive modules such that $P \preceq^L Q$. Prove that there is a monitor R for P such that there is a refinement mapping from $K_{P\|R}$ to K_Q. ■

Exercise 8.19 {`P3`} [Verifying refinements] Write an algorithm that given two observation structures K_1 and K_2 and a mapping hom from the states of K_1 to the states of K_2 verifies whether or not hom is a refinement map. ■

8.6 Stutter-closed Implementation

In Chapter 6, we saw how each state equivalence leads to its stutter-closed version obtained by adding extra transitions that obliterate the distinction due to the number of rounds for which an observation stays unchanged. In the same manner, every state preorder leads to a stutter-closed version: two states of an observation structure K are related according to the stutter-closed version of a preorder $\preceq$, if those two states are related according to $\preceq$ in the stutter-closure of K.

Stutter closure of state preorders

Let $\preceq$ be a state preorder, and let K be an observation structure. For two states s and t of K, $s \underline{\preceq}_K t$, for the stutter closure $\underline{\preceq}$ of $\preceq$, if $s \preceq_{K^S} t$. The induced state preorder $\underline{\preceq}$ is called the *stutter closure* of $\preceq$.

Remark 8.10 [Alternative characterization of stutter closure of trace preorder] Let A be a set of symbols. Let $\overline{a}_{0\ldots m}$ be a word over A. A *stutter-extension* of $\overline{a}$ is a word that can be obtained from $\overline{a}$ by repeating each symbol of $\overline{a}$ finitely many times: a word $\overline{b}$ over A is a *stutter-extension* of $\overline{a}$ iff there exist positive integers $i_0, i_1, \ldots i_m$ such that $\overline{b} = a_0^{i_0} a_1^{i_1} \ldots a_m^{i_m}$. For two states s and t of an observation structure K, $s \underline{\preceq}^L t$ holds if for every source-s trace $\overline{a}$ there exists a source-t trace $\overline{b}$ and a word $\overline{c}$ such that $\overline{c}$ is a stutter-extension of $\overline{a}$ and is also a stutter-extension of $\overline{b}$. ■

Stutter closure of trace equivalence is the weakest equivalence we have considered so far: it is less distinguishing than trace equivalence, and it is less distinguishing than weak bisimilarity.

The stutter closure of the trace preorder over observation structures leads to a way of comparing two modules, called *weak implementation*, denoted $\underline{\preceq}^L$.

Example 8.13 [Equivalence of synchronous vs.asynchronous message passing] Recall the modules *SyncMsg* and *AsyncMsg* for synchronous and asynchronous message passing protocols. The two modules are not trace equivalent, however, they are equivalent according to the equivalence induced by weak implementation. ■

The weak-implementation relation plays an important role in reasoning about asynchronous systems.

Remark 8.11 [Stutter-extensions and asynchronous modules] If P is an asynchronous module, then its language L_P is closed under stutter-extension: if $\overline{a}$ is a trace of P then every stutter-extension of $\overline{a}$ is also a trace of P. ■

The weak implementation relation is compositional as long as we use only asynchronous modules.

Theorem 8.11 [Compositionality of weak implementation] *For two modules P and Q, if $P \underline{\preceq}^L Q$ then (1) for variable x of P,* **hide** x **in** $P \underline{\preceq}^L$ **hide** x **in** Q*; (2) for every variable renaming ρ, $P[\rho] \underline{\preceq}^L Q\,[\rho]$. For asynchronous modules P, Q, and R, if $P \underline{\preceq}^L Q$ and R is compatible with P, then R is compatible with Q and $P \| R \underline{\preceq}^L Q \| R$.*

Exercise 8.20 {T3} [Compositionality of weak implementation] Prove Theorem 8.11. Show that the preorder $\underline{\preceq}^L$ is not compositional with respect to parallel composition with all modules; that is, find modules P, Q, and R such that $P \underline{\preceq}^L Q$, but $P \| R \underline{\preceq}^L Q \| R$ does not hold. ■

The assume-guarantee theorem for the implementation relation holds for the weak-implementation relation provided we consider only asynchronous modules.

Theorem 8.12 [Assume-guarantuee reasoning for weak implementation] *Let P_1 and P_2 be two compatible asynchronous reactive modules, and let Q_1 and Q_2 be two compatible asynchronous reactive modules. If $P_1 \| Q_2 \underline{\preceq}^L Q_1$, $Q_1 \| P_2 \underline{\preceq}^L Q_2$, and every external variable of $Q_1 \| Q_2$ is an observable variable of $P_1 \| P_2$, then $P_1 \| P_2 \underline{\preceq}^L Q_1 \| Q_2$.*

Exercise 8.21 {T3} [Assume-guarantuee for weak implementation] Prove Theorem 8.12. ■

Weak similarity

The stutter closure $\underline{\preceq}^S$ of simulation preorder is called *weak simulation*, and the stutter closure $\cong^S$ of similarity is called *weak similarity.*

$$\begin{array}{ccc}
\text{Weak Trace Equivalence } \cong^L & \sqsubset & \text{Trace Equivalence } \simeq^L \\
\sqcap & & \sqcap \\
\text{Weak Similarity } \cong^S & \sqsubset & \text{Similarity } \simeq^S \\
\sqcap & & \sqcap \\
\text{Weak Bisimilarity } \cong^B & \sqsubset & \text{Bisimilarity } \simeq^B
\end{array}$$

Figure 8.13: Relationship among state equivalences

Remark 8.12 [Weak simulation] Weak simulation (and weak similarity) can be defined directly without considering the stutter closure operation on structures explicitly. Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. A *weak-simulation* $\underline{\preceq} \subseteq \Sigma^2$ of K is a binary relation on the state space such that for all states s and t of K, if $s \,\underline{\preceq}\, t$ then (1) $\langle\!\langle s\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle$ and (2) if $s \rightarrow s'$, then there is a state t' such that $s' \,\underline{\preceq}\, t'$ and there exists a trajectory $\overline{t}_{0\ldots m}$ of K with $t_0 = t$, $t_m = t'$, and $\langle\!\langle t_i\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle$ for $0 \leq i < m$. The state t *weakly-simulates* the state s if there is a weak simulation $\underline{\preceq}$ such that $s \,\underline{\preceq}\, t$. Now, $s \,\underline{\preceq}^S\, t$ if t weakly-simulates s. ■

We know that similarity is more distinguishing than trace equivalence, but less distinguishing than bisimilarity. Analogously, weak similarity is more distinguishing than weak trace equivalence, but less distinguishing than weak bisimilarity.

Exercise 8.22 {T3} [Weak similarity vs. trace equivalence] Establish that weak similarity and trace equivalence are incomparable. ■

Exercise 8.23 {T4} [Weak similarity and waiting-for fragment of ∀Stl] Let $\forall\text{Stl}^{\mathcal{W}}$ be the fragment of ∀Stl that contains no next-time operators, that is, its formulas are generated by the grammar

$$\phi \ ::= \ p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \forall\mathcal{W}\phi.$$

Establish that weak similarity is a fully abstract semantics for this fragment. ■

Contents

Chapter 9

Fair Modules

9.1 Safety versus Liveness

So far, we have considered safety requirements of reactive modules. Intuitively, a *safety requirement* is a requirement that can be violated by a finite trace. For example, the mutual-exclusion and equal-opportunity requirements can be violated by finite traces. More generally, all SAL requirements can be violated by finite traces. Why would we care about requirements that cannot be violated by finite traces? Such requirements would not be violated within the next year, nor within our lifetime, nor within the lifetime of the universe. The answer is convenience in system and requirement description.

Let us assume, for the sake of argument, that there is no truly nondeterministic physical process. Even with this assumption, the nondeterministic update command

```
**update** $x$
   $[\!] \; \mathit{true} \to x' := 0$
   $[\!] \; \mathit{true} \to x' := 1$
```

is useful for describing systems that assign 0s and 1s to x, because the nondeterminism frees us from the responsibility of being specific when x is 0, and when x is 1. The actual law that determines the value of x in each round may be arbitrarily complex, and yet irrelevant for our purposes of proving certain system requirements. Similarly, it is often convenient to assert that an event will happen, without giving detailed information on when it will happen. For example, we may want to assert that x never stays 0, and it never stays 1, without being specific on how many rounds can expire between consecutive changes in the value of x. No finite trace can violate this assertion, yet an infinite trace can violate it by having one the value of x remain unchanged after some round. We use the notion of *fair update choices* for enforcing the eventual execution of particular guarded assignments of a nondeterministic update command:

```
update $x$ weaklyfair $a, b$
    $\| \; \mathit{true} \xrightarrow{a} x' := 0$
    $\| \; \mathit{true} \xrightarrow{b} x' := 1$
```

The declaration **weakly fair** a ensures that the update choice a, which sets x to 0, cannot be neglected forever, and the declaration **weakly fair** b ensures the same for the update choice b, which sets x to 1.

Requirements that can be violated by infinite traces only are called *liveness requirements*. Trajectories and traces of reactive modules are finite sequences of states and observations, respectively. In order to specify whether a module satisfies or violates a liveness requirement, we need to define infinite trajectories and infinite traces, called ω*-trajectories* and ω*-traces*.

9.1.1 ω-Words and ω-Languages

Let A be a set of symbols. An ω*-word* $\underline{a} = a_0 a_1 a_2 \cdots$ *over the alphabet* A is an infinite sequence of symbols a_i from A. We write A^ω for the set of ω-words over A. An ω*-language* $\mathcal{L}$ *over the alphabet* A is a set of ω-words over A; that is, $\mathcal{L} \subseteq A^\omega$.

For a word $\overline{b} \in A^*$ and an ω-word $\underline{a}$, by $\overline{b} \cdot \underline{a}$ we denote the ω-word that results from *concatenating* the two. The word $\overline{a}$ is a *prefix* of the ω-word $\underline{b}$ if there exists an ω-word $\underline{c}$ such that $\underline{b} = \overline{a} \cdot \underline{c}$, and $\underline{a}$ is a *suffix* of $\underline{b}$ if there exists a word $\overline{c}$ such that $\underline{b} = \overline{c} \cdot \underline{a}$. The set of prefixes of the ω-word $\underline{a}$ is denoted by $\mathit{pref}(\underline{a})$. For an ω-language $\mathcal{L}$, $\mathit{pref}(\mathcal{L})$ is the language $(\cup \underline{a} \in \mathcal{L}.\, \mathit{pref}(\underline{a}))$. For a language $L \subseteq A^*$, the ω-language L^ω consists of ω-words $\underline{a}$ such that $\underline{a} = \overline{a}_0 \cdot \overline{a}_1 \cdot \overline{a}_2 \cdots$ with $\overline{a}_i \in L$ for all $i \geq 0$. In other words, the ω-words in L^ω are obtained by concatenating infinitely many words in L. Consequently, we freely use the superscript ω in regular expressions.

An ω-word $\underline{a}$ is *periodic* if there is a word $\overline{b}$ such that $\underline{a} = \overline{b}^\omega$, that is, $\underline{a}$ is obtained by concatenating infinitely many copies of the finite word $\overline{b}$. An ω-word $\underline{a}$ is *eventually periodic* if it has a periodic suffix. An eventually periodic word is of the form $\overline{a} \cdot \overline{b}^\omega$ for two words $\overline{a}$ and $\overline{b}$.

The ω-language $\mathcal{L}$ is *suffix-closed* if for every ω-word $\underline{a}$ in $\mathcal{L}$, all suffixes of $\underline{a}$ are also in $\mathcal{L}$. The ω-language $\mathcal{L}$ is *fusion-closed* if for all symbols a, if $\overline{b} \cdot a \cdot \underline{c}$ and $\overline{b}' \cdot a \cdot \underline{c}'$ are in $\mathcal{L}$, then so is $\overline{b} \cdot a \cdot \underline{c}'$. An ω-word $\underline{a}$ is called a *limit* of the language L if $\mathit{pref}(\underline{a}) \subseteq L$. An ω-word $\underline{a}$ is a *limit* of the ω-language $\mathcal{L}$ if it is a limit of $\mathit{pref}(\mathcal{L})$. In other words, $\underline{a}$ is a limit of $\mathcal{L}$ if every finite prefix of $\underline{a}$ can be extended to an ω-word in $\mathcal{L}$. The ω-language $\mathcal{L}$ is *limit-closed* if it contains all its limits: for all ω-words $\underline{a} \in A^\omega$, if $\mathit{pref}(\underline{a}) \subseteq \mathit{pref}(\mathcal{L})$, then $\underline{a} \in \mathcal{L}$; that is, if every prefix of $\underline{a}$ can be extended to an ω-word in $\mathcal{L}$, then $\underline{a}$ itself is also in $\mathcal{L}$.

Example 9.1 [ω-languages] Let $A = \{a, b\}$. Consider the ω-language $\mathcal{L}_1$ consisting of all ω-words $\underline{a} \in (a + b)^\omega$ that contain infinitely many a symbols: $\mathcal{L}_1 = (b^* a)^\omega$. Then, $\mathit{pref}(\mathcal{L}_1) = (a + b)^*$. The language $\mathcal{L}_1$ is suffix-closed and fusion-closed. However, $\mathcal{L}_1$ is not limit-closed: the ω-word b^ω is a limit of $\mathcal{L}_1$, but is not in $\mathcal{L}_1$.

Consider the ω-language $\mathcal{L}_2$ consisting of all ω-words $\underline{a}$ such that $a_i = a$ for all odd positions i: $\mathcal{L}_2 = ((a + b)a)^\omega$. The language $\mathcal{L}_2$ is limit-closed, but neither suffix-closed nor fusion-closed.

The ω-language $\mathcal{L}_2$ contains the periodic ω-word $(ba)^\omega$. It also contains the eventually periodic ω-word $bababa^\omega$. Not all ω-words in $\mathcal{L}_2$ are eventually periodic, for instance, the ω-word $baba^3ba^5ba^7ba^9\ldots$.

Consider the ω-language $\mathcal{L}_3$ that is the complement of the language $\mathcal{L}_1$ (with respect to A^ω). The ω-word $\underline{a}$ belongs to $\mathcal{L}_3$ iff it contains only finitely many a symbols. Thus, $\mathcal{L}_3 = (a + b)^* b^\omega$ and $\mathit{pref}(\mathcal{L}_3) = (a + b)^*$. The language $\mathcal{L}_3$ is suffix-closed, fusion-closed, but not limit-closed. ■

Remark 9.1 [Limit-closed ω-languages] A limit-closed ω-language $\mathcal{L}$ is completely characterized by its prefix language $\mathit{pref}(\mathcal{L})$: $\mathcal{L} = \{\underline{a} \mid \mathit{pref}(\underline{a}) \subseteq \mathit{pref}(\mathcal{L})\}$. ■

9.1.2 The safety-liveness distinction

Consider an ω-language $\mathcal{L}$. If for every ω-word $\underline{a}$ it can be checked whether $\underline{a}$ belongs to $\mathcal{L}$ by looking only at the finite prefixes of $\underline{a}$, then the ω-language $\mathcal{L}$ is called *safe*. If for every ω-word $\underline{a}$ it cannot be checked whether $\underline{a}$ belongs to $\mathcal{L}$ by looking at any finite prefix of $\underline{a}$, then the ω-language $\mathcal{L}$ is called *live*.

SAFETY, LIVENESS, AND MACHINE CLOSURE

Let A be a set of symbols, and let $\mathcal{L}$ be an ω-language over the alphabet A. The ω-language $\mathcal{L}$ is *safe* if $\mathcal{L}$ is limit-closed. The ω-language $\mathcal{L}$ is *live* if $\mathit{pref}(\mathcal{L}) = A^*$. Given a safe ω-language $\mathcal{L}_S$ and a live ω-language $\mathcal{L}_L$ over A, the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is *machine-closed* if $\mathit{pref}(\mathcal{L}_S \cap \mathcal{L}_L) = \mathit{pref}(\mathcal{L}_S)$. If the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is machine-closed and the ω-language $\mathcal{L}$ equals $\mathcal{L}_S \cap \mathcal{L}_L$, then the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is said to be a *machine-closed specification* of $\mathcal{L}$.

Remark 9.2 [Safe and live language] The ω-language A^ω is the only ω-language over the alphabet A that is both safe and live. ■

If $\mathcal{L}$ is a safe ω-language, and $\underline{a}$ is an ω-word, then $\underline{a} \in \mathcal{L}$ iff all finite prefixes of $\underline{a}$ can be extended to ω-words in $\mathcal{L}$. If $\mathcal{L}$ is a live ω-language, and $\overline{a}$ is a word, then $\overline{a}$ can be extended to an ω-word in $\mathcal{L}$. If $(\mathcal{L}_S, \mathcal{L}_L)$ is a machine-closed pair of ω-languages, then all finite words that can be extended to ω-words in $\mathcal{L}_S$ can also be extended to ω-words in $\mathcal{L}_S \cap \mathcal{L}_L$.

Example 9.2 [Safety, liveness, and machine closure] Let $A = \{a, b\}$. The ω-language $\mathcal{L}_1 = (b^* a)^\omega$ is not safe, but is live. On the other hand, the ω-language $\mathcal{L}_2 = ((a+b)a)^\omega$ is safe, but not live. The pair $(\mathcal{L}_2, \mathcal{L}_1)$ is machine-closed, since $\mathcal{L}_1 \cap \mathcal{L}_2 = \mathcal{L}_2$. The ω-language $\mathcal{L}_3 = A^* b^\omega$ is live, but not safe. The pair $(\mathcal{L}_2, \mathcal{L}_3)$ is not machine-closed, since $\mathcal{L}_2 \cap \mathcal{L}_3$ is the empty language.

Consider the language $\mathcal{L}_4 = a^\omega + b^\omega$. The language $\mathcal{L}_4$ is safe. The pair $(\mathcal{L}_4, \mathcal{L}_1)$ is not machine-closed: no prefix of b^ω can be extended to an ω-word in $\mathcal{L}_4 \cap \mathcal{L}_1$. ■

As we will see later, the desired set of infinite trajectories of a module will be specified by a machine-closed pair of ω-languages. The safety component is specified by the transition relation, and the liveness component is specified by fairness assumptions about update choices. Machine-closure ensures that the fairness assumptions constrain only what is allowed in the limit, and can be ignored while verifying safety properties of the system. This aspect of machine-closure is captured by the following proposition.

Proposition 9.1 [Safety verification] *Let $(\mathcal{L}_S, \mathcal{L}_L)$ be a machine-closed specification of the ω-language $\mathcal{L}$, and let $\mathcal{L}'$ be a safe language. Then, $\mathcal{L} \subseteq \mathcal{L}'$ iff $\mathcal{L}_S \subseteq \mathcal{L}'$.*

Exercise 9.1 {T2} [Safety verification] Prove Proposition 9.1. ■

Requiring machine-closure is not restrictive since every ω-language can be specified by a machine-closed pair:

Theorem 9.1 [Safety-liveness decomposition] *Let A be a set of symbols. Every ω-language $\mathcal{L}$ over the alphabet A can be specified by a machine-closed pair $(\mathcal{L}_S, \mathcal{L}_L)$ consisting of a safe ω-language $\mathcal{L}_S$ and a live ω-language $\mathcal{L}_L$ over A.*

Proof. Let $\mathcal{L}_S$ be the limit closure of $\mathcal{L}$; that is, $\mathcal{L}_S$ contains all the limits of $\mathcal{L}$. Thus, $\mathcal{L}_S$ is completely characterized by *pref*$(\mathcal{L})$, and is safe. Let $\mathcal{L}_L$ be $(A^\omega \backslash \mathcal{L}_S) \cup \mathcal{L}$; that is, $\mathcal{L}_L$ contains all ω-words, except the limits of $\mathcal{L}$ not in $\mathcal{L}$. Every word is either a prefix of $\mathcal{L}$, or not a prefix of $\mathcal{L}_S$, and hence, a prefix of $(A^\omega \backslash \mathcal{L}_S)$. It follows that *pref*$(\mathcal{L}_L) = A^*$, and $\mathcal{L}_L$ is live. Since $\mathcal{L} \subseteq \mathcal{L}_S$, $\mathcal{L}_S \cap \mathcal{L}_L = \mathcal{L}$, and $(\mathcal{L}_s, \mathcal{L}_L)$ is machine-closed. ■

9.1.3 The safety-progress hierarchy

To understand the structure of ω-languages, we consider various ways of building ω-languages from languages over finite words.

Safety languages

Given a language $L \subseteq A^*$, the corresponding safety language consists of all ω-words whose all prefixes belong to L.

Safety

For a language $L \subseteq A^*$ over an alphabet A, *safe*(L) is the ω-language $\{\underline{a} \mid \forall i \geq 0.\, \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *safety language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{safe}(L)$.

Remark 9.3 [Safety] If $\mathcal{L} = \mathit{safe}(L)$ then $L = \mathit{pref}(\mathcal{L})$. This implies that both definitions of safety coincide: $\mathcal{L}$ is limit-closed iff $\mathcal{L} = \mathit{safe}(L)$ for some language L. ■

While specifying requirements of a reactive module, the alphabet A corresponds to the set of observations. A safe language *safe*(L), then, can be used to specify that "nothing bad ever happens" as the specification requires every possible finite trace to be in the set L. A classical safety property is the mutual exclusion property of resource allocation algorithms that requires that the same resource is not allocated to two different processes simultaneously.

Example 9.3 [Safe languages] Let $A = \{a, b\}$. The ω-language $\mathcal{L}_2 = (Aa)^\omega$ is safe, and equals $\mathit{safe}((Aa)^* + (Aa)^* A)$. The empty language is safe: $\emptyset = \mathit{safe}(\emptyset)$. The universal language A^ω is safe: $A^\omega = \mathit{safe}(A^*)$. The ω-language consisting of ω-words $\underline{a}$ such that for all $i \geq 0$, if i is a prime number, then $a_i = a$, is safe. The ω-language $\mathcal{L}_1 = (b^* a)^\omega$ is not safe. The ω-language consisting of the single ω-word a^ω is safe; however, its complement $A^* b A^\omega$ consisting of ω-words with at least one b symbol, is not safe. ■

The next proposition asserts that union of two safe languages is safe, and intersection of two safe languages is also safe. The complement of a safe language need not be safe, as illustrated in Example 9.3.

Proposition 9.2 [Closure for safety languages] *Safety languages are closed under union and intersection, but not under complementation.*

Proof. Consider $\mathcal{L}_1 = \mathit{safe}(L_1)$ and $\mathcal{L}_2 = \mathit{safe}(L_2)$. An ω-word $\underline{a}$ is in $\mathcal{L}_1 \cap \mathcal{L}_2$, iff $\underline{a} \in \mathcal{L}_1$ and $\underline{a} \in \mathcal{L}_2$, iff for all $i \geq 0$, $\overline{a}_{0...i} \in L_1$ and $\overline{a}_{0...i} \in L_2$, iff for all $i \geq 0$, $\overline{a}_{0...i} \in L_1 \cap L_2$, iff $\underline{a} \in \mathit{safe}(L_1 \cap L_2)$. This establishes that $\mathcal{L}_1 \cap \mathcal{L}_2 = \mathit{safe}(L_1 \cap L_2)$, and hence, $\mathcal{L}_1 \cap \mathcal{L}_2$ is safe.

To establish closure under union, let L'_1 be the language $\{\overline{a} \mid \mathit{pref}(\overline{a}) \subseteq L_1\}$ consisting of words all of whose prefixes are in L_1. Similarly, let $L'_2 = \{\overline{a} \mid \mathit{pref}(\overline{a}) \subseteq L_2\}$. We prove that $\mathcal{L}_1 \cup \mathcal{L}_2 = \mathit{safe}(L'_1 \cup L'_2)$.

Consider an ω-word $\underline{a} \in \mathcal{L}_1 \cup \mathcal{L}_2$. Without loss of generality, suppose $\underline{a} \in \mathcal{L}_1$. Then, for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L_1$. Hence, for all $i \geq 0$, for all $0 \leq j \leq i$, $\overline{a}_{0\ldots j} \in L_1$. Hence, for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L'_1$. Hence, $\underline{a} \in \mathit{safe}(L'_1 \cup L'_2)$.

Consider an ω-word $\underline{a} \in \mathit{safe}(L'_1 \cup L'_2)$. Then, for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L'_1 \cup L'_2$. Without loss of generality, for infinitely many positions i, $\overline{a}_{0\ldots i} \in L'_1$. This implies for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L_1$ (for, if $\overline{a}_{0\ldots j} \notin L_1$ for some j, then for all $i \geq j$, $\overline{a}_{0\ldots i} \notin L'_1$). Hence, $\underline{a} \in \mathit{safe}(L_1)$. Hence, $\underline{a} \in \mathcal{L}_1 \cup \mathcal{L}_2$. ■

Exercise 9.2 {`T2`} [*safe* does not distribute over union] Show that $\mathit{safe}(L_1) \cup \mathit{safe}(L_2)$ is not necessarily equal to $\mathit{safe}(L_1 \cup L_2)$. ■

Guarantee languages

Given a language $L \subseteq A^*$, the corresponding guarantee language consists of all ω-words whose *some* prefix belongs to L.

> GUARANTEE
>
> For a language $L \subseteq A^*$ over an alphabet A, $\mathit{guar}(L)$ is the ω-language $\{\underline{a} \mid \exists i \geq 0.\, \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *guarantee language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{guar}(L)$.

Remark 9.4 [Guarantee] The ω-language $\mathcal{L} \subseteq A^\omega$ is a guarantee language iff there is a language $L \subseteq A^*$ such that $\mathcal{L} = L \cdot A^\omega$. ■

While specifying requirements of a reactive module, a guarantee language $\mathit{guar}(L)$ can be used to specify that "something good eventually happens" as the specification requires the module to produce a trace in L after executing for finitely many steps. A classical guarantee property is the termination requirement that a program eventually produces an output.

Example 9.4 [Guarantee languages] Let $A = \{a, b\}$. The empty language is a guarantee language: $\emptyset = \mathit{guar}(\emptyset)$. The universal language A^ω is also a guarantee language: $A^\omega = \mathit{guar}(A^+)$. The ω-language A^*bA^ω consisting of ω-words with at least one b symbol, is a guarantee language: $A^*bA^\omega = \mathit{guar}(A^*b)$. ■

The safety and guarantee languages are closely related, namely, they are duals of each other: the complement of a safe language is a guarantee language, and the complement of a guarantee language is a safe language.

Proposition 9.3 [Duality of safety and guarantee] *The ω-language $\mathcal{L}$ is a safety language iff the complementary language $A^\omega \backslash \mathcal{L}$ is a guarantee language.*

Exercise 9.3 {T2} [Duality of safety and guarantee] Prove that $\mathcal{L} = \mathit{safe}(L)$ iff $A^\omega \backslash \mathcal{L} = \mathit{guar}(A^+ \backslash L)$. Proposition 9.3 follows. ■

Exercise 9.4 {T2} [Safe and guarantee languages] Characterize the class of ω-languages that are both safety and guarantee languages. ■

Since safe languages are closed under union and intersection, it follows that so are guarantee languages.

Proposition 9.4 [Closure for guarantee languages] *Guarantee languages are closed under union and intersection, but not under complementation.*

Remark 9.5 [Closure for guarantee languages] The closure properties of guarantee languages can be established directly also:

$$\mathit{guar}(L_1) \cup \mathit{guar}(L_2) \ = \ \mathit{guar}(L_1 \cup L_2),$$

and

$$\mathit{guar}(L_1) \cap \mathit{guar}(L_2) \ = \ \mathit{guar}((L_1 \cdot A^*) \cap (L_2 \cdot A^*)).$$

■

Exercise 9.5 {T2} [Obligation languages] Obligation languages are obtained by boolean combinations of safety or guarantee languages. In other words, the set of obligation languages is the least set that contains all safety languages, and is closed under union, intersection, and complementation. Every obligation language can be expressed in a normal form: $\bigcup 0 \leq i \leq k. \mathit{safe}(L_i) \cap \mathit{guar}(L'_i)$. For example, for $A = \{a, b\}$, the ω-language a^*ba^ω consisting of ω-words with precisely one b symbol, is an obligation language: $\mathit{safe}(a^*ba^* + a^*) \cap \mathit{guar}(a^*b)$. Show that the obligation language a^*ba^ω is neither a safety language nor a guarantee language. ■

Response languages

Given a language $L \subseteq A^*$, the corresponding response language consists of all ω-words whose infinitely many prefixes belong to L.

RESPONSE

For a language $L \subseteq A^*$ over an alphabet A, $\mathit{recur}(L)$ is the ω-language $\{\underline{a} \mid \forall j \geq 0. \exists i \geq j. \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *response language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{recur}(L)$.

While specifying requirements of a reactive module, a response language $\mathit{recur}(L)$ is used to specify that “something good happens repeatedly” as the specification requires infinitely many traces to be in the set L. A classical response property is the progress requirement for resource allocation algorithms: if some process is requesting a resource then some process is eventually granted the resource.

Example 9.5 [Response languages] Let $A = \{a, b\}$, and let $L = (a^*b)^*$. Then, $recur(L) = (a^*b)^\omega$ is the corresponding response language, and consists of all ω-words with infinitely many b symbols. Observe that $recur(L)$ is neither a safety language nor a guarantee language. ■

The next proposition asserts that union of two response languages is a response language, and intersection of two response languages is also a response language. However, the response languages are not closed under complementation.

Proposition 9.5 [Closure for response languages] *Response languages are closed under union and intersection, but not under complementation.*

Proof. Consider $\mathcal{L}_1 = recur(L_1)$ and $\mathcal{L}_2 = recur(L_2)$. An ω-word $\underline{a}$ is in $\mathcal{L}_1 \cup \mathcal{L}_2$, iff $\underline{a} \in \mathcal{L}_1$ or $\underline{a} \in \mathcal{L}_2$, iff for infinitely many i, $\overline{a}_{0\ldots i} \in L_1$ or for infinitely many i, $\overline{a}_{0\ldots i} \in L_2$, iff for infinitely many i, $\overline{a}_{0\ldots i} \in L_1 \cup L_2$, iff $\underline{a} \in recur(L_1 \cup L_2)$. This establishes that $\mathcal{L}_1 \cup \mathcal{L}_2 = recur(L_1 \cup L_2)$, and hence, $\mathcal{L}_1 \cup \mathcal{L}_2$ is a response language.

For closure under intersection, consider the language L_{12} that contains a word $\overline{a}_{0\ldots m}$ iff (1) $\overline{a}_{0\ldots m} \in L_2$, and (2) there exits $i < m$ such that $\overline{a}_{0\ldots i} \in L_1$ and $\overline{a}_{0\ldots k} \notin L_2$ for $i < k < m$. We prove that $\mathcal{L}_1 \cap \mathcal{L}_2 = recur(L_{12})$.

Consider an ω-word $\underline{a} \in recur(L_{12})$. There exists an infinite increasing sequence of integers $i_0, i_1 \ldots$ such that for all $j \geq 0$, $\overline{a}_{0\ldots i_j}$ is in L_{12}. By definition of L_{12}, for all $j \geq 0$, (1) $\overline{a}_{0\ldots i_j}$ is in L_2, and (2) there exists i'_j such that $i_{j-1} \leq i'_j < i_j$ and $\overline{a}_{0\ldots i'_j}$ is in L_1. Thus, $\underline{a} \in recur(L_1)$, and $\underline{a} \in recur(L_2)$. Hence, $\underline{a} \in \mathcal{L}_1 \cap \mathcal{L}_2$.

Consider an ω-word $\underline{a} \notin recur(L_{12})$. Then there exists a position i such that for all $j \geq i$, $\overline{a}_{0\ldots j} \notin L_{12}$. We wish to establish $\underline{a} \notin \mathcal{L}_1 \cap \mathcal{L}_2$. Assume to the contrary. Since $\underline{a} \in recur(L_1)$, there exists a position $k \geq i$ such that $\overline{a}_{0\ldots k} \in L_1$. Let k' be the least position such that $k' > k$ and $\overline{a}_{0\ldots k'} \in L_2$ (such a position exists since $\underline{a} \in recur(L_2)$). By definition of L_{12}, it contains $\overline{a}_{0\ldots k'}$, leading to a contradiction.

For non-closure under complement, let $A = \{a, b\}$. Let $L = (a^*b)^*$. The response language $recur(L)$ consists of all ω-words with infinitely many b symbols. Consider the complement of $recur(L)$, that is, the ω-language $\mathcal{L} = (a + b)^*a^\omega$ that contains ω-words with only finitely many b symbols. We prove that $\mathcal{L}$ is not a response language. Suppose, to the contrary, $\mathcal{L} = recur(L')$. We show that there exists a sequence of integers $i_0, i_1, \ldots$ such that for all $j \geq 0$, the word $a^{i_0}ba^{i_1}b \cdots a^{i_j}$ is in L'. The proof is by induction.

The ω-word a^ω is in $\mathcal{L}$. Hence, infinitely many prefixes of it are in L', and hence, there exists an integer i_0 such that $a^{i_0} \in L'$.

Assume that there exist integers $i_0, i_1, \ldots i_k$ such that for all $0 \leq j \leq k$, the word $a^{i_0}ba^{i_1}b \cdots a^{i_j}$ is in L'. Consider the ω-word $a^{i_0}ba^{i_1}b \cdots a^{i_k}b\,a^\omega$. Since it belongs to $\mathcal{L}$, it has infinitely many prefixes in L', and in particular, there exists an integer i_{k+1} such that $a^{i_0}ba^{i_1}b \cdots a^{i_k}ba^{i_{k+1}}$ is in L'.

Now consider ω-word $a^{i_0}ba^{i_1}b \cdots$. It has infinitely many prefixes in L', but it contains infinitely many b symbols, and is not in $\mathcal{L}$. ■

Exercise 9.6 {`T3`} [*recur* does not distribute over intersection] Show that $\mathit{recur}(L_1) \cap \mathit{recur}(L_2)$ is not necessarily equal to $\mathit{recur}(L_1 \cap L_2)$. ■

Proposition 9.6 [Hierarchy of languages] *Every safety language and every guarantee language is a response language.*

Proof. To establish that every safety language is a response language, verify that $\mathit{safe}(L) = \mathit{recur}(\mathit{pref}(L))$. To establish that every guarantee language is a response language, verify that $\mathit{guar}(L) = \mathit{recur}(L \cdot A^*)$. ■

It follows that every obligation language is also a response language.

Exercise 9.7 {`T4`} [From guarantee to response] Prove that the ω-language $\mathcal{L}$ is a response language iff $\mathcal{L}$ is the intersection of countably many guarantee languages. ■

Exercise 9.8 {`T3`} [ω-repetition and response] Show that for every language $L \subseteq A^*$, the ω-language L^ω is a response language. ■

Persistence languages

Given a language $L \subseteq A^*$, the corresponding persistence language consists of all ω-words whose all, but finitely many, prefixes belong to L.

PERSISTENCE

For a language $L \subseteq A^*$ over an alphabet A, $\mathit{persist}(L)$ is the ω-language $\{\underline{a} \mid \exists j \geq 0.\, \forall i \geq j.\, \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *persistence language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{persist}(L)$.

While specifying requirements of a reactive module, a persistence language $\mathit{persist}(L)$ is used to specify that "something good happens eventually, and stays unchanged" as the specification requires all, but finitely many, traces to be in the set L. A classical persistence property is the eventual stabilization requirement for self-stabilizing algorithms: the system eventually attains the stable state, and stays stable.

Example 9.6 [Persistence languages] Let $A = \{a, b\}$, and let $L = A^* a^*$. Then, $persist(L) = A^* a^\omega$ is the corresponding persistence language, and consists of all ω-words with only finitely many b symbols. Observe that $persist(L)$ is neither a safety language nor a guarantee nor a response language, and its complement is a response language. ■

The response and persistence languages are closely related, namely, they are duals of each other: the complement of a response language is a persistence language, and the complement of a persistence language is a response language.

Proposition 9.7 [Duality of response and persistence] *The ω-language $\mathcal{L}$ is a response language iff the complementary language $A^\omega \backslash \mathcal{L}$ is a persistence language.*

Exercise 9.9 {T2} [Duality of response and persistence] Prove that $\mathcal{L} = recur(L)$ iff $A^\omega \backslash \mathcal{L} = persist(A^* \backslash L)$. Proposition 9.7 follows. ■

It follows that every safety or guarantee language is a persistence language.

Exercise 9.10 {T4} [Response and persistence languages] Show that an ω-language is both a response language and a persistence language iff it is an obligation language. Show that the response language $(a^* b)^\omega$ is not an obligation language, and thus, the persistence language $A^* a^\omega$ is not an obligation language. ■

Since response languages are closed under union and intersection, it follows that so are persistence languages.

Proposition 9.8 [Closure for persistence languages] *Persistence languages are closed under union and intersection, but not under complementation.*

Exercise 9.11 {T3} [From safety to persistence] Prove that the ω-language $\mathcal{L}$ is a persistence language iff $\mathcal{L}$ is the union of countably many safety languages. ■

Reactivity languages

Reactivity languages are obtained by boolean combinations of response and persistence languages. In other words, the set of reactivity languages is the least set that contains all response languages, and is closed under union, intersection, and complementation.

Reactivity

The ω-langauge $\mathcal{L} \subseteq A^\omega$ is a *1-reactivity language* if there exists a persistence language $\mathcal{L}_1$ and a response language $\mathcal{L}_2$ such that $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$. The ω-langauge $\mathcal{L} \subseteq A^\omega$ is a *k-reactivity language*, for a natural number k, if there exist k 1-reactivity languages $\mathcal{L}_1, \ldots \mathcal{L}_k$ such that $\mathcal{L} = \mathcal{L}_1 \cap \cdots \cap \mathcal{L}_k$. The ω-langauge $\mathcal{L} \subseteq A^\omega$ is a *reactivity language* if it is a k-reactivity language for some natural number k.

Remark 9.6 [Disjunctive form of reactivity] Every reactivity language $\mathcal{L}$ can, alternatively, be expressed in a disjunctive normal form: $\bigcup 0 \leq i \leq k.\mathit{recur}(L_i) \cap \mathit{persist}(L_i')$. ■

A typical 1-reactivity requirement is the conditional repetition: if the symbol a repeats infinitely often, then the symbol b also repeats infinitely often. As we will see shortly, reactivity languages are useful in specification of fairness requirements for reactive modules: an individual strong fairness requirement is a 1-reactivity language.

Example 9.7 [Reactivity languages] Let $A = \{a, b, c\}$. The ω-language consisting of ω-words with infinitely many b symbols or only finitely many a symbols is a 1-reactivity language: $\mathit{recur}((A^*b)^*) \cup \mathit{persist}(A^*(b+c)^*)$. The ω-language consisting of ω-words with infinitely many b symbols and only finitely many a symbols is a 2-reactivity language: $\mathit{recur}((A^*b)^*) \cap \mathit{persist}(A^*(b+c)^*)$. ■

Exercise 9.12 {`T5`} [Hierarchy of reactivity languages] Show that there is a 1-reactivity language that is neither a response language nor a persistence language. Then, show that, for every natural number k, there is a k-reactivity language that is not a $(k-1)$-reactivity language. ■

By definition, reactivity languages are closed under all boolean operations.

Proposition 9.9 [Closure for reactivity languages] *Reactivity languages are closed under all boolean operations.*

All the ω-languages of interest to us will be reactivity languages. Let us recap the construction of ω-languages starting from languages of words. A safety language is the set of limits of a language over words. Safe languages are closed under union and intersection, but complementing a safe language gives a guarantee language. By considering intersection of infinitely (countable) many guarantee languages, we obtain response languages. Response languages are closed under union and intersection, but complementing a response language gives a persistence language. Equivalently, persistence languages are obtained by infinite unions of safety languages. Boolean combinations of persistence and response languages give reactivity languages. The relationship among these classes is illustrated in Figure 9.1.

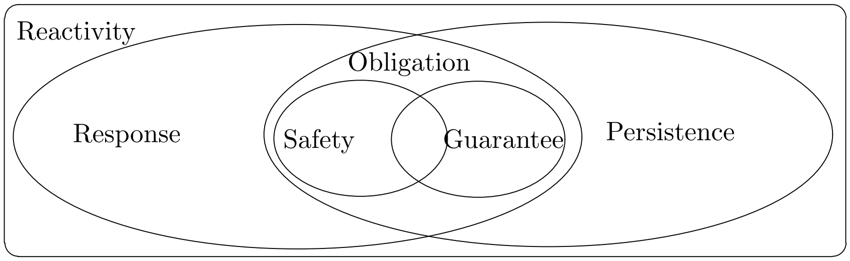


Figure 9.1: Classes of ω-languages

Exercise 9.13 {T5} [Topological characterization] Consider a metric on ω-words such that the distance between two ω-words shrinks exponentially with the length of the longest common prefix. In particular, define the metric d over the set A^ω such that $d(\underline{a}, \underline{b})$ equals 0 if $\underline{a} = \underline{b}$, and 2^{-i} otherwise, where i is the maximum integer j such that $\overline{a}_{0\ldots j} = \overline{b}_{0\ldots j}$. (1) Prove that the safe languages are precisely the closed sets of the resulting topology on ω-words. (2) Prove that the live languages are precisely the dense sets. (3) Which languages correspond to the open sets? ■

Exercise 9.14 {T4} [Machine closure in safety-progress hierarchy] Prove that every C-language is the machine-closed intersection of a safe C-language and a live C-language, where C is one of the following classes: safety; guarantee; obligation; response; persistence; reactivity. ■

9.1.4 ω-Trajectories

The execution of a transition graph G for finitely many steps results in a trajectory of G, which is a finite sequence of states. The execution of a transition graph for infinitely many steps results in a ω-trajectory of G, which is an infinite sequence of states.

> ω-TRAJECTORY
>
> Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. An *ω-trajectory* of G is an ω-word $\underline{s} = s_0 s_1 s_2 \ldots$ over the alphabet Σ of states such that for all $i \geq 0$, $s_i \rightarrow s_{i+1}$. The first state s_0 is the source. The ω-trajectory $\underline{s}$ is *initialized* if $s_0 \in \sigma^I$. The *ω-language* $\mathcal{L}_G$ of the transition graph G is the set of initialized ω-trajectories of G.

Remark 9.7 [Seriality] Let G be a serial transition graph. Then, for every state s of G, there is a source-s trajectory of G. The ω-language $\mathcal{L}_G$ is nonempty. ■

Remark 9.8 [Safety of graph languages] The ω-language $\mathcal{L}_G$ of the transition graph G is safe, and equals $\mathit{safe}(L_G)$. ■

Exercise 9.15 {T3} [ω-languages of transition graphs] Prove that the ω-language $\mathcal{L}_G$ of a transition graph G is limit-closed and fusion-closed. Conversely, let $\mathcal{L}$ be a limit-closed and fusion-closed ω-language over the alphabet A. Prove that there exists a transition graph G with states A such that $\mathcal{L}_G = \mathcal{L}$. ■

Example 9.8 [ω-trajectories of mutual exclusion protocol] Let us revisit the asynchronous solution to the mutual exclusion problem (Figure 1.23). The initialized ω-trajectories of *Pete* can be obtained from the reachable subgraph of G_{Pete} (see Figure 2.4). One possible ω-trajectory *Pete* is the periodic trajectory

$$[(o0o0)(r0o0)(i0o0)]^\omega$$

in which process P_1 repeatedly requests and enters its critical section, while process P_2 stays idle. Another possible ω-trajectory is the periodic trajectory

$$[(o0o0)(r0o0)(i0o0)(o0r1)(o0i1)(r1o1)(i1o1)(o1r0)(o1i0)(r0o0)(i0o0)]^\omega$$

in which both processes alternately request and enter thir critical sections. Since all the atoms of *Pete* are lazy, each state has a transition to itself. Consequently, $(o0o0)^\omega$ is also a ω-trajectory of *Pete*. Finally, consider the eventually periodic ω-trajectory $(o0o0)(r0o0)^\omega$ in which process P_1 requests the critical section, but never enters the critical section. ■

9.2 Fairness

Fair modules are obtained from modules by adding two types of fairness requirements.

9.2.1 Weak Fairness

A nondeterministic update command may offer, for a given state, several choices for updating the variables. For instance, consider the module *AsyncCount*:

```
module $AsyncCount$ is
  interface $Count$:
  atom controls $count$
    init
      $[\!]$ $true \to count' := 0$
    update
      $[\!]$ $true \xrightarrow{\alpha} count' := count + 1$
      $[\!]$ $true \xrightarrow{\beta} count' := count$
```

The counter is initially zero. The update action of the module has two guarded assignments. The guarded assignment α is enabled in every update round, and increments the counter. The guarded assignment β is also enabled in every update round, and leaves the counter unchanged. During the execution of the module, the choice between executing α and executing β is nondeterministic. Thus, for every natural number i, the counter may stay unchanged for the first i update rounds, and get updated to 1 in the round $(i+1)$. This is a convenient abstraction of the assumption that the rate at which the counter is incremented is unknown (or irrelevant). However, consider the limit ω-trajectory $\underline{s}$ in which the counter never gets updated: for every $i \geq 0$, $s_i[\mathit{count}] = 0$. The ω-trajectory $\underline{s}$ is *unfair* to the update choice α; the choice α is enabled in every round, and never executed.

Definition of fair modules provides a way to rule out the ω-trajectory $\underline{s} = 0^\omega$. This is achieved by requiring that the resolution of the update choices be *weakly fair* to the choice α. The module *FairCount* is a fair version of the asynchronous counter *AsyncCount*:

```
module $\mathit{FairCount}$ is
  interface $\mathit{Count}$:
  atom controls $\mathit{count}$
    init
      $[\!] \; \mathit{true} \rightarrow \mathit{count}' := 0$
    update weaklyfair $\alpha$
      $[\!] \; \mathit{true} \xrightarrow{\alpha} \mathit{count}' := \mathit{count} + 1$
      $[\!] \; \mathit{true} \xrightarrow{\beta} \mathit{count}' := \mathit{count}$
```

The annotation **weaklyfair** α requires that the guarded command α is executed infinitely often. The ω-trajectories that satisfy this requirement will be called fair trajectories. The ω-trajectory $\underline{s} = 0^\omega$ is not a fair trajectory of *FairCount*. The ω-trajectory $\underline{t} = 012345^\omega$ in which the counter is not incremented beyond 5, is also not a fair trajectory of *FairCount*. On the other hand, consider the ω-trajectory $\underline{u} = 0123\cdots$ in which the counter is incremented in every round. The update choice β is always enabled, but never executed, and yet, the ω-trajectory $\underline{u}$ is a fair trajectory of *FairCount*. This is because *FairCount* makes no assumption about fairness towards the choice β.

In general, an update choice α for an atom U of a module P is a subset of the update action update_U. Consider an ω-trajectory $\underline{s}$ of P. For $i \geq 1$, consider the update round i in which the state s_i is determined from the state s_{i-1}. Recall that the atom U is executed only after the updated values of the variables in $\mathsf{await}X_U$ have been determined. The update choice α is said to be *available* in round i, if the state s_{i-1}, together with the values of the awaited variables in state s_i, satisfies the guard p_γ of some guarded assignment γ in α. The update choice is said to be *executed* in round i, if the values of the controlled variables in state s_i are determined by executing some available guarded assignment γ in α. The ω-trajectory $\underline{s}$ is *weakly fair* with respect to α, if there is no round i such that the choice α is available in every round after i, and is not executed in any round after i. Intuitively, a weakly fair update choice cannot be available forever without being executed.

9.2.2 Strong Fairness

Weak fairness requires that a choice that is continuously available is eventually executed. Suppose a choice is available in all even rounds, and unavailable in all odd rounds, and is never executed. This scenario meets the requirement of weak fairness, but may not be a reasonable scenario in certain cases. For example, consider the module *LossyBuffer*:

```
module LossyBuffer is
  interface $y$:
  external $x$:
  passive atom controls $y$ reads $x, y$ awaits $x$
    update
      $\|\ x? \xrightarrow{\alpha} y?$
      $\|\ x? \xrightarrow{\beta}$
```

In every round in which the external event x is present, both the update choices α and β are available. If the choice α is executed, then the interface event y is issued, and if the choice β is executed, then the module stutters without issuing the event y. The module *LossyBuffer* can be viewed as an abstraction of a lossy buffer, that either outputs the input event, or loses the input event. Consider the periodic ω-trajectory

$$\underline{s} \;=\; [(y=0, x=0)(y=0, x=1)]^{\omega};$$

in every update round the external event x is present, but the module always executes the update choice β. Requiring weak fairness for the choice α will rule out the ω-trajectory $\underline{s}$. Now consider the periodic ω-trajectory

$$\underline{t} \;=\; [(y=0, x=0)(y=0, x=0)(y=0, x=1)(y=0, x=1)]^{\omega};$$

the external event x is present only in alternate rounds, and the module always executes the update choice β. Note that the ω-trajectory $\underline{t}$ is weakly fair with respect to the choice α, because the choice α is unavailable in infinitely many rounds. If we wish to model the assumption that only some, but not all, messages are lost, then we would like to rule out the ω-trajectory $\underline{t}$ also. This can be achieved by requiring that the resolution of the update choices be *strongly fair* to the choice α: if α is available in infinitely many round, then α is executed in infinitely many rounds. The module *FairBuffer* is a fair version of the buffer *LossyBuffer*:

```
module FairBuffer is
  interface $y$:
  external $x$:
  passive atom controls $y$ reads $x, y$ awaits $x$
    update stronglyfair $\alpha$
      [] $x? \xrightarrow{\alpha} y?$
      [] $x? \xrightarrow{\beta}$
```

The annotation **stronglyfair** α classifies an ω-trajectory to be fair if either the update choice α is executed infinitely often, or is available only finitely often. Consequently, the ω-trajectory $\underline{t}$ is not a fair trajectory of *FairBuffer*. Intuitively, a strongly fair update choice cannot be available infinitely often without being executed.

A fair module may declare, for each atom, some of its choices to be weakly fair, and some to be strongly fair. An *update choice* of a reactive module P is a subset of the update command update_U, for some atom $U \in \mathsf{atoms}_P$.

Fair module

A *fair module* $\mathcal{P}$ consists of (1) a reactive module P, (2) [weak fairness] a finite set WeakF_P of update choices of P, and (3) [strong fairness] a finite set $\mathit{StrongF}_P$ of update choices of P.

Since one of the components of a fair module is a reactive module, we freely attribute the properties of a reactive module to a fair module. For instance, every fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$ defines the transition graph $G_{\mathcal{P}} = G_P$. Different classifications of modules, such as finite versus infinite, closed versus open, apply to fair modules also. For instance, an asynchronous fair module is a fair module all of whose interface variables are controlled by lazy atoms.

9.3 Fair Graphs

We define the semantics of a fair module by associating a fair graph with it. In Chapter 7, we defined automata by augmenting observation structures with accepting regions. The accepting region of an automaton classifies trajectories into accepting and non-accepting, and consequently, automata can define languages that are not necessarily prefix-closed. Now we wish to augment a transition graph with an accepting condition that will classify its ω-trajectories into accepting and non-accepting. By considering only accepting ω-trajectories, we will be able to define live ω-languages.

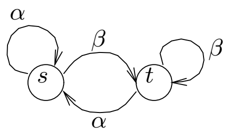


Figure 9.2: Fair graph

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. An *action* of G is a subset of the transition relation $\rightarrow$. For an action α of G, we write $s \xrightarrow{\alpha} t$ if the transition (s, t) belongs to α. A *fairness constraint* f for a transition graph G is a pair (α, β) of actions of G, and a *fairness assumption* F for G is a finite set of fairness constraints for G.

FAIR GRAPH

A *fair graph* $\mathcal{G}$ consists of (1) a transition graph G, and (2) [fairness] a fairness assumption F for G.

Intuitively, a fairness constraint (α, β) requires that if the action α repeats infinitely often then the action β also repeats infinitely often. Fair trajectories of a fair graph are those ω-trajectories that meet the requirements stipulated by all the fairness constraints.

FAIR TRAJECTORY

Let G be a transition graph. An ω-trajectory $\underline{s}$ of of G is *α-fair*, for an action α of G, if $s_i \xrightarrow{\alpha} s_{i+1}$ for infinitely many natural numbers i. The ω-trajectory $\underline{s}$ is *f-fair*, for a fairness constraint $f = (\alpha, \beta)$ of G, if either $\underline{s}$ is not α-fair, or $\underline{s}$ is β-fair. The ω-trajectory $\underline{s}$ is *F-fair*, for a fairness assumption F of G, if $\underline{s}$ is f-fair for all fairness constraints f in F. A *fair trajectory* of a fair graph $\mathcal{G} = (G, F)$ is an F-fair ω-trajectory of G. The *fair language* $\mathcal{L}_{\mathcal{G}}$ of a fair graph $\mathcal{G}$ is the set of initialized fair trajectories of $\mathcal{G}$.

Remark 9.9 [Fair graphs] Let $\mathcal{G} = (G, F)$ be a fair graph. The ω-language $\mathcal{L}_{\mathcal{G}}$ is a subset of the safe language $\mathcal{L}_G$. Furthermore, if the fairness assumption F is an empty set then every ω-trajectory is fair, and $\mathcal{L}_{\mathcal{G}}$ equals $\mathcal{L}_G$. ■

Fair languages are not necessarily safe languages, and different fairness assumptions can be used to identify different subsets of the ω-trajectories of a transition graph.

Example 9.9 [Fair graph] Consider the two-state transition graph shown in Figure 9.2. Both states are initial. The actions α and β contain two transitions each, as shown. An ω-trajectory $\underline{t}$ is α-fair if it contains infinitely many visits to the state s, and $\underline{t}$ is β-fair if it contains infinitely many visits to the state t. Consider the fairness constraint $f_1 = (\rightarrow, \beta)$. The ω-trajectory $\underline{t}$ is f_1-fair if it contains infinitely many visits to the state t. Thus, the fair language of the fair graph $(G, \{f_1\})$ is $(s^*t)^\omega$ (note that this is a response language).

Different fairness constraints can define the same language. For instance, for the fairness constraint $f_2 = (\alpha, \beta)$, the ω-trajectory $\underline{t}$ is f_2-fair iff it is f_1-fair. Thus, the fair languages of $(G, \{f_1\})$ and $(G, \{f_2\})$ coincide.

Let $f_3 = (\rightarrow, \alpha)$. Then, an ω-trajectory is f_3-fair if it contains infinitely many visits to the state s. The fair language of the fair graph $(G, \{f_1, f_3\})$ contains ω-trajectories that have infinitely many visits to both the states, and equals $(s^*t)^\omega \cap (t^*s)^\omega$ (note that this is a reactivity language).

Consider the fairness constraint $f_4 = (\alpha, \emptyset)$. Observe that there is no $\emptyset$-fair trajectory. Thus, an ω-trajectory $\underline{t}$ is f_4-fair iff it is not α-fair; that is, if it contains only finitely many visits to the state s. The fair language of the fair graph $(G, \{f_4\})$ is $(s+t)^*t^\omega$ (note that this is a persistence language). ■

The above example shows that fair languages of fair graphs can be reactivity languages. Can fair graphs define more complex languages? The answer is no.

Proposition 9.10 [Languages of fair graphs] *If $\mathcal{G}$ is a fair graph, then the language $\mathcal{L}_\mathcal{G}$ is a reactivity language.*

Proof. Let G be a transition graph. For an action α, let L_α be the set of initialized trajectories $\overline{s}_{0\ldots m}$ of G such that $s_{m-1} \xrightarrow{\alpha} s_m$. Now, an initialized ω-trajectory $\underline{s}$ of G is α-fair iff $\underline{s}$ belongs to the response language $\mathit{recur}(L_\alpha)$. For a fairness constraint $f = (\alpha, \beta)$, an initialized ω-trajectory $\underline{s}$ of G is f-fair iff it is either α-unfair or β-fair; that is, iff it belongs to $(\Sigma^\omega \backslash \mathit{recur}(L_\alpha)) \cup \mathit{recur}(L_\beta)$. Thus, the set of f-fair ω-trajectories is a union of a persistence and a response language, that is, a 1-reactivity language.

Consider the fair graph $\mathcal{G} = (G, F)$. Verify that $\mathcal{L}_\mathcal{G}$ equals

$$\bigcap(\alpha, \beta) \in F.\ (\Sigma^\omega \backslash \mathit{recur}(L_\alpha)) \cup \mathit{recur}(L_\beta).$$

It follows that if F has k fairness constraints then $\mathcal{L}_\mathcal{G}$ is a k-reactivity language. ■

Types of fairness constraints

We consider three types of fairness constraints.

Weak-fair constraint

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. A fairness constraint (α, β) for G is a *weak-fair* constraint if α equals $\rightarrow$. A weak-fair graph is a fair graph (G, F) such that F contains only weak-fair constraints.

While a fairness constraint specifies infinite repetition of an action conditioned upon the repetition of another, a weak-fair constraint specifies unconditional repetition of an action.

Remark 9.10 [Weak-fair constraints] For a weak-fair constraint $f = (\rightarrow, \beta)$, f-fair trajectories are precisely the β-fair trajectories. For a weak-fair graph $\mathcal{G} = (G, F)$, $\mathcal{L}_{\mathcal{G}}$ is the response language

$$\bigcap(\rightarrow, \beta) \in F.\ \mathit{recur}(L_\beta).$$

■

Sometimes we consider actions that are defined by regions. For a region σ of a transition graph G, the action $\alpha_\sigma = \{(s,t) \mid s \in \sigma \text{ and } s \rightarrow t\}$ contains all transitions with source in σ. Consequently, we will use regions in place of actions when there is no cause for confusion. For instance, an ω-trajectory $\underline{s}$ is σ-fair, for a region σ, if it is α_σ-fair, or equivalently, if $s_i \in \sigma$ for infinitely many i. For two regions σ and τ, the fairness constraint $(\alpha_\sigma, \alpha_\tau)$ will be denoted, more concisely, as (σ, τ).

Machine closure

A fair graph $\mathcal{G}$ is said to be machine-closed if every trajectory of $\mathcal{G}$ can be extended to a fair trajectory. Machine closure ensures that a stepwise simulator for fair graphs cannot paint itself into a corner from which the fairness constraints cannot be satisfied. Machine closure for fair graphs is the analog of seriality for transition graphs.

Machine-closed fair graph

A fair graph $\mathcal{G}$ is machine-closed if for every state s of $\mathcal{G}$ there exists a source-s fair trajectory of $\mathcal{G}$.

Remark 9.11 [Machine-closed fair graph] If $\mathcal{G} = (G, F)$ is machine-closed then $\mathit{pref}(\mathcal{L}_{\mathcal{G}}) = \mathit{pref}(\mathcal{L}_G) = L_G$. ■

Remark 9.12 [Machine-closure] Recall the definition of machine-closure from Section 9.1.2: for a safe language $\mathcal{L}_S$ and a live language $\mathcal{L}_L$, the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is machine-closed if every prefix of $\mathcal{L}_S$ can be extended to an ω-word in $\mathcal{L}_S \cap \mathcal{L}_L$.

The above definition of machine-closure has the same spirit. To be precise, consider a fair graph (G, F). Let $\mathcal{L}_S$ be the set of all ω-trajectories of G (this is a superset of $\mathcal{L}_G$ that contains only initialized ω-trajectories). Verify that the set $\mathcal{L}_S$ is safe. Let $\mathcal{L}_L$ be the set of all ω-words $\underline{s}$ over the alphabet Σ_G that are f-fair for every fairness constraint $f \in F$. This set includes all fair trajectories of $\mathcal{G}$, along with ω-words that are not necessarily ω-trajectories of G. Alternatively, the set $\mathcal{L}_L$ is the fair language of a fair graph with state-space Σ_G, initial region Σ_G, transition relation $\Sigma_G \times \Sigma_G$, and fairness assumption F. Verify that the set $\mathcal{L}_L$ is live. Now, the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is machine-closed, that is, $\mathit{pref}(\mathcal{L}_S \cap \mathcal{L}_L) = \mathit{pref}(\mathcal{L}_S)$ iff the fair graph $\mathcal{G}$ is machine-closed, that is, every state is the source of some fair trajectory. ■

Exercise 9.16 {T3} [Intersection and machine-closure] Suppose f_1 and f_2 are two fairness constraints for a transition graph G. Prove or disprove the claim that $(G, \{f_1, f_2\})$ is machine-closed iff both the fair graphs $(G, \{f_1\})$ and $(G, \{f_2\})$ are machine-closed. ■

Local fairness

A local fairness constraint is a type of fairness constraint that, intuitively, stipulates a fair resolution of choice, and nothing more.

Local fairness

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. A fairness constraint (α, β) is *local* if for all $s \xrightarrow{\alpha} t$, there is a state $u \in \Sigma$ such that $s \xrightarrow{\beta} u$. A *locally-fair* graph is a fair graph (G, F) such that (1) G is serial, and (2) F contains only local fairness constraints.

For a locally-fair constraint $f = (\alpha, \beta)$, whenever the action α is available, so is β. Consequently, in every α-fair ω-trajectorry, the choice to execute β is also available infinitely often, and thus, an f-unfair trajectory can be produced only by continuously ignoring the choice to execute β.

Figure 9.3 shows an interpreter for producing fair trajectories of a locally-fair graph. The input to the interpreter is a locally-fair graph $\mathcal{G}$, and a state s of $\mathcal{G}$. The algorithm uses two subroutines. The subroutine *Available*(α, s) takes an action α and a state s as input, and returns Yes if there is a state u such that the pair (s, u) belongs to the action α. The subroutine *Execute*(α, s) takes an action α and a state s such that *Available*(α, s) holds, and returns a state u such that the pair (s, u) belongs to the action α.

The algorithm maintains a queue of the fairness constraints. The queue is initialized to contain all the fairness constraints of $\mathcal{G}$ in some arbitrary order. Let us say that a fairness constraint $f = (\alpha, \beta)$ is enabled in a state if the action

Algorithm 9.1 [Execution of locally fair graph]

Input: a locally fair graph $\mathcal{G} = (\Sigma, \sigma^I, \rightarrow, F)$ and a state s;
Output: a source-s fair trajectory $\underline{s}$ of $\mathcal{G}$.

Queue: a queue of fairness constraints

```
Initialization
  $s_0 := s$;
  $Queue$ contains all the fairness constraints in $F$ in some order.

Update rounds.
for $i := 0$ to $\infty$ do
  Let $Queue$ be $f_1 f_2 \dots f_n$ with $f_k = (\alpha_k, \beta_k)$ for $1 \leq k \leq n$;
  if $Available(\alpha_k, s_i)$ for some $1 \leq k \leq n$ then
      $j := min\{k \mid Available(\alpha_k, s_i)\}$;
      $s_{i+1} := Execute(\beta_j, s_i)$;
      $Queue := f_1 \dots f_{j-1} f_{j+1} \dots f_n f_j$
    else $s_{i+1} := Execute(\rightarrow, s_i)$
    fi
  od.
```

Figure 9.3: An interpreter for locally fair graphs

α is available. At every step, the algorithm checks if there is some fairness constraint that is enabled at the current state. If no such constraint exists, an arbitrary successor of the current state is chosen to be the next state (since G is serial, each state has at least one successor). If there are one or more enabled fairness constraints, then the algorithm chooses the constraint $f = (\alpha, \beta)$ such that f is enabled, and all the constraints that appear before f in the queue are disabled. Since f is local, availability of α implies availability of β, and the algorithm extends the trajectory by choosing some β-successor of the current state. Finally, the constraint f is moved from its current position in the queue to the end of the queue so that the other constraints get priorities in the subsequent rounds.

Proposition 9.11 [Execution of locally fair graph] *Given a locally-fair graph $\mathcal{G}$ and a state s, Algorithm 9.1 produces a source-s fair trajectory of $\mathcal{G}$.*

Proof. Let $\underline{s}$ be the ω-trajectory produced by the algorithm of Figure 9.3 in the limit. For every fairness constraint $f = (\alpha, \beta)$ of $\mathcal{G}$, and for every natural number i, let $\mathit{Unfair}(f, i)$ be true iff $(s_i, s_{i+1}) \notin \beta$; let $\mathit{StrongUnfair}(f, i)$ be true iff $\mathit{Unfair}(f, i)$ and $\mathit{Available}(\alpha, s_i)$; and let $\mathit{Rank}(f, i)$ be the position $1 \leq k \leq n$ of the constraint f in the queue at the beginning of round i. Observe that if $\mathit{Unfair}(f, i)$ then, in round i, the constraint f is not moved to the end of the queue, and thus, its rank cannot increase.

(1) For all $f \in F$ and $i \geq 0$, if $\mathit{Unfair}(f, i)$ then
$\mathit{Rank}(f, i+1) \leq \mathit{Rank}(f, i)$.

If a fairness constraint f is enabled in round i, then it is executed unless some other constraint f' with $\mathit{Rank}(f', i) < \mathit{Rank}(f, i)$ is also enabled, in which case the constraint with least rank among the enabled constraints is executed, and moved to the end of the queue, which decreases the rank of f. This leads to:

(2) For all $f \in F$ and $i \geq 0$, if $\mathit{StrongUnfair}(f, i)$ then
$\mathit{Rank}(f, i+1) < \mathit{Rank}(f, i)$.

We wish to establish that $\underline{s}$ is a fair trajectory of $\mathcal{G}$. Consider $f = (\alpha, \beta)$. We prove that if $\underline{s}$ is not β-fair then it is not α-fair. Suppose $\underline{s}$ is not β-fair. Then, there exists $i \geq 0$ such that for all $j \geq i$, $\mathit{Unfair}(f, j)$ holds. By (1), for all $j \geq i$, $\mathit{Rank}(f, j+1) \leq \mathit{Rank}(f, j)$. Since for all $j \geq 0$, $\mathit{Rank}(f, j) \geq 1$, there can be only finitely many rounds j such that $\mathit{Rank}(f, j+1) < \mathit{Rank}(f, j)$. By (2), there can be only finitely many rounds j such that $\mathit{StrongUnfair}(f, j)$. Hence, the action α is available only in finitely many rounds, and $\underline{s}$ is α-unfair. ■

The execution strategy to produce fair trajectories also implies the following proposition.

Proposition 9.12 [Machine closure of local fairness] *Every locally-fair graph is machine-closed.*

Exercise 9.17 {T3} [Execution of weak-locally-fair graphs] Let $\mathcal{G}$ be a locally-fair weak-fair graph. Show that, to produce a fair trajectory of $\mathcal{G}$, it suffices to maintain a modulo-$|F|$ counter, instead of the queue used by the interpreter of Figure 9.3. ■

9.4 Fair Modules

We associate a fair graph with every fair module by mapping the weak and strong fairness constraints of the module to the fairness constraints for the associated transition graph. Towards this goal, we associate with every update choice a of P, two actions of the graph G_P. The *availability action* of an update choice a contains a transition $s \rightarrow_P t$ if the choice a is enabled according to the values of the read variables in s and the awaited variables in t. The *execution action* of an update choice a contains a transition $s \rightarrow_P t$ if the values of the controlled variables in t are assigned by executing the choice a.

Actions of an update choice

Let P be a module, and a be an update choice of an atom U of P. The *availability action* $avail_a$ contains a transition $s \rightarrow_P t$ of P iff there is a guarded assignment γ in a such that

$$(\mathsf{read}X_U[s] \cup \mathsf{await}X'_U[t']) \;\in [\![p_\gamma]\!].$$

The *execution action* $exec_a$ contains a transition $s \rightarrow_P t$ of P iff there is a guarded assignment γ in a such that

$$(\mathsf{read}X_U[s] \cup \mathsf{await}X'_U[t'], \mathsf{ctr}X'_U[t']) \;\in [\![a]\!].$$

Remark 9.13 [Actions of an update choice] For every update choice a of a module P, the action $exec_a$ is a subset of the action $avail_a$. ■

Example 9.10 [Actions of an update choice] Consider the update choice α of the module *AsyncCount*. The availability action $avail_\alpha$ contains all transitions of *AsyncCount*. The execution action $exec_\alpha$ contains the transition $s \rightarrow t$ if $count[t] = count[s] + 1$.

Consider the update choice α of the module *LossyBuffer*. The availability action $avail_\alpha$ contains the transition $s \rightarrow t$ iff $x[t] \neq x[s]$. The execution action $exec_\alpha$ contains the transition $s \rightarrow t$ iff $x[t] \neq x[s]$ and $y[t] \neq y[s]$. ■

Weak fairness for a choice a requires that the choice cannot be available forever without being executed, and strong fairness for a choice a requires that if the choice is available infinitely often then it is executed infinitely often.

Fairness constraints of an update choice

Let P be a module, and a be an update choice of P. The *weak-fairness constraint* f_a^W of a is the pair $(\rightarrow_P, \mathit{exec}_a \cup (\rightarrow_P \setminus \mathit{avail}_a))$. The *strong-fairness constraint* f_a^S of a is the pair $(\mathit{avail}_a, \mathit{exec}_a)$.

An ω-trajectory $\underline{s}$ of P is weakly fair to the update choice a precisely when it is f_a^W-fair: for infinitely many rounds $i \geq 0$, $(s_i, s_{i+1}) \in \mathit{exec}_a$ or $(s_i, s_{i+1}) \notin \mathit{avail}_a$. An ω-trajectory $\underline{s}$ of P is strongly fair to the update choice a precisely when it is f_a^S-fair: if for infinitely many rounds $i \geq 0$, $(s_i, s_{i+1}) \in \mathit{avail}_a$, then for infinitely many rounds $j \geq 0$, $(s_j, s_{j+1}) \in \mathit{exec}_a$.

Remark 9.14 [Strong fairness implies weak fairness] Let P be a module, a be an update choice of P, and $\underline{s}$ be an ω-trajectory of P. If $\underline{s}$ is f_a^S-fair then $\underline{s}$ is f_a^W-fair. The converse does not hold. ■

Remark 9.15 [Local fairness] Let P be a module, and a be an update choice of P. Both the fairness constraints f_a^S and f_a^W are local fairness constraints on the transition graph G_P. ■

The fair graph of a reactive module is obtained by adding all the fairness constraints corresponding to the declaration of weak and strong fair update choices.

Fair graph of a fair module

For a fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$, the fairness assumption $F_\mathcal{P}$ is the set

$$\{f_a^W \mid a \in \mathit{WeakF}_P\} \cup \{f_a^S \mid a \in \mathit{StrongF}_P\}$$

of fairness constraints of G_P. The fair module $\mathcal{P}$ defines the fair graph $\mathcal{G}_\mathcal{P} = (G_P, F_\mathcal{P})$.

A fair trajectory of a fair module $\mathcal{P}$ is a fair trajectory of the fair graph $\mathcal{G}_\mathcal{P}$.

Remark 9.16 [Fair trajectories of a fair module] The set of fair trajectories of a fair module P is a reactivity language. Furthermore, if the module employs only weak fairness, that is, for every atom U, the set $\mathit{StrongF}_U$ is empty, then the graph $\mathcal{G}_P$ is weak-fair, and the set of fair trajectories of a fair module is a response language. ■

9.4.1 Operations on Fair Modules

As in case of reactive modules, we combine fair modules using three operations —parallel composition, variable renaming, and variable hiding.

Parallel Composition

The composition operation combines two fair modules into a single fair module whose behavior captures the interaction between the two component modules. Again, composition is the key operation that allows modular descriptions of complex systems. Two fair modules can be composed only if their modules are compatible. Observe that for compatible reactive modules P and Q, an update choice of P is also an update choice of $P \| Q$. Consequently, to compose two compatible fair modules, we simply compose their reactive modules, and take union of the respective weak and strong fairness constraints.

> FAIR MODULE COMPOSITION
>
> If $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$ and $\mathcal{Q} = (Q, \mathit{WeakF}_Q, \mathit{StrongF}_Q)$ are compatible fair modules, then the *parallel composition* $\mathcal{P} \| \mathcal{Q}$ is the fair module $(P \| Q, \mathit{WeakF}_P \cup \mathit{WeakF}_Q, \mathit{StrongF}_P \cup \mathit{StrongF}_Q)$.

The composition operator over fair modules has all the properties listed for the composition operator over modules in Chapter 1. For instance, the composition operator is commutative and associative.

Variable Renaming

As in modules, we can rename variables to avoid name-conflicts among private variables, and to create multiple copies. To apply a variable renaming to a fair module, we simply apply the renaming to each of its components.

> RENAMING OF FAIR MODULE
>
> Given a fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$, and a renaming ρ for the set X_P of module variables, the *renamed module* $\mathcal{P}[\rho]$ is the fair module with the module $P[\rho]$, the set $\{a[\rho] \mid a \in \mathit{WeakF}_P\}$ of weakly-fair update choices, and the set $\{a[\rho] \mid a \in \mathit{StrongF}_P\}$ of strongly-fair update choices.

Variable Hiding

The hiding of interface variables of a fair module allows abstractions at different levels of details. Hiding of an interface variable of fair modules changes only its variable declaration.

> HIDING OF FAIR MODULE
>
> Given a fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$, and a typed variable x, by *hiding* x in P we obtain the fair module $(\textbf{hide } x \textbf{ in } P, \mathit{WeakF}_P, \mathit{StrongF}_P)$, denoted **hide** x **in** $\mathcal{P}$.

9.4.2 Machine Closure and Receptiveness

We know that the transition graph of a module is serial. This means that a stepwise interpreter of a module never gets stuck, and can always extend a trajectory by adding one more step. The analog of seriality in the case of ω-trajectories is machine-closure. It says that every finite trajectory can be extended to a fair trajectory. In particular, there is a strategy to systematically resolve the choices so that the limit ω-trajectory is a fair one. Proposition 9.12 asserts that every locally-fair graph is machine-closed. Since the fairness constraints of an update choice are local, it follows that the fair graph of a fair module is locally-fair, and hence, machine-closed.

Proposition 9.13 [Machine closure of fair modules] *The fair graph $\mathcal{G}_\mathcal{P}$ of a fair module $\mathcal{P}$ is locally-fair, and hence, machine-closed.*

Exercise 9.18 {P2} [Fair module execution] Recall the interpreter for reactive modules from Chapter 2. Using the strategy outlined in Figure 9.3 for executing locally-fair graphs, write an interpreter for fair modules. The input to the interpreter should be a fair module $\mathcal{P}$, and it should produce a fair trajectory of $\mathcal{P}$, if we let it execute forever. ■

For reactive modules, the property of seriality is preserved under parallel composition. It means that every trajectory of a module can be extended by adding another state no matter how the environment updates the external variables. The same applies to machine-closure also. Every finite trajectory of a fair module can be extended to a fair trajectory, no matter how the environment updates the external variables in each round. Thus, the ability to produce a fair trajectory does not demand cooperation from the environment. This fact is captured by the next proposition.

Proposition 9.14 [Receptiveness] *Let $\mathcal{P}$ be a fair module, s be a state of $\mathcal{P}$, and Q be a module that is compatible with $\mathcal{P}$. There exists an ω-trajectory $\underline{t}$ of the composition $\mathcal{P}\|Q$ such that $X_\mathcal{P}[t_0] = s$ and $X_\mathcal{P}[\underline{t}]$ is a fair trajectory of $\mathcal{P}$.*

Proof. By definition of composition, there is a state t of $\mathcal{P}\|Q$ such that $X_\mathcal{P}[t] = s$. By Proposition 9.13, the fair module $\mathcal{P}\|Q$ is machine-closed, and hence, has a source-t fair trajectory. ■

The ability to produce a fair trajectory in the face of an adversarial environment is known as *receptiveness*. Consider a module P and a set $\mathcal{L}$ of ω-trajectories of P. Consider a two-player game in which the *protagonist* attempts to produce an ω-trajectory in $\mathcal{L}$, while the *adversary* tries to prevent this. Initially, the adversary chooses a trajectory $s_{0..m}$ of P. In each round, the adversary chooses the new external state, and then the protagonist extends the current trajectory by choosing the new controlled state. The choices of the protagonist are constrained by the transition relation of the module P. The protagonist wins the game if the resulting infinite ω-trajectory belongs to $\mathcal{L}$. The ω-language $\mathcal{L}$ is *receptive* for the module P if the protagonist has a winning strategy. Now, Proposition 9.14 can reformulated to assert that for every fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$ the set $\mathcal{L}_{\mathcal{P}}$ of fair trajectories is receptive for P.

9.5 Examples of Fair Modules

We revisit examples of modules, and add appropriate fairness constraints.

9.5.1 Shared-variables Protocols

Our canonical example of a shared-variable protocol is mutual exclusion. So far we have considered only the mutual exclusion requirement of such protocols, namely, that the two processes are not inside their critical sections simultaneously. When we consider ω-trajectories, the parallel composition of the two processes should also meet the *deadlock-freedom* requirement: if some process requests an entry to its critical section, then some process eventually enters its critical section. Consider a protocol that never allows any process to enter the critical section; that is, none of its guarded assignments assign the value *inC* to *pc*. Such a protocol satisfies the mutual exclusion requirement, but not the deadlock-freedom requirement, and hence, is not an acceptable solution to the problem. An even stronger requirement for the problem is the *starvation-freedom* property that: if a process requests an entry to its critical section, then the same process eventually enters its critical section. Thus, while deadlock-freedom admits solutions that always prefer one process over the other, starvation-freedom requires a fair resolution of the choice.

Both deadlock-freedom and starvation-freedom are liveness properties, and cannot be violated by trajectories. Requiring all ω-trajectories to satisfy such requirements is too strong, as it would rule out asynchronous solutions like *Pete*. For instance, the eventually periodic ω-trajectory $(o0o0)(r0o0)^\omega$ of *Pete* does not meet the deadlock-freedom requirement. Instead, we will add reasonable fairness assumptions, and require that all the fair trajectories satisfy the liveness properties.

Our formulation of the problem allows each process to remain outside the critical section for an arbitrary number of rounds, and to remain inside the critical section for an arbitrary number of rounds. Consequently, the update of *pc* from *outC* to *reqC* is nondeterministic. We do not add any fairness on the resolution of this choice: an ω-trajectory in which some process never requests an entry, is an acceptable scenario. The update of *pc* from *inC* to *outC* is also nondeterministic. To model the assumption that a process may not stay inside its critical section forever, we add weak fairness for the choice to exit the critical section.

Synchronous mutual exclusion

For the synchronous solution *SyncMutex* the update from *reqC* to *inC* is deterministic. Consequently, the only fairness assumption concerns the choice to leave the critical section. The resulting fair modules are shown in Figure 9.4. In Chapter 9, we will present an algorithm to verify that the module *FairSyncMutex* satisfies both the liveness requirements of deadlock-freedom and starvation-freedom.

Asynchronous mutual exclusion

The asynchronous solution *Pete* uses nondeterminism to model the assumption that the two processes execute at independent speeds. We would like to add fairness to rule out executions in which stuttering is always preferred over another available choice. For instance, if a choice to update *pc* from *reqC* to *inC* is available, then it should eventually be executed. The resulting fair modules are shown in Figure 9.5.

9.5.2 Circuits

Synchronous circuits

Recall the definitions of the logic gates *SyncAnd* and *SyncNot* from which all combinational circuits can be built. Both these modules are deterministic: in every round, once the values of the awaited external variables are determined, precisely one guarded assignment is available. Consequently, there is no need to add fairness constraints in the definitions of the logic gates. Our basic unit for memory cell is the module *SyncLatch* (Figure 1.17) with two boolean inputs *set* and *reset*. In every round, if the updated values of both the inputs *set* and *reset* are 1, then the private state of the latch is updated nondeterministically. In this case, both the guarded assignments are available, and the next state may be either 0 or 1. Nondeterminism, in this context, models the fact that the behavior of the latch is unknown, and we expect the latch to be used in an environment that never sets both *set* and *reset* simultaneously to 1. Consequently, we do not add any fairness constraint to *SyncLatch*.

Asynchronous circuits

Recall the modeling of asynchronous logic gates from Section 1.3.4. An asynchronous logic gate is unstable when its output is not according the desired function of the inputs. The gate can stay unstable for an arbitrary number of rounds, and then, it becomes stable by changing its output. Now, we can use fairness to ensure that the gate does not stay unstable forever. The asynchronous AND gate with fairness is shown in Figure 9.6. It requires weak fairness

```
module $\mathcal{Q}_1$ is
  interface $pc_1 \colon \{outC, reqC, inC\}$
  external $pc_2 \colon \{outC, reqC, inC\}$
  atom
    controls $pc_1$
    reads $pc_1, pc_2$
    init
      $[\!] \; true \to pc'_1 := outC$
    update weaklyfair $\alpha_1$
      $[\!] \; pc_1 = outC \to$
      $[\!] \; pc_1 = outC \to pc'_1 := reqC$
      $[\!] \; pc_1 = reqC \wedge pc_2 \neq inC \to pc'_1 := inC$
      $[\!] \; pc_1 = inC \to$
      $[\!] \; pc_1 = inC \overset{\alpha_1}{\to} pc'_1 := outC$

module $\mathcal{Q}_2$ is
  interface $pc_2 \colon \{outC, reqC, inC\}$
  external $pc_1 \colon \{outC, reqC, inC\}$
  atom
    controls $pc_2$
    reads $pc_1, pc_2$
    init
      $[\!] \; true \to pc'_2 := outC$
    update weaklyfair $\alpha_2$
      $[\!] \; pc_2 = outC \to$
      $[\!] \; pc_2 = outC \to pc'_2 := reqC$
      $[\!] \; pc_2 = reqC \wedge pc_1 = outC \to pc'_2 := inC$
      $[\!] \; pc_2 = inC \to$
      $[\!] \; pc_2 = inC \overset{\alpha_2}{\to} pc'_2 := outC$

$FairSyncMutex \;=\; \mathcal{Q}_1 \parallel \mathcal{Q}_2$
```

Figure 9.4: Fair synchronous mutual exclusion

```
module $\mathcal{P}_1$ is
  interface $pc_1 : \{outC, reqC, inC\};\ x_1 : \mathbb{B}$
  external $pc_2 : \{outC, reqC, inC\};\ x_2 : \mathbb{B}$
  lazy atom
    controls $pc_1, x_1$
    reads $pc_1, pc_2, x_1, x_2$
    init
      $[\!]\ true \rightarrow pc_1' := outC$
    update weaklyfair $\alpha_1, \beta_1$
      $[\!]\ pc_1 = outC$                                        $\rightarrow pc_1' := reqC;\ x_1' := x_2$
      $[\!]\ pc_1 = reqC \wedge (pc_2 = outC \vee x_1 \neq x_2)$ $\xrightarrow{\beta_1} pc_1' := inC$
      $[\!]\ pc_1 = inC$                                         $\xrightarrow{\alpha_1} pc_1' := outC$

module $\mathcal{P}_2$ is
  interface $pc_2 : \{outC, reqC, inC\};\ x_2 : \mathbb{B}$
  external $pc_1 : \{outC, reqC, inC\};\ x_1 : \mathbb{B}$
  lazy atom
    controls $pc_2, x_2$
    reads $pc_1, pc_2, x_1, x_2$
    init
      $[\!]\ true \rightarrow pc_2' := outC$
    update weaklyfair $\alpha_2, \beta_2$
      $[\!]\ pc_2 = outC$                                        $\rightarrow pc_2' := reqC;\ x_2' := \neg x_1$
      $[\!]\ pc_2 = reqC \wedge (pc_1 = outC \vee x_1 = x_2)$    $\xrightarrow{\beta_2} pc_2' := inC$
      $[\!]\ pc_2 = inC$                                         $\xrightarrow{\alpha_2} pc_2' := outC$

$FairPete\ =\ $ hide $x_1, x_2$ in $\mathcal{P}_1 \,\|\, \mathcal{P}_2$
```

Figure 9.5: Fair asynchronous mutual exclusion

```
module $FairAnd$ is
  private $pc\colon \{stable, unstable, hazard\}$
  interface $out\colon \mathbb{B}$
  external $in_1, in_2\colon \mathbb{B}$
  lazy atom controls $out$ reads $pc, out$
    update weaklyfair $\alpha$
      $[\!]$ $pc = unstable \xrightarrow{\alpha} out' := \neg out$
      $[\!]$ $pc = hazard \rightarrow out' := \neg out$
  passive atom controls $pc$ reads $pc, out$ awaits $in'_1, in'_2, out'$
    init
      $[\!]$ $\textsc{And}(in'_1, in'_2, out') \rightarrow pc' := stable$
      $[\!]$ $\neg\textsc{And}(in'_1, in'_2, out') \rightarrow pc' := unstable$
    update
      $[\!]$ $pc = stable \wedge \neg\textsc{And}(in'_1, in'_2, out') \rightarrow pc' := unstable$
      $[\!]$ $pc = unstable \wedge \textsc{And}(in'_1, in'_2, out') \wedge out' \neq out \rightarrow pc' := stable$
      $[\!]$ $pc = unstable \wedge \textsc{And}(in'_1, in'_2, out) \wedge out' = out \rightarrow pc' := hazard$
```

Figure 9.6: Fair asynchronous And gate

for the update choice to toggle the output when the state variable *pc* equals the unstable value.

9.5.3 Message-passing Protocols

Our canonical example of a distributed system of agents communicating via messages consists of a sender and a receiver.

Synchronous communication

Let us first consider the module *SyncSender* of Figure 1.25. The messages are produced by the asynchronous atom *Producer*, which requires an unknown number of rounds to produce a message. The atom *Producer* has two choices, one to produce a message and another to stutter, available in every round. We do not impose any fairness on the resolution of this choice, because a scenario in which no message is ever produced is acceptable. On the other hand, consider the atom *Consumer* of the receiver that consumes the messages. Again, the atom *Consumer* is asynchronous, and requires an unknown number of rounds to consume a message. We would like to use fairness to ensure that once the message is ready to be consumed ($pc = consume$), the event $done_C$ is eventually issued and the message is copied into msg_C. Weak fairness suffices for this purpose. Similarly, we use fairness to rule out a scenario in which the receiver is ready for reception of the message ($pc = receive$), but delays issuing the

```
module $FairReceiver$ is
  private $pc\colon \{receive, consume\};\ msg_R\colon\ ;\ done_C\colon$
  interface $ready\colon\ ;\ msg_C\colon$
  external $transmit\colon\ ;\ msg_S\colon$
  passive atom
    controls $pc, msg_R$
    reads $pc, transmit, done_C$
    awaits $transmit', msg'_S, done'_C$
    init
      $\|\ true \rightarrow pc' := receive$
    update
      $\|\ pc = receive \wedge transmit? \rightarrow msg'_R := msg'_S;\ pc' := consume$
      $\|\ pc = consume \wedge done_C? \rightarrow pc' := receive$
  lazy atom controls $ready$
    update weaklyfair $\alpha$
      $\|\ pc = receive \xrightarrow{\alpha} ready!$
  lazy atom $Consumer$
    controls $done_C, msg_C$
    reads $pc, done_C, msg_R$
    update weaklyfair $\beta$
      $\|\ pc = consume \xrightarrow{\beta} done_C!;\ msg'_C := msg_R$
```

FairSyncMsg = **hide** *ready*, *transmit*, msg_S **in** *SyncSender* ‖ *FairReceiver*

Figure 9.7: Fair synchronous message passing

event *ready* forever. The receiver with these fairness assumptions is shown in Figure 9.7. The fair module *FairSyncMsg* is obtained by composing the fair receiver with *SyncSender* and hiding the variables used for communication.

Exercise 9.19 {P2} [Fair asynchronous sender] What are the appropriate fairness constraints for the module *AsyncSender* of Figure 1.30. Define the fair version of the modules *AsyncSender* and *AsyncMsg*. ■

Exercise 9.20 {P3} [Fair timed message passing] The protocol for timed message passing (Figure 1.32) refers to the external clock *AsyncClock* for measuring time. Define a fair version of the clock *AsyncClock* so that every ω-trajectory contains infinitely many *tick* events. Define a fair version of the module *TimedMsg* by adding appropriate fairness assumptions to *TimedSender* and *TimedReceiver*. ■

```
module $FairStick$ is
  private $pc\colon \{free, left, right\}$
  interface $grant_L, grant_R\colon$
  external $req_L, release_L, req_R, release_R\colon$
  passive atom
    controls $pc, grant_L, grant_R$
    reads $pc, req_L, grant_L, release_L, req_R, grant_R, release_R$
    awaits $req'_L, release'_L, req'_R, release'_R$
    init
      $[\!]\ true \rightarrow pc' := free$
    update stronglyfair $\alpha_L, \alpha_R$
      $[\!]\ pc = free \wedge req_L? \overset{\alpha_L}{\rightarrow} grant_L!;\ pc' := left$
      $[\!]\ pc = free \wedge req_R? \overset{\alpha_R}{\rightarrow} grant_R!;\ pc' := right$
      $[\!]\ pc = left \wedge release_L? \rightarrow pc' := free$
      $[\!]\ pc = right \wedge release_R? \rightarrow pc' := free$
```

Figure 9.8: A fair chopstick for the dining philosophers

Dining philosophers

Recall the problem of dining philosophers from Exercise 1.13, and consider the module *Stick*. When both the philosophers on the two sides of a chopstick request the chopstick simultaneously, it is granted to either one of them nondeterministically. To model the assumption that if some philosopher requests a particular chopstick repeatedly, it is eventually granted, we can use strong fairness. Such a fair chopstick is shown in Figure 9.8. Observe that even if the philosopher reissues the request in every round until the chopstick is granted, the choice to grant the request is not always available (due to the conjunct $pc = free$ in the guard). Consequently, the weaker assumption that some philosopher requests a particular chopstick *continuously*, it is eventually granted, is not captured by requiring weak fairness for the choices α_L and α_R.

Exercise 9.21 {P3} [Fair dining philosophers] Consider your definitions of the modules that describe philosophers, together with guards, in Exercise 1.13. Replace the chopstick by the fair chopstick of Figure 9.8. Add fairness assumptions to each philosopher to ensure that once the philosopher has both the chopsticks, (s)he will eventually release both of them. Define the fair module *FairDine4* consisting of four copies of fair chopsticks and fair philosophers. Does your solution satisfy starvation-freedom, that is, is there fair trajectory of *FairDine4* in which some philosopher, after some round, waits forever? ■

Contents

Chapter 10

Response Verification

10.1 Response Requirements

10.1.1 The fair emptiness problem

The basic problem in the analysis of transition graphs is to determine whether some state in a target region is reachable. The corresponding basic problem in the analysis of fair graphs is to determine whether a fair graph has an initialized fair trajectory.

Fair emptiness problem

An instance of the fair-emptiness problem is a fair graph $\mathcal{G}$. The answer to the fair-emptiness problem $\mathcal{G}$ is Yes if the ω-language $\mathcal{L}_{\mathcal{G}}$ is nonempty, and No otherwise.

Remark 10.1 [Fair emptiness for machine-closed graphs] If $\mathcal{G}$ is a machine-closed fair graph then the answer to the fair-emptiness problem $\mathcal{G}$ is Yes. ■

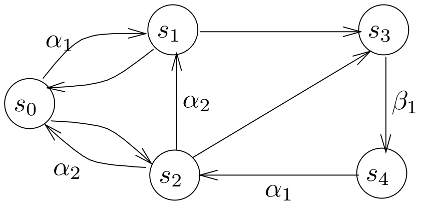


Figure 10.1: Fair Cycles

The fair cycle problem

To establish that the ω-language of a fair graph is empty, we need to find an initialized fair trajectory. Our algorithms will search for initialized fair trajectories of a special form, namely, the eventually periodic ones. This is because eventually periodic trajectories can be represented in a finite manner using cycles.

A *cycle* of the transition graph G is a trajectory $\overline{s}_{0..m}$ such that $s_m = s_0$. Fairness of a cycle with respect to an action, fairness constraint, and a fairness assumption is defined by considering all the actions involved in the cycle.

Fair cycle

Let G be a transition graph. The cycle $\overline{s}_{0..m}$ of G is *α-fair*, for an action α of G, if $s_i \xrightarrow{\alpha} s_{i+1}$ for some $0 \leq i < m$. The cycle $\overline{s}_{0..m}$ is f-fair for a fairness constraint $f = (\alpha, \beta)$ of G, if it is either β-fair or not α-fair. The cycle $\overline{s}_{0..m}$ is F-fair, for a fairness assumption F of G, if it is f-fair for all fairness constraints f in F.

A *fair cycle* of the fair graph (G, F) is a F-fair cycle of G.

Remark 10.2 [Fair trajectory of a fair cycle] If $\overline{s}_{0...m}$ is a fair cycle of the fair graph $\mathcal{G}$, then the periodic ω-trajectory $(\overline{s}_{0...m-1})^{\omega}$ is a fair trajectory of $\mathcal{G}$. ■

Example 10.1 [Fair cycles] Consider the fair graph of Figure 10.1 with $F = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2)\}$. The cycle $s_0 s_1 s_3 s_4 s_2 s_0$ is not F-fair. Similarly, the cycle $s_1 s_3 s_4 s_2 s_1$ is not F-fair. On the other hand, the cycle $s_2 s_3 s_4 s_2$ is F-fair. ■

To solve the fair-emptiness problem, we look for fair cycles that are reachable. The cycle $\overline{s}_{0..m}$ of a transition graph G is reachable if the state s_0 is reachable, or equivalently, each s_i, for $0 \leq i \leq m$, belongs to the reachable region σ^R of G.

FAIR CYCLE PROBLEM

An instance of the fair-cycle problem is a fair graph $\mathcal{G}$. The answer to the fair-cycle problem is YES if there exists a reachable fair cycle of $\mathcal{G}$, and NO otherwise. A *witness* for the fair-cycle problem $\mathcal{G}$ consists of (1) an initialized trajectory $\overline{s}_{0\ldots m}$ of $\mathcal{G}$, and (2) a source-s_m fair cycle $\overline{t}_{0\ldots k}$ of $\mathcal{G}$.

Consider a fair graph $\mathcal{G}$. Suppose the answer to fair-cycle problem is YES, and let $(\overline{s}_{0\ldots m}, \overline{t}_{0\ldots k})$ be a witness to the fair-cycle problem. Then, the eventually periodic ω-trajectory $\overline{s}_{0\ldots m-1}(\overline{t}_{0\ldots k-1})^\omega$ is an initialized fair trajectory of $\mathcal{G}$, and thus, the answer to the fair-emptiness problem $\mathcal{G}$ is YES. Conversely, existence of an initialized fair trajectory guarantees the existence of a reachable fair cycle, provided that the graph is finite.

Proposition 10.1 [Fair cycle vs. fair trajectory] *Let $\mathcal{G}$ be a finite fair graph, and let s be a state of $\mathcal{G}$. Then, there exists a source-s fair cycle of $\mathcal{G}$ iff there exists a source-s fair trajectory of $\mathcal{G}$.*

Exercise 10.1 {T2} [Fair cycle vs. fair trajectory] (1) Prove Proposition 10.1. (2) Show that Proposition 10.1 does not hold for infinite fair graphs. ■

Corollary 10.1 [Fair emptiness vs. fair cycle problems] *For a finite fair graph $\mathcal{G}$, the answer to the fair-emptiness problem $\mathcal{G}$ coincides with the answer to the fair-cycle problem $\mathcal{G}$.*

10.1.2 The recurrence verification problem

The basic problem in the analysis of modules is invariant verification which asks whether a given predicate is an invariant of the module. The basic problem in the analysis of fair modules is recurrence verification.

RECURRENT

Let $\mathcal{P}$ be a fair module, and let p be an observation predicate for $\mathcal{P}$. The predicate p is a *recurrent* of $\mathcal{P}$ if every fair ω-trajectory of $\mathcal{P}$ is p-fair

In other words, given fair module $\mathcal{P}$ with fairness assumption F, the observation predicate p is a recurrent of $\mathcal{P}$ iff every fair trajectory of $\mathcal{P}$ is guaranteed to contain infinitely many p-states: for every F-fair ω-trajectory $\underline{s}$, for infinitely many positions i, the state s_i satisfies the observation predicate p.

RECURRENCE-VERIFICATION PROBLEM

An instance $(\mathcal{P}, p)$ of the *recurrence-verification problem* consists of (1) a fair module $\mathcal{P}$ and (2) an observation predicate p for $\mathcal{P}$. The answer to the recurrence-verification problem is YES if p is a recurrent of $\mathcal{P}$, and otherwise NO.

Example 10.2 [Starvation freedom of mutual exclusion] Let us revisit the mutual exclusion problem from Chapter 1. A liveness requirement for the mutual exclusion algorithms is the starvation freedom property which asserts that if a process requests the critical section, then that process eventually enters the critical section. Requiring starvation freedom rules out solutions that always prefer one process over the other. Starvation freedom for process i corresponds to checking whether the predicate $pc_i \neq reqC$ is recurrent. Verify that the answers to both the recurrence-verification problems $(SyncMutex, pc_1 \neq reqC)$ and $(FairPete, pc_2 \neq reqC)$ are YES. ■

Exercise 10.2 {P2} [Starvation freedom for railroad controller] Recall the railroad example from Chapter 2. Show that the requirement that a train does not wait at the signal forever can be formulated as a recurrence-verification problem. Does *RailroadSystem* satisfy the requirement? ■

From recurrence verification to fair emptiness

The answer to the recurrence-verification problem $(\mathcal{P}, p)$ is NO when there is a fair trajectory of $\mathcal{P}$ that is not $[\![p]\!]$-fair. Observe that an ω-trajectory is not σ-fair, for a region σ, iff it is $(\sigma, \emptyset)$-fair. Consequently, to solve recurrence verification problem $(\mathcal{P}, p)$ we consider the fair graph $\mathcal{G}_{\mathcal{P}}$, add the region-constraint $([\![p]\!], \emptyset)$, and check whether the resulting fair graph has an initialized fair trajectory. For an instance $(\mathcal{P}, p)$ of the recurrence-verification problem, the fair graph $(G_{\mathcal{P}}, F \cup \{([\![p]\!], \emptyset)\}$ is denoted $\mathcal{G}_{\mathcal{P},p}$.

Proposition 10.2 [Recurrence-verification to fair emptiness] *The answer to the recurrence-verification problem* $(\mathcal{P}, p)$ *is* YES *iff the answer to the fair emptiness problem* $\mathcal{G}_{\mathcal{P},p}$ *is* NO.

As discussed earlier, to solve a fair emptiness problem, we solve the corresponding fair cycle problem. Thus, to solve the recurrence-verification problem $(\mathcal{P}, p)$, we solve the fair-cycle problem $\mathcal{G}_{\mathcal{P},p}$ If the answer to fair cycle problem is YES, the answer to the recurrence-verification problem is NO; if the answer to the fair-cycle problem is YES, and the module is finite, then the answer to the recurrence-verification problem is YES.

When the answer to the fair-cycle problem $\mathcal{G}_{\mathcal{P},p}$ is YES, the corresponding witness can be reported as an *error trajectory* for the recurrence-verification problem. It consists of an initialized trajectory $\overline{s}_{0\ldots m}$ of $\mathcal{P}$, and a source-s_m fair cycle $\overline{t}_{0\ldots k}$ of $\mathcal{P}$ such that t_i does not satisfy p for all $0 \leq i \leq k$. The fair cycle $\overline{t}_{0\ldots k}$ corresponds to a loop in which all fairness requirements of $\mathcal{P}$ are satisfied, but the predicate p never holds. Thus, it corresponds to a "bad" cycle in the execution. The initialized trajectory $\overline{s}_{0\ldots m}$ is an evidence that the bad cycle is reachable. Together, they provide useful information for debugging.

Exercise 10.3 {T3} [Eventual Invariants] Let $\mathcal{P}$ be a fair module, and let p be an observation predicate for $\mathcal{P}$. The observation predicate is an *eventual invariant* of $\mathcal{P}$ if for every fair trajectory $\underline{s}$ of $\mathcal{P}$, for some $i \geq 0$, $s_i \models p$ for all $j \geq i$. Thus, p is an eventual invariant of $\mathcal{P}$ if every fair trajectory of $\mathcal{P}$ has a suffix all of whose states satisfy p. The eventual-invariant verification problem is to check, given a fair module $\mathcal{P}$ and an observation predicate p for $\mathcal{P}$, whether or not p is an eventual invariant of $\mathcal{P}$.

Show that the eventual-invariant verification problem $(\mathcal{P}, p)$ can be reduced to fair emptiness problem. ■

10.1.3 The response verification problem

An observation predicate p is a recurrent of a fair module $\mathcal{P}$ if every fair trajectory contains infinitely many p-states. A more general and common requirement is the response requirement that stipulates that every request be followed by an eventual response.

RESPONSE

Let $\mathcal{P}$ be a fair module, and let p and q be two observation predicates of $\mathcal{P}$. The predicate q is said to be a *response* to the predicate p in $\mathcal{P}$, denoted $p \leadsto_{\mathcal{P}} q$, if for every fair trajectory $\underline{s}$ of $\mathcal{P}$, for all $i \geq 0$, if $s_i \models p$ then for some $j \geq i$, $s_i \models q$.

Intuitively, if p denotes a request, and q denotes the fulfillment of the request, then "q is a response to p" corresponds to "every request is eventually fulfilled." In particular, every fair trajectory that contains infinitely many p-states must contain infinitely many q-states.

Remark 10.3 [Transitivity of response] Let $\mathcal{P} = (P, F)$ be a fair module, and let p, q, and r be observation predicates of $\mathcal{P}$. (1) If $p \leadsto_P q$ then $p \leadsto_{\mathcal{P}} q$. (2) If $p \rightarrow q$ is an invariant of P (i.e., for every reachable state s of P, if s satisfies p, then s satisfies q) then $p \leadsto_P q$. (3) If $p \leadsto_{\mathcal{P}} q$ and $q \leadsto_{\mathcal{P}} r$, then $p \leadsto_{\mathcal{P}} r$. ■

RESPONSE VERIFICATION

An instance $(\mathcal{P}, p, q)$ of the *response-verification problem* consists of (1) a fair module $\mathcal{P}$, (2) [request predicate] an observation predicate p for $\mathcal{P}$, and (3) [response predicate] an observation predicate q for $\mathcal{P}$. The answer to the response-verification problem $(\mathcal{P}, p, q)$ is YES if $p \leadsto_{\mathcal{P}} q$, and otherwise NO.

Remark 10.4 [Invariant verification as response verification] Let P be a module, and let p be an observation predicate of P. The predicate p is an invariant of the module P iff $\neg p \leadsto_P \mathit{false}$. Thus, the invariant verification problem (P, p) reduces to the response verification problem $(P, \neg p, \mathit{false})$. ■

Remark 10.5 [Recurrence verification as response verification] Let $\mathcal{P}$ be a fair module, and let p be an observation predicate of $\mathcal{P}$. The predicate p is a recurrent of $\mathcal{P}$ iff $\mathit{true} \leadsto_P p$. Thus, the recurrence-verification problem (P, p) reduces to the response verification problem (P, true, p). ■

Example 10.3 [Mutual exclusion] Let us revisit the mutual exclusion problem. The observation predicate

$$p_{\mathit{reqC}} \;=\; (\mathit{pc}_1 = \mathit{reqC}) \vee (\mathit{pc}_2 = \mathit{reqC})$$

characterizes the states in which some process is requesting the critical section. The observation predicate

$$p_{\mathit{inC}} \;=\; (\mathit{pc}_1 = \mathit{inC}) \vee (\mathit{pc}_2 = \mathit{inC})$$

characterizes the states in which some process is inside the critical section. Checking absence of deadlocks corresponds to checking whether p_{inC} is a response to p_{reqC}. Verify that the answer to the response verification problem $(\mathit{SyncMutex}, p_{\mathit{reqC}}, p_{\mathit{inC}})$ is Yes. The answer to the response verification problem $(\mathit{Pete}, p_{\mathit{reqC}}, p_{\mathit{inC}})$ is No (why?). However, along every fair trajectory of *FairPete* a state satisfying p_{reqC} is eventually followed by a state satisfying p_{inC}, and thus, the answer to the response verification problem $(\mathit{FairPete}, p_{\mathit{reqC}}, p_{\mathit{inC}})$ is Yes. Observe that requiring p_{inC} to be a response to p_{reqC} is a weaker requirement than requiring $\mathit{pc}_1 \neq \mathit{reqC}$ to be recurrent (why?). ■

Example 10.4 [Fair synchronous communication] Consider the fair module *FairSyncMsg* obtained by composing the fair receiver with *SyncSender* and hiding the variables used for communication (see Section 9.5). A response requirement for the module *FairSyncMsg* stipulates that every message produced by the sender is eventually consumed by the receiver. If v is a value of the type , then we can use $\mathit{msg}_P = v$ as the request predicate, and $\mathit{msg}_C = v$ as the response predicate. Verify that the answer to the response verification problem $(\mathit{FairSyncMsg}, \mathit{msg}_P = v, \mathit{msg}_C = v)$ is Yes. ■

Exercise 10.4 {P2} [Dining philosophers] Recall the fair version of the dining philosophers problem from Chapter 9. Formulate the starvation freedom requirement for an individual philosopher as a response verification problem. ■

From response-verification to recurrence-verification

Response-verification problem can be reduced to a recurrence-verification problem by adding monitors. Consider the response-verification problem $(\mathcal{P}, p, q)$.

Consider the reactive module *ResponseMonitor*:

```
module ResponseMonitor is
  external $p$, $q$
  private $alert\colon \mathbb{B}$
  atom controls $alert$ reads $p$, $q$
    init
      $[\!] \; true \rightarrow alert' := 0$
    update
      $[\!] \; alert = 0 \wedge p \wedge \neg q \rightarrow alert' := 1$
      $[\!] \; alert = 1 \wedge q \qquad\quad \rightarrow alert' := 0$
```

The monitor *ResponseMonitor* observes the behavior of the module $\mathcal{P}$ and updates its private state *alert* accordingly. In the description of *ResponseMonitor*, the declaration **external** p, q stands for **external** $x_1, x_2, \ldots x_k$, where $x_1, x_2, \ldots x_k$ are the variables appearing in the predicates p and q. The value of the private variable *alert* is initially 0. When a state that satisfies the request predicate p, but not the response predicate q, is encountered, the variable *alert* is updated to 1. Thus, $alert = 1$ indicates a situation in which the request has been issued, but the subsequent response has not been issued. Once a state satisfying the response predicate is encountered, the variable *alert* is reset to 0. The following proposition asserts that q is a response to p in $\mathcal{P}$ iff $alert = 0$ is a recurrent of the compound module $\mathcal{P} \parallel ResponseMonitor$.

Proposition 10.3 [From response-verification to recurrence-verification] *The answer to the response-verification problem* $(\mathcal{P}, p, q)$ *coincides with the answer to the recurrence-verification problem* $(\mathcal{P} \parallel ResponseMonitor, alert = 0)$.

Proof. Consider a ω-trajectory $\underline{s}$ of $\mathcal{P}$. Since the update of the state of *ResponseMonitor* is deterministic, and from the definition of the parallel composition, there exists a unique ω-trajectory $\underline{t}$ of $\mathcal{P} \parallel ResponseMonitor$ such that $s_i = X_{\mathcal{P}}[t_i]$ for all $i \geq 0$. Furthermore, $\underline{s}$ is a fair-trajectory of $\mathcal{P}$ iff $\underline{t}$ is a fair trajectory of $\mathcal{P} \parallel ResponseMonitor$.

From the initialization and update commands for the variable *alert*, for all $i \geq 0$, $alert[t_i] = 1$ iff there exists $j < i$ such that $s_j \models p$ and $s_k \not\models q$ for $i \leq k < j$. Consequently, every p-state is followed by an q-state along $\underline{s}$ (i.e. for all $i \geq 0$, if $s_i \models p$ then there some $j \geq i$, $s_j \models q$) iff for infinitely many indices $i \geq 0$, $alert[t_i] = 0$. Consequently, q is a response to p in $\mathcal{P}$ iff every fair trajectory of $\mathcal{P} \parallel ResponseMonitor$ is $(alert = 0)$-fair. ■

Exercise 10.5 {T2} [Response-verification for action predicates] In our formulation of the response-verification problem, the request and the response predicates define regions. Formulate a variant of the problem in which the request and the response predicates for a module P are predicates over $\mathsf{obs}X_P \cup \mathsf{obs}X'_P$, and

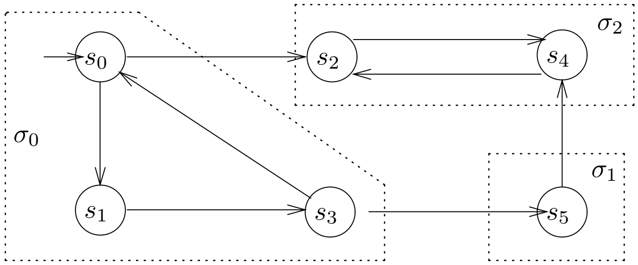


Figure 10.2: Strongly connected components

define actions of P. Show that this variant of the response-verification problem can be reduced to a recurrence-verification problem by adding an appropriate monitor. ■

10.2 Enumerative Search

In this section, we present a solution to the fair cycle problem based on classical enumerative graph-search algorithms. A naive way to solve the fair-cycle problem (G, F) would be to find all reachable cycles in the graph G, and check if any of them is F-fair. This is not efficient, since the number of cycles in a graph can be exponential in the number of its states. A more reasonable approach is to consider cycles of a special form, namely, the strongly connected components of the graph.

10.2.1 Strongly connected regions

Two states of a transition graph are strongly connected if there is a cycle that contains both of them. A strongly connected component is a maximal set of states that are strongly connected to one another.

Strongly connected component

Let G be a transition graph. Two states s and t of G are *strongly connected*, written $s \cong_{scc} t$, iff $s \in post^*(t)$ and $t \in post^*(s)$. A region σ of G is *strongly connected* if $s \cong_{scc} t$ for all states s and t in σ. A region σ of G is a *strongly connected component* if (1) σ is strongly connected, and (2) [maximality] no strongly connected region of G is a strict superset of σ.

Remark 10.6 [Strongly connected components] The relation $\cong_{scc}$ is an equivalence relation on the state space of G. A strongly connected component is an equivalence class of $\cong_{scc}$. ■

Example 10.5 [Strongly connected components] For the transition graph of Figure 10.1, there is a single strongly connected component σ that contains all the states. Consider the transition graph of Figure 10.2. It has three strongly connected components: $\sigma_0 = \{s_0, s_1, s_3\}$, $\sigma_1 = \{s_2, s_4\}$, and $\sigma_2 = \{s_5\}$. ■

Figure 10.3 shows an algorithm to compute the partition $\cong_{scc}$ in time linear in the number of states and transitions. The algorithm involves two depth first searches. The first search involves the function *DepthFirstSearch*. For every reachable state s, there is precisely one invocation of *DepthFirstSearch* with input s. Let us order states in σ^R according to the termination times of the corresponding invocations of *DepthFirstSearch*: with each state $s \in \sigma^R$, associate a number $1 \leq done_s \leq |\sigma^R|$ such that if *DepthFirstSearch*(s) terminates before *DepthFirstSearch*(t) then $done_s < done_t$. Verify that at the end of the first search, the stack E contains all reachable states ordered in reverse according to the numbering *done*.

For every state s, the *forefather* of s, denoted $forefather_s$, is the state $t \in post^*(s)$ such that for all $u \in post^*(s)$, $done_u \leq done_t$. Thus, forefather of a state s is the state for which *DepthFirstSearch* terminates last among all the states reachable from s. States belonging to the same strongly connected component share the forefather:

Lemma 10.1 [Forefather and strongly connected component] *For two states s and t in σ^R, $s \cong_{scc} t$ iff $forefather_s = forefather_t$.*

Exercise 10.6 {T3} [Forefather and strongly connected component] Prove Lemma 10.1. ■

Now consider the state s with the highest value of $done_s$. Clearly, s is its own forefather, and furthermore, if $s \in post^*(t)$, then $forefather_t = s$. By Lemma 10.1, the strongly connected component containing s contains precisely those states from which s is reachable, that is, states in $pre^*(s)$. The second depth first search begins by invoking *DFS2*(s) since s is on top of the stack, and searches the graph to compute $pre^*(s)$. When *DFS2*(s) terminates, the region τ equals $pre^*(s)$, and is the first strongly connected component. The remaining strongly connected components are computed in the same fashion.

Theorem 10.1 [Strongly connected components] *Let G be a finite transition graph. Algorithm 10.1 correctly computes the strongly connected components of the reachable region of G.*

Example 10.6 [Computation of strongly connected components] Figure 10.4 illustrates a possible execution of Algorithm 10.1 on the transition graph of Figure 10.2. The value of *done* and *forefather* is listed along with each state.

Algorithm 10.1 [Strongly connected components]

Input: a finite transition graph $G = (\Sigma, \sigma^I, \rightarrow)$.
Output: the partition π_{scc} of the reachable region σ^R of G into strongly connected components.

```
input $G$: enumgraph
local $\sigma, \tau$: enumreg; $E$: stack of state; $s$: state; $\pi$: partition
begin
  $\sigma := EmptySet$;
  $\pi := EmptySet$;
  $E := EmptyStack$;
  foreach $s$ in $InitQueue(G)$ do
    if not $IsMember(s, \sigma)$ then $DepthFirstSearch(s)$ fi
    od;
  $\sigma := EmptySet$;
  while not $EmptyStack(E)$ do
    $s := Top(E)$;
    $E := Pop(E)$;
    if not $IsMember(s, \sigma)$ then
      $\tau := EmptySet$;
      $DFS2(s)$;
      $\pi := Insert(\tau, \pi)$
      fi
    od;
  return $\pi$
  end.

function $DepthFirstSearch$
  input $s$: state;
  local $t$: state;
  begin
    $\sigma := Insert(s, \sigma)$;
    foreach $t$ in $PostQueue(s, G)$ do
      if not $IsMember(t, \sigma)$ then $DepthFirstSearch(t)$ fi
      od;
    $E := Push(s, E)$
    end.

function $DFS2$
  input $s$: state;
  local $t$: state;
  begin
    $\sigma := Insert(s, \sigma)$;
    $\tau := Insert(s, \tau)$;
    foreach $t$ in $PreQueue(s, G)$ do
      if not $IsMember(t, \sigma)$ then $DFS2(t)$ fi
      od;
    end.
```

Figure 10.3: Computing strongly connected components

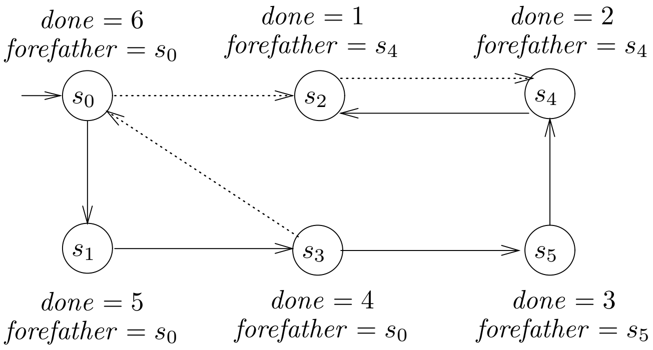


Figure 10.4: Computation of strongly connected components

The undotted edges correspond to the edges that lead to exploration of states for the first time. At the end of the first depth-first search, s_0 is on top of the stack E with $done_{s_0} = forefather_{s_0}$. The first strongly connected component equals $\sigma_0 = pre^*(s_0) = \{s_0, s_1, s_3\}$. Once $DFS2(s_0)$ terminates, the value of σ equals σ_0, and the state s_5 is on top of the stack with $done_{s_5} = forefather_{s_5}$. Consequently, the second strongly connected component equals $\sigma_2 = pre^*(s_5)\backslash\sigma_0 = \{s_5\}$. Once $DFS2(s_5)$ terminates, the value of σ equals $\sigma_0 \cup \sigma_2$, and the state s_4 is on top of the stack with $done_{s_4} = forefather_{s_4}$. Consequently, the third strongly connected component equals $\sigma_1 = pre^*(s_4)\backslash(\sigma_0 \cup \sigma_2) = \{s_2, s_4\}$. ■

Given an enumerative representation of the transition graph G, Algorithm 10.1 can be implemented so that it requires time $O(n + m)$, where n is the number of states in G and m is the number of transitions in G. For this purpose, in the second depth first search, the computation of *PreQueue* needs to be efficient. Given an adjacency list representation of a transition graph that gives, for every state s, a list of states in $post(s)$, one can construct, in linear time, a representation that gives, for every state s, a list of states in $pre(s)$.

The optimization techniques discussed in Section 2.3.4 can be applied to improve the performance of Algorithm 10.1. In particular, we can use an on-the-fly representation of a graph.

Exercise 10.7 {P3} [Computing predecessor region] Consider the on-the-fly representation of the transition graph G_P of a propositional module P. Write an algorithm for computing the operation *PreQueue* for the on-the-fly representation. What is the running time of your algorithm? ■

Exercise 10.8 {T3} [Tarjan's algorithm] Algorithm 10.1 uses two separate depth first searches to compute strongly connected components. An alternative strategy is to explicitly compute the forefather of every state. Then, the strongly connected components can be computed in the first search itself. During the call *DepthFirstSearch*(s), once all the successors of s are explored, the forefather of s can be computed from the forefathers of its successors. The set τ is used to store the current strongly connected component, and the set π is used to store all the strongly connected components computed so far. Once $\mathit{forefather}_s$ is determined, s is added to τ. If s is its own forefather, then we can conclude that τ is a strongly connected component, add it to π, and reinitialize τ to the empty region. Write an algorithm that implements this strategy. Unlike Algorithm 10.1, this strategy does not require implementation of the operation *PreQueue*, but it needs to store forefathers explicitly. In an on-the-fly implementation, this strategy would require $O(n \log n)$ memory (that gets accessed randomly) as opposed to $O(n)$ memory required by Algorithm 10.1. ■

10.2.2 Fair components

For a region σ of a transition graph G, a σ-cycle is a cycle $\overline{s}_{0..m}$ of G such that $s_i \in \sigma$ for each $0 \leq i \leq m$. A strongly connected region σ is *F-fair* for the fairness assumption F if there exists a F-fair σ-cycle. Since every strongly connected region is a subset of a strongly connected component, if σ is a F-fair strongly connected region, then the strongly connected component that contains σ is also F-fair.

Proposition 10.4 [Fair components] *A fair graph (G, F) has a reachable fair cycle iff some strongly connected component of the reachable subgraph is F-fair.*

Thus, to solve the fair cycle problem, it suffices to compute the strongly connected components of the reachable subgraph, and check if one of them is F-fair. For a single fairness constraint $f = (\alpha, \beta)$, the strongly connected component σ is f-fair precisely when either one of the following two conditions holds: (1) $(s, t) \in \beta$ for two states s and t in σ, or (2) there is a σ-cycle $\overline{s}_{0..m}$ such that $(s_i, s_{i+1}) \notin \alpha$ for all $0 \leq i < m$. Each of these conditions may be tested using a depth-first search of the region σ. For a fairness assumption with several constraints, checking the fairness of a strongly connected component is trickier.

Exercise 10.9 {T3} [Fair regions] Let (G, F) be a finite fair graph, and σ be a strongly connected component of G. Prove or disprove the claim: σ is F-fair iff σ is f-fair for each fairness constraint $f \in F$. Is the claim true when F is local? ■

Consider a strongly connected component σ of G. If σ is β-fair for every (α, β) in F then σ is F-fair. Let $f = (\alpha, \beta)$ be a fairness constraint in F. If there are no two states s and t in σ such that $s \xrightarrow{\beta} t$, then σ cannot contain a β-fair cycle. Consequently, for a σ-cycle to be F-fair, it needs to be α-unfair. This implies that to search for F-fair σ-cycles, we can delete all the transitions in α. After this transformation we no longer need to consider the constraint f. However,

Algorithm 10.2 [Fair strongly connected components]

```
function $FSCC$
Input: a finite transition graph $G = (\Sigma, \sigma^I, \rightarrow)$, and a fairness as-
    sumption $F$ for $G$.
Output: YES if there is a $F$-fair strongly connected component of
    $G$, and NO otherwise.
foreach $\sigma \in SCC(G)$ do
  $\rightarrow' := \{(s,t) \mid s \rightarrow t$ and $s, t \in \sigma\}$;
  $F' := \emptyset$;
  if $\rightarrow' \neq \emptyset$ then
    foreach $(\alpha, \beta) \in F$ do
      if $\rightarrow' \cap \beta \neq \emptyset$
        then $F' := Insert((\alpha, \beta), F')$
        else $\rightarrow' := \rightarrow' \setminus \alpha$
        fi
      od;
    if $F' = F$
      then return YES
      else if $FSCC(\sigma, \sigma, \rightarrow', F') =$ YES then return YES fi
      fi
  od;
return NO.
```

Figure 10.5: Computing fair strongly connected components

following this transformation, σ may no longer be strongly connected, hence, we need to compute strongly connected components of σ again. Algorithm 10.5 presents a recursive scheme that implements this strategy.

The fair components are computed by the recursive function *FSCC*. It uses the subroutine *SCC* that, given an input transition graph, returns the list of its strongly connected components. The function *SCC* can be implemented using, for instance, Algorithm 10.1.

Proposition 10.5 [Fair components] *Let G be a transition graph with finitely many reachable states, and let F be a fairness assumption for G. Algorithm 10.2 computes the F-fair strongly connected components of G.*

Example 10.7 [Computation of fair components] Consider the transition graph of Figure 10.1. The fairness assumption F contains two fairness constraints

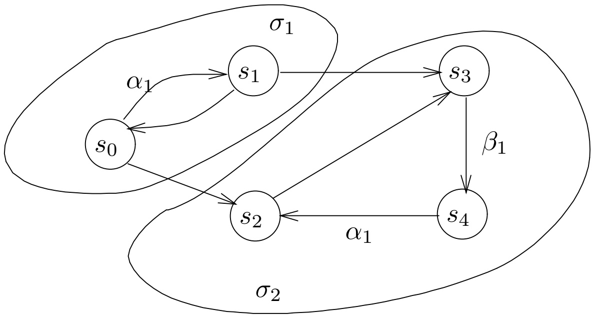


Figure 10.6: Computation of fair components

(α_1, β_1) and (α_2, β_2). The original graph contains a single strongly-connected component σ. Since σ does not contain any transition in β_2, all the transitions in α_2 are deleted along with the fairness constraint (α_2, β_2). Consequently, *FSCC* calls itself recursively on the graph shown in Figure 10.6 with a single fairness constraint (α_1, β_1). Observe that this graph is no longer strongly connected, and has two strongly connected components $\sigma_1 = \{s_0, s_1\}$, and $\sigma_2 = \{s_2, s_3, s_4\}$.

The component σ_1 does not contain any transition in β_1. So the algorithm deletes the α_1-transition from s_0 to s_1. A recursive call would split σ_1 into two singleton components $\{s_0\}$ and $\{s_1\}$, and since both of these do not contain any transitions, $\mathit{FSCC}(\{s_0, s_1\}, \{s_0, s_1\}, \{(s_1, s_0)\}, \emptyset)$ returns No.

The component σ_2 contains a transition in β_1, and hence, is fair. ■

Exercise 10.10 {P3} [Fair cycle computation] Write an algorithm to solve the fair-cycle problem. The input to the algorithm is a finite fair graph $\mathcal{G}$, and if the answer to the fair-cycle problem $\mathcal{G}$ is Yes, it should return a witness. ■

Observe that in Algorithm 10.2, every time *FSCC* calls itself recursively, at least one fairness constraint is removed. Consequently, the depth of recursion is bounded by the number of fairness constraints in the original fairness assumption.

One possible way to implement Algorithm 10.2 is to employ an enumerative representation of the transition graph, and to represent each action of each fairness constraint as a list of pairs of states.

Theorem 10.2 [Enumerative fair components] *Let G be a finite transition graph with n states and m transitions, and let F be a fairness assumption with ℓ constraints. Algorithm 10.2 computes the F-fair strongly connected components of G in time $O((n + m) \cdot \ell^2)$.*

As in case of invariant verification, instead of computing the transition graph of a module a priori, one can use an on-the-fly representation. Each fairness constraint of a fair module is represented by its update choice and its type (i.e. weak versus strong). The request and the response regions are represented by the corresponding predicates. The computation within the function *FSCC* involves deletion of transitions. Instead of performing this deletion explicitly, it suffices to remember which actions have been deleted, and pass this information to *SCC*.

10.3 Recurrence Verification for Weak Fairness

Consider the recurrence verification problem $(\mathcal{P}, p)$ where the module $\mathcal{P}$ has only weak-fairness constraints. Then, the graph $\mathcal{G}_{\mathcal{P},p}$ has weak-fairness constraints with an additional constraint $([\![p]\!], \emptyset)$. In this case, the fair-cycle problem can solved more efficiently without explicitly computing the strongly-connected components.

10.3.1 Single Büchi constraint

A weak-fairness constraint is of the form $(\rightarrow, \alpha)$. When the action α is specified by a region, then the weak-fairness constraint is called Büchi constraint. Thus, a Büchi constraint for a transition graph G is specified by a region σ^T of G, and an ω-trajectory $\underline{s}$ is fair if it is σ^T-fair (i.e. $s_i \in \sigma^T$ for infinitely many $i \geq 0$).

Let us consider the special case of the fair-cycle problem (G, F) when the fairness assumption F contains a single Büchi constraint specified by the region σ^T. The fair-cycle problem is, thus, to decide if there exists a reachable cycle that contains some state in σ^T. One solution is to first compute the reachable strongly connected components of G, and then check if the region σ^T has nonempty intersection with some strongly connected component. However, in this special case, there is no need to explicitly compute the strongly connected components. The algorithm of Figure 10.7 presents an improved solution.

The algorithms involves two nested searches, a primary search performed by the function *DepthFirstSearch* and a secondary (or nested) search performed by the function *NDFS*. The states encountered during the primary search are stored in the set σ, while the states visited during the secondary search are stored in the set τ. As in a standard depth first search, for every reachable state s of G, the function *DepthFirstSearch* is invoked at most once with input state s. Once the primary search originating at s terminates, if the state s belongs to the target region σ^T, then a secondary search is initiated by calling *NDFS* with input s. The objective of this secondary search is to find a cycle starting at the state s. When *NDFS*(s) is invoked, the stack E contains an initialized trajectory leading to state s. Thus, if the secondary search visits a state belonging to the stack, then it concludes that there is a cycle that contains s. This establishes that whenever the algorithm returns the answer YES the graph contains a reachable cycle containing a state in σ^T.

Let us again order the states according the termination times of the primary search: with each state $s \in \sigma^R$, associate a number $1 \leq done_s \leq |\sigma^R|$ such that if *DepthFirstSearch*(s) terminates before *DepthFirstSearch*(t) then $done_s < done_t$. Suppose the graph G contains a reachable cycle containing some state in the target region σ^T. Let $s_0, \ldots s_k$ be the ordering of states in $\sigma^T \cap \sigma^R$ according to the numbering *done*. Let s_i be the first state in this ordering that belongs to a cycle, and let $\upsilon = \cup 0 \leq j < i.\, post^*(s_j)$. Verify that s_i does not belong to υ (otherwise, there is a cycle containing some s_j for $j < i$). In fact, the cycle that contains s_i is disjoint from υ. When the primary search from state s_i is over, the set τ containing the states visited by the secondary search so far equals υ. Consequently, *NDFS* will be invoked with input s_i, and will find a cycle containing s_i.

Proposition 10.6 [Nested search for single Büchi] *Let G be a transition graph with finitely many reachable states, and let σ^T be a region of G. Then Algorithm 10.3 solves the fair cycle problem $(G, \{(\Sigma, \sigma^T)\})$.*

For complexity analysis of Algorithm 10.3, for a reachable state s, both the routines *DepthFirstSearch* and *NDFS* are invoked at most once with input s. Using a standard enumerative representation, the algorithm can be implemented with linear running time.

Theorem 10.3 [Fair-cycle for single Büchi] *Let G be a finite transition graph, and let σ^T be a region of G. Given the input $\{G\}_e$ and $\{\sigma\}_e$, Algorithm 10.3 solves the fair-cycle problem $(G, \{(\Sigma, \sigma^T)\})$ in $O(n+m)$ time and $\Theta(n+m)$ space, where n is the number of states and m is the number of transitions of G.*

Observe that, unlike the solution to the fair cycle problem from the previous section, Algorithm 10.3 does not involve computation of *PreQueue*, and may terminate even before visiting all the reachable states of the graph.

Exercise 10.11 {T3} [Witness reporting] Given an input graph G and a region σ^T, suppose Algorithm 10.3 terminates with answer YES. Let $\overline{s}_{0\ldots m}$ be the contents of the stack E, in reverse order, upon termination of the algorithm. Show that $\overline{s}_{0\ldots m}$ is an initialized trajectory of G, and $s_i \in post(s_m)$ for some $0 \leq i \leq m$. Modify Algorithm 10.3 so that it returns either the answer NO or a witness to the fair-cycle problem. ■

Exercise 10.12 {P3} [Nested DFS for a single weak-fairness constraint] Algorithm 10.3 solves the fair-cycle problem for a single Büchi constraint specified

Algorithm 10.3 [Nested Depth-first Search for Single Büchi]

```
Input: a finitely branching transition graph $G$, and a finite region
  $\sigma^T$ of $G$.
Output: the answer to the fair-cycle problem $(G, \{(\Sigma, \sigma^T)\})$
input $G$: enumgraph; $\sigma^T$: enumreg;
local $\sigma, \tau$: enumreg; $E$: stack of state; $s$: state
begin
  $\sigma := EmptySet$; $\tau := EmptySet$; $E := EmptyStack$;
  foreach $s$ in $InitQueue(G)$ do
    if not $IsMember(s, \sigma)$ then
      if $DepthFirstSearch(s)$ then return YES fi;
    fi;
  od;
  return NO
  end.
function $DepthFirstSearch$: $\mathbb{B}$
  input $s$: state;
  local $t$: state;
  begin
    $E := Push(s, E)$; $\sigma := Insert(s, \sigma)$;
    foreach $t$ in $PostQueue(s, G)$ do
      if not $IsMember(t, \sigma)$ then
        if $DepthFirstSearch(t)$ then return $true$ fi;
        fi;
        od;
    if $IsMember(s, \sigma^T)$ and not $IsMember(s, \tau)$ then
      if $NDFS(s)$ then return $true$ fi
      fi;
    $E := Pop(E)$;
    return $false$
    end.
function $NDFS$: $\mathbb{B}$
  input $s$: state;
  local $t$: state;
  begin
    $\tau := Insert(s, \tau)$;
    foreach $t$ in $PostQueue(s, G)$ do
      if $IsMember(t, E)$ then return $true$ fi;
      if not $IsMember(t, \tau)$ then
        if $NDFS(t)$ then return $true$ fi;
        fi;
      od;
    return $false$
    end.
```

Figure 10.7: Nested search for a fair cycle with a single Büchi constraint

by a region. Modify the algorithm to solve the fair-cycle problem for a single weak-fairness constraint. That is, write an algorithm that takes as input, a transition graph G and an action α of G, returns the answer to the fair-cycle problem $(G, \{(\rightarrow, \alpha)\})$. ■

10.3.2 Multiple weak-fairness constraints

Now, let us consider the fair-cycle problem (G, F) when F is a weak-fairness assumption. It is possible to translate this problem to a fair-cycle problem for a single weak-fairness constraint by augmenting the states of G with a counter variable. Let $\alpha_1, \ldots \alpha_\ell$ be an enumeration of the actions in F. The counter is initially 1, and is incremented from i to $i+1$, treating $\ell+1 = 1$, when transition in α_i is encountered. Visiting all of α_i infinitely often corresponds to updating the counter from ℓ to 1 infinitely often.

From multiple weak constraints to a single weak constraint

Let $\mathcal{G} = (\Sigma, \sigma^I, \rightarrow, F)$ be a weakly-fair graph, where F is a weak-fair assumption with ℓ weak-fair constraints specified by actions $\alpha_1 \ldots \alpha_\ell$. The fair graph $B_\mathcal{G}$ has the following components:

- for every state s of $\mathcal{G}$, and for every $1 \leq i \leq \ell$, the pair (s, i) is a state of $B_\mathcal{G}$;
- for every initial state s of $\mathcal{G}$, the pair $(s, 1)$ is an initial state of $B_\mathcal{G}$;
- for every action α_i in F, for every transition $s \xrightarrow{\alpha_i} t$ in α_i, if $i < \ell$ then $B_\mathcal{G}$ has a transition from (s, i) to $(t, i+1)$, and if $i = \ell$ then $B_\mathcal{G}$ has a transition from (s, i) to $(t, 1)$;
- the fairness assumption of $B_\mathcal{G}$ contains a single weak constraint α: for every transition $s \xrightarrow{\alpha_\ell} t$, $(s, \ell) \xrightarrow{\alpha} (t, 1)$.

Proposition 10.7 [Multiple weak-fairness to single weak constraint] *For every weakly-fair graph $\mathcal{G}$, the answer to the fair-cycle problem $\mathcal{G}$ coincides with the answer to the fair-cycle problem $B_\mathcal{G}$.*

Exercise 10.13 {T2} [Multiple weak-fairness to single weak constraint] Prove Proposition 10.7. ■

If the weakly-fair graph $\mathcal{G}$ has n states, m transitions, and ℓ fairness constraints, the weakly-fair graph $B_\mathcal{G}$ has $n \cdot \ell$ states and $m \cdot \ell$ transitions. It follows that the fair-cycle problem $\mathcal{G}$ can be solved in time $O((n+m) \cdot \ell)$. Contrast this with the complexity $O((n+m) \cdot \ell^2)$ of solving the fair-cycle problem in the general case using Algorithm 10.2.

10.3.3 Recurrence verification

Now we are ready to present the solution to the recurrence-verification problem $(\mathcal{P}, p)$ when $\mathcal{P}$ has only weak-fairness constraints. In this case, to answer positively to the fair-cycle problem, the algorithm needs to find a reachable cycle that contains at least one transition from each of the weak-fairness constraints of $\mathcal{P}$, and does not contain any p-state. This can be formulated as searching for a F-fair $[\![\neg p]\!]$-cycle. Figure 10.8 shows an algorithm to solve this problem. The input is a transition graph G together with a set F of weak-fairness constraints and a region σ^T. The goal is to find a reachable σ^T-cycle that is F-fair.

The algorithm is similar in spirit to Algorithm 10.3, and involves two nested depth-first searches. Multiple weak-fairness constraints are reduced to a single weak-fairness constraint by adding a counter variable as discussed in Section 10.3.2. If F contains ℓ constraints, then the modified search space is $\Sigma_G \times \{1, \ldots \ell\}$. The algorithm uses the following operations:

Size : **set** $\mapsto$ **integer.** The operation *Size* returns the number of elements in its set argument.

MultiWeakPost : **state** $\times$ **integer** $\times$ **enumgraph** $\times$ **set of action** $\mapsto$ **queue of state** $\times$ **integer.** The operation $\mathit{MultiWeakPost}(s, i, G, F)$ returns a queue that contains the successors of the state (s, i) in the fair graph $B_{G,F}$.

The operation $\mathit{MultiWeakPost}(s, i, G, F)$ can be implemented from the operation $\mathit{PostQueue}(s, G)$ and membership test for the actions in F.

The fair graph $B_{G,F}$ has a single weak-fairness constraint α containing transitions in which the counter is updated from ℓ to 1. The secondary search is invoked at the conclusion of the primary search from a source state of some transition in α. The secondary search is performed by the function *NDFS* of Figure 10.9. The goal of the secondary search is to find a fair cycle. Consider a fair transition from (s, ℓ) to $(t, 1)$. Then, the variable *Root* is set to s, and the function *NDFS* checks if the state (s, ℓ) is reachable from the state $(t, 1)$. Since the fair cycle is required to stay within the region σ^T, search is restricted to the region σ^T. The correctness of Algorithm 10.4 can be established as in case of Algorithm 10.3: if $(s, \ell) \rightarrow (t, 1)$ and $(s', \ell) \rightarrow (t', 1)$ are two fair transitions such that (1) the primary search from (s, ℓ) terminates before the primary search from (s', ℓ), (2) there is no σ^T-path from $(t, 1)$ to (s, ℓ), then if there is a σ^T-path from $(t', 1)$ to (s', ℓ), then it does not contain any state reachable from $(t, 1)$.

Proposition 10.8 [Recurrence verification for weak-fairness] *Let G be a transition graph with finitely many reachable states, let F be a set of weak-fairness constraints for G, and let σ^T be a region of G. Then Algorithm 10.4 solves the fair cycle problem $(G, F \cup \{(\neg\sigma^T, \emptyset)\})$.*

Algorithm 10.4 [Primary Depth First Search for Weak Recurrence Verification]

```
Input: a finitely branching transition graph $G$, weak fairness as-
    sumption $F$, and a region $\sigma^T$ of $G$.
Output: the answer to the fair-cycle problem $(G, F \cup \{(\neg\sigma^T, \emptyset)\})$
input $G$: enumgraph; $F$: set of action; $\sigma^T$: enumreg;
local $\sigma, \tau$: set of state × integer; $s, Root$: state
begin
  $\sigma := EmptySet$;
  $\tau := EmptySet$;
  foreach $s$ in $InitQueue(G)$ do
    if not $IsMember((s, 1), \sigma)$ then
      if $DepthFirstSearch(s, 1)$ then return YES fi;
      fi;
    od;
  return NO
  end.
function $DepthFirstSearch$: $\mathbb{B}$
  input $s$: state; $i$: integer;
  local $t$: state;
  begin
    $\sigma := Insert((s, i), \sigma)$;
    foreach $(t, j)$ in $MultiWeakPost(s, i, G, F)$ do
      if not $IsMember((t, j), \sigma)$ then
        if $DepthFirstSearch(t, j)$ then return $true$ fi;
        fi;
      od;
    if $i = Size(F)$ and $IsMember(s, \sigma^T)$
        and not $IsMember((s, i), \tau)$ then
      $Root := s$;
      foreach $(t, j)$ in $MultiWeakPost(s, i, G, F)$ do
        if $j = 1$ and $IsMember(t, \sigma^T)$
            and not $IsMember((t, j), \tau)$ then
          if $NDFS(t, j)$ then return $true$ fi
          fi
        od;
      fi;
    return $false$
    end.
```

Figure 10.8: Primary search for a fair cycle for weak recurrence verification

```
function $NDFS$: $\mathbb{B}$
  input $s$: state;  $i$: integer
  local $t$: state;  $j$: integer
  begin
    $\tau := \mathit{Insert}((s,i),\tau)$;
    foreach $(t,j)$ in $\mathit{MultiWeakPost}(s,i,G,F)$ do
      if $t = \mathit{Root}$ and $j = \mathit{Size}(F)$ then return $\mathit{true}$ fi;
      if not $\mathit{IsMember}((t,j),\tau)$ and $\mathit{IsMember}(t,\sigma^T)$ then
        if $\mathit{NDFS}(t,j)$ then return $\mathit{true}$ fi;
        fi;
      od;
    return $\mathit{false}$
    end.
```

Figure 10.9: Secondary search for a fair cycle for weak recurrence verification

For complexity analysis of Algorithm 10.4, for a reachable state s, and an integer $1 \leq i \leq \ell$, both the routines *DepthFirstSearch* and *NDFS* are invoked at most once with input (s,i). Using a standard enumerative representation, the algorithm can be implemented with running time $((n+m)\cdot\ell)$.

Exercise 10.14 {P3} [Witness reporting] Modify Algorithm 10.4 so that it returns either the answer No or a witness to the fair-cycle problem. ■

Exercise 10.15 {T3} [Modifying the weak recurrence verification algorithm] Suppose we modify Algorithm 10.4 so that at the conclusion of the primary search from a state $(s,\ell) \notin \tau$ with $s \in \sigma^T$, the state (s,ℓ) is added to the set τ before invoking the secondary search (i.e. add the line $\tau := \mathit{Insert}((s,i),\tau)$ immediately after the assignment $\mathit{Root} := s$ in Figure 10.8). Does the modified algorithm correctly solve the fair-cycle problem? ■

Exercise 10.16 {P3} [Fair-cycle for single strong fairness constraint] Consider the fair-cycle problem $(G,\{(\sigma,\tau)\})$ with a single strong-fairness constraint. The goal is, then, to find a reachable cycle that is either τ-fair or not σ-fair. Write a nested depth first search algorithm to solve this special case. ■

Contents

Chapter 11

Temporal Liveness Requirements

Not all liveness requirements of a reactive module can be formulated as a response verification problem. In Chapter 5, we studied temporal logics over observation structures to specify logical safety requirements of a module. Along the same lines, we now consider temporal logics over observation structures with fairness constraints to specify logical liveness requirements of a fair module.

11.1 Fair Structures

11.1.1 ω-Traces

A trace of an observation structure is obtained by executing the underlying transition graph for finitely many steps, and mapping each state to its observation. Similarly, an ω-trace of an observation structure is obtained by considering an ω-trajectory of the underlying transition graph, and mapping each state to its observation.

ω-TRACES

Let $K = (G, A, \langle\!\langle \cdot \rangle\!\rangle)$ be an observation structure. An ω*-trace* of K is an ω-word $\underline{a}$ over the alphabet A of observations such that there is an ω-trajectory $\underline{s}$ of G with $\underline{a} = \langle\!\langle \underline{s} \rangle\!\rangle$. The ω-word $\underline{a}$ is an *initialized w-trace* of K if there is an initialized ω-trajectory $\underline{s}$ of G with $\underline{a} = \langle\!\langle \underline{s} \rangle\!\rangle$. The ω-language $\mathcal{L}_K$ of the observation structure K is the set of initialized ω-traces of K.

Remark 11.1 [Fusion-closure] The ω-language $\mathcal{L}_K$ of an observation structure K is not necessarily fusion-closed. ■

We know that the ω-language of a transition graph is safe. What about the ω-language of an observation structure? Clearly, if $\underline{a}$ is an ω-trace of an observation structure K, then every prefix of $\underline{a}$ is a trace of K. However, to establish that the set $\mathcal{L}_K$ contains every limit of L_K, we use the fact that the structure is finitely-branching over observations: for every observation a, there are only finitely many initial states with observation a, and every state has only finitely many successors with observation a.

Proposition 11.1 [Limit closure of ω-traces] *Let K be an observation structure. Then, $\mathcal{L}_K = \mathit{safe}(L_K)$.*

Proof. Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle)$ be an observation structure. If $\underline{a}$ is an ω-trace then, by definition, for all $i \geq 0$, the prefix $\overline{a}_{0\ldots i}$ is a trace of K. This establishes $\mathcal{L}_K \subseteq \mathit{safe}(L_K)$.

We wish to prove that every limit of L_K is an ω-trace of K. Consider an ω-word $\underline{a}$ over the set of observations of K, and suppose for all $i \geq 0$, $\overline{a}_{0\ldots i}$ is an initialized trace of K. Let us define a transition graph H. The states of H are pairs of the form (s, i), for a state s of K and a natural number i, such that $\langle\!\langle s \rangle\!\rangle = a_i$. The state (s, i) is an initial state of H if s is an initial state of K and $i = 0$. The transition graph H has a transition from the state (s, i) to the state (t, j) if there is a transition from s to t in K and $j = i + 1$. The graph H has finitely many initial states, and each state has finitely many successors. For every $i \geq 0$, $\underline{a}_{0\ldots i}$ is a trace of K, and hence, there exists a trajectory $\overline{s}_{0\ldots i}$ of K with $\langle\!\langle s_j \rangle\!\rangle = a_j$ for $0 \leq j \leq i$. Hence, for every $i \geq 0$, there is a trajectory of H of length i. From König's lemma, the graph H has an infinite path: $(s_0, 0)(s_1, 1)(s_2, 2)\cdots$ The corresponding ω-word $\underline{s}$ is an ω-trajectory of K, and $\langle\!\langle \underline{s} \rangle\!\rangle = \underline{a}$ is an ω-trace of K. ■

Since a safe language is completely characterized by the set of its prefixes, the next theorem follows.

Theorem 11.1 [Language inclusion] *For two observation structures K_1 and K_2, $L_{K_1} \subseteq L_{K_2}$ iff $\mathcal{L}_{K_1} \subseteq \mathcal{L}_{K_2}$. For two reactive modules P_1 and P_2, $L_{P_1} \subseteq L_{P_2}$ iff $\mathcal{L}_{P_1} \subseteq \mathcal{L}_{P_2}$.*

11.1.2 Fair traces

A fairness constraint and a fairness assumption for an observation structure is a fairness constraint and a fairness assumption for the underlying transition graph. Fair structures are obtained from observation structures by adding fairness assumptions.

FAIR STRUCTURE

A *fair structure* $\mathcal{K} = (K, F)$ consists of an observation structure K and a fairness assumption F for K.

If all the fairness constraints in F are weak, then (K, F) is a *weakly-fair* structure.

Fair traces of a fair structure are obtained from fair trajectories by projecting states to observations.

> FAIR TRACE
>
> Let $\mathcal{K}$ be a fair structure with observations A and observation function $\langle\langle\cdot\rangle\rangle$. An ω-word $\underline{a}$ is a fair trace of $\mathcal{K}$ if there exists a fair trajectory $\underline{s}$ of $\mathcal{K}$ such that $\underline{a} = \langle\langle\underline{s}\rangle\rangle$. An ω-word $\underline{a}$ is an initialized fair trace of $\mathcal{K}$ if there exists an initialized fair trajectory $\underline{s}$ of $\mathcal{K}$ such that $\underline{a} = \langle\langle\underline{s}\rangle\rangle$. The *fair language* $\mathcal{L}_{\mathcal{K}}$ of $\mathcal{K}$ the set of initialized fair traces of $\mathcal{K}$.

Exercise 11.1 {`T3`} [Fair traces] Show that the set of fair initialized traces of a fair structure is not necessarily the intersection of the set of fair traces and the set of initialized ω-traces. ■

In Chapter 11, we will study fair structures as a specification formalism for fair languages.

The fair structure of a fair module

Every reactive module P defines the observation structure K_P. The fair structure of a fair module $(P, \mathit{WeakF}_P, \mathit{StrongF}_P)$ is obtained from the observation structure K_P by adding all the fairness constraints corresponding to the declaration of weak and strong fair update choices.

> FAIR STRUCTURE OF A FAIR MODULE
>
> The fair module $\mathcal{P}$ defines the fair structure $\mathcal{K}_{\mathcal{P}} = (K_P, F_{\mathcal{P}})$.

Observe that if $\mathcal{P}$ has only weak-fairness constraints, then the corresponding fair structure $\mathcal{K}_{\mathcal{P}}$ is also weakly fair.

11.2 The Temporal Logic CTL

We specify requirements of fair modules using fair state logics. The formulas of fair state logics are interpreted over the states of fair structures, and may refer to the infinite behavior of fair structures. The satisfaction relation of a fair state logic defines, for each formula ϕ and each fair ϕ-structure $\mathcal{K}$, the characteristic region $[\![\phi]\!]_{\mathcal{K}}$. We start by extending STL to a fair state logic called CTL.

Remark 11.2 [State logics vs. Fair state logics] Every state logic is a fair state logic. For every formula ϕ of a state logic and a fair structure $\mathcal{K} = (K, F)$, the characteristic regions $[\![\phi]\!]_{\mathcal{K}}$ and $[\![\phi]\!]_K$ coincide. Every fair state logic is also a state logic: for a formula ϕ of a fair state logic, an observation structure K, and a state s of K, $s \models_K \phi$ iff $s \models_{(K,\emptyset)} \phi$. ■

11.2.1 Syntax and Semantics of Ctl

The fair state logic Ctl is obtained from the state logic Stl by adding the unary temporal connective *possibly-always*, written $\exists\Box$. Consider a state s of a fair structure $\mathcal{K}$, and let p be an observation predicate of $\mathcal{K}$. The state s satisfies the formula $\exists\Box p$ if there is a source-s fair trajectory all of whose states satisfy p. In other words, the formula $\exists\Box p$ asserts that it is possible to execute infinitely many rounds in a fair fashion so that p is satisfied at every step.

Computation tree logic

The formulas of Ctl are defined inductively by the grammar

$$\phi ::= p \mid \phi \vee \phi \mid \neg\phi \mid \exists\bigcirc\, \phi \mid \exists\Box\phi \mid \phi\exists\mathcal{U}\phi$$

for atomic formulas p. For a Ctl formula ϕ, if $\mathcal{K} = (K, F)$ is a fair ϕ-structure, and s is a state of K, then

$$s \models_{\mathcal{K}} \exists\Box\phi \quad \text{iff} \quad \begin{array}{l}\text{there is an source-}s\text{ fair trajectory } \underline{s} \text{ of } \mathcal{K} \\ \text{such that for all } i \geq 0,\ s_i \models_{\mathcal{K}} \phi.\end{array}$$

The interpretation of the temporal connectives $\exists\bigcirc$ and $\exists\mathcal{U}$ is the same as in Stl: a state s satisfies the possibly-next formula $\exists\bigcirc\, p$ if some successor of s satisfies p; a state s satisfies the possibly-until formula $p\,\exists\mathcal{U}\,q$ if there exists a source-s trajectory $\overline{s}_{0\ldots m}$ such that s_m satisfies q and s_i satisfies p for all $0 \leq i < m$. The temporal connectives $\forall\bigcirc$, $\exists\Diamond$, $\forall\Box$, and $\forall\mathcal{W}$ are defined from $\exists\bigcirc$ and $\exists\mathcal{U}$ as in Stl. In addition, we define the following temporal connectives in Ctl:

Inevitably	$\forall\Diamond\phi$	for	$\neg\exists\Box\neg\phi$;
Inevitably-until	$\psi\forall\mathcal{U}\phi$	for	$\psi\forall\mathcal{W}\phi \wedge \forall\Diamond\phi$;
Possibly-waiting-for	$\psi\exists\mathcal{W}\phi$	for	$\psi\exists\mathcal{U}\phi \vee \exists\Box\psi$.

The modality $\forall\Diamond$ is the dual of $\exists\Box$: a state s of the fair structure $\mathcal{K}$ satisfies the Ctl formula $\forall\Diamond\, p$ if every source-s fair ω-trajectory contains a p-state.

Exercise 11.2 {T2} [Inevitably-until connective] Let ϕ and ψ be two Ctl formulas, let $\mathcal{K}$ be a (ϕ, ψ)-structure, and let s be a state of K. Show that $s \models_K \phi\forall\mathcal{U}\,\psi$ iff for all source-s fair ω-trajectories $\underline{s}$ of $\mathcal{K}$, there exists a position $m \geq 0$ such that (1) $s_m \models_{\mathcal{K}} \psi$ and (2) for all $0 \leq i < m$, $s_i \models_{\mathcal{K}} \phi$. ■

Remark 11.3 [Fair emptiness] The fair-emptiness problem is a special case of Ctl model checking: for a fair structure $\mathcal{K}$, the answer to the fair emptiness problem $\mathcal{K}$ is Yes iff $s \models_{\mathcal{K}} \exists\Box true$ for some initial state s of $\mathcal{K}$. ■

Response verification problem is also a special case of Ctl verification:

Proposition 11.2 [Response verification in CTL] *Let $\mathcal{P}$ be a fair module and let p and q be two observation predicates of $\mathcal{P}$. Then, $p \rightsquigarrow_{\mathcal{P}} q$ iff $\mathcal{P} \models \forall\Box(p \rightarrow \forall\Diamond\, q)$.*

Remark 11.4 [Recurrence verification] Recurrence verification is also a special case of CTL verification: The observation predicate p is a recurrent of a fair structure $\mathcal{K}$ if $\mathcal{K} \models \forall\Box\, \forall\Diamond\, p$. ■

Example 11.1 [Mutual exclusion] For a mutual-exclusion protocol with weak fairness, the *deadlock-freedom requirement* asserts that if a process requests the critical section, then some process is eventually in the critical section:

$$\phi_{df} : \forall\Box((pc_1 = reqC \;\vee\; pc_2 = reqC) \;\rightarrow\; \forall\Diamond(pc_1 = inC \;\vee\; pc_2 = inC))$$

The *starvation-freedom requirement* asserts that if a process requests the critical section, then that process eventually enters the critical section:

$$\phi_{sf} : \forall\Box((pc_1 = reqC \;\rightarrow\; \forall\Diamond pc_1 = inC) \wedge (pc_2 = reqC \;\rightarrow\; \forall\Diamond pc_2 = inC)).$$

The fair module *FairPete* satisfies both ϕ_{df} and ϕ_{sf}. It also satisfies the stronger until-requirement:

$$\forall\Box\left(\begin{array}{c} (pc_1 = reqC \;\rightarrow\; (pc_1 = reqC)\forall\mathcal{U}(pc_1 = inC)) \\ \wedge \\ (pc_2 = reqC \;\rightarrow\; (pc_2 = reqC)\forall\mathcal{U}(pc_2 = inC)) \end{array}\right).$$

■

Exercise 11.3 {T2} [CTL connectives] The CTL formula ϕ *implies* the CTL formula ψ if $[\![\phi]\!]_{\mathcal{G}} \subseteq [\![\psi]\!]_{\mathcal{G}}$ for all fair (ϕ, ψ)-structures $\mathcal{G}$ (i.e., the CTL formula $\phi \rightarrow \psi$ is valid). Let p be an atomic state formula. Group the 16 CTL formulas of the form $Q_1 T_1 Q_2 T_2\, p$, where $Q_1, Q_2 \in \{\forall, \exists\}$ and $T_1, T_2 \in \{\Box, \Diamond\}$, into eight pairs (ϕ, ψ) (any such grouping is fine). Prove or disprove that ϕ implies ψ for each of your pairs. ■

Exercise 11.4 {T3} [Interdefinability of temporal connectives] Assuming that a fair state logic contains the temporal connective $\exists\mathcal{U}$, show that each of the connectives $\exists\Box$, $\forall\Diamond$, $\forall\mathcal{U}$, and $\exists\mathcal{W}$ can be used to define the remaining three. ■

Every STL formula is also a CTL formula. For a STL formula ϕ and a fair structure $\mathcal{K} = (K, F)$, $[\![\phi]\!]_{\mathcal{K}} = [\![\phi]\!]_{K}$. This implies that to check STL specifications of fair structures we can ignore the fairness constraints.

Remark 11.5 [Fair semantics of STL] Note that while interpreting STL formulas over states of fair structures, we have retained the original semantics of STL over observation structures. To account for the fairness constraints, suppose we redefine the semantics of STL over fair structures the following way. The state s of a fair structure $\mathcal{K}$ satisfies the possibly-until formula $\phi\, \exists\mathcal{U}\, \psi$ if there exists a source-s fair trajectory $\underline{s}$ of $\mathcal{K}$ such that for some $m \geq 0$, $s_m \models_{\mathcal{K}} \psi$ and $s_i \models_{\mathcal{K}} \phi$ for $0 \leq i < m$. For machine-closed fair structures $\mathcal{K}$, since every finite trajectory is a prefix of some fair ω-trajectory, this new definition of $\exists\mathcal{U}$ coincides with the old definition. ■

11.2.2 CTL Model Checking

In the model-checking problem for CTL, we are given a CTL formula ϕ and a fair ϕ-structure $\mathcal{K}$. To compute the characteristic region $[\![\phi]\!]_K$, we proceed inductively on the structure of the formula ϕ, by first finding the characteristic regions for the subformulas of ϕ. For this purpose, we first compute the the set $\mathit{Sub}(\phi)$ of subformulas of ϕ. The function Sub is extended to include the new connective $\exists\Box$:

$$\mathit{Sub}(\exists\Box\psi) \;=\; \{\exists\Box\psi\} \cup \mathit{Sub}(\psi).$$

The function *OrderedSub* is also redefined so that it accepts a CTL formula ϕ as input, and returns a queue with the formulas in $\mathit{Sub}(\phi)$ such that if $\psi \in \mathit{Sub}(\chi)$ and $\chi \in \mathit{Sub}(\phi)$, then ψ precedes χ in $\mathit{OrderedSub}(\phi)$. As in case of STL, the CTL formula ϕ has at most $|\phi|$ subformulas.

For the enumerative algorithm, assume that the atomic formulas of ϕ are propositions, and the fair structure $\mathcal{K}$ is finite. The algorithm computes, for each state s of $\mathcal{K}$, the set $\lambda(s) \subseteq \mathit{Sub}(\phi)$ of subformulas of ϕ that are satisfied by the state s. Initially, $\lambda(s)$ is empty for each state s. The algorithm considers each subformula ψ, in the order given by $\mathit{OrderedSub}(\phi)$, and decides, for every state s, whether s satisfies ψ, and updates $\lambda(s)$ accordingly (see Algorithm 5.1 for enumerative STL model checking). The structure of ψ leads to various cases. The cases corresponding to propositions, logical connectives, and the temporal connectives $\exists\bigcirc$ and $\exists\mathcal{U}$ are handled as in the case of STL. The case when $\psi = \exists\Box\chi$ is reduced to the fair-region problem.

The fair-region problem

The fair-region problem is to determine which states belong to the fair ω-trajectories of a fair graph. Let G be a transition graph, and let F be a fairness assumption for G. The F-fair region σ^F of G consists of precisely the states s such that there is a source-s F-fair ω-trajectory of G.

An instance of the *fair-region* problem is a fair graph (G, F). The answer to the fair-region problem (G, F) is the F-fair region σ^F of G.

Remark 11.6 [Fair-region problem vs. fair-emptiness problem] For a fair graph $\mathcal{G}$, the answer to the fair-emptiness problem $\mathcal{G}$ is YES iff $\sigma^I \cap \sigma^F$ is nonempty. ■

For CTL model checking, we need to construct the characteristic region $[\![\exists\Box\chi]\!]_{\mathcal{K}}$ from the characteristic region $[\![\chi]\!]_{\mathcal{K}}$ for the fair structure $\mathcal{K} = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle, F)$. Let G_χ be the transition graph with the state space $[\![\chi]\!]_{\mathcal{K}}$ and the transition relation $\rightarrow$ restricted to $[\![\chi]\!]_{\mathcal{K}}$. The region $[\![\exists\Box\chi]\!]_{\mathcal{K}}$ is precisely the answer to the fair-region problem (G_χ, F).

To solve the fair-region problem (G, F), observe that a state s belongs to the fair region σ^F iff there exists a F-fair strongly connected component σ of G such that $post^*(s) \cap \sigma$ is nonempty. Thus,

$$\sigma^F \;=\; \bigcup\{\sigma \mid \sigma \text{ is a } F\text{-fair component of } G\}.\, pre^*(\sigma).$$

Hence, σ^F can be by first computing the F-fair strongly connected components of G using Algorithm 9.2, and then computing the region σ^F by a depth-first search. If G has n states and m transitions, and F contains ℓ fairness constraints, then the overall time-complexity is $O((n+m)\cdot\ell^2)$.

Exercise 11.5 {P3} [Fair-region problem] Write an enumerative algorithm to solve the fair-region problem $\mathcal{G}$ using an on-the-fly representation of the fair graph. ■

Theorem 11.2 [Model checking of CTL] *Let $\mathcal{K} = (K, F)$ be a fair structure, and let ϕ be an* CTL *formula. Suppose K has n states and m transitions, and F has ℓ fairness constraints. The model-checking problem $(\mathcal{K}, \phi)$ can be solved in $O((n+m)\cdot\ell^2\cdot|\phi|)$ time.*

The algorithms of Section 9.3 that employ nested depth-first search can be used to solve the fair-region problem when the fairness assumption is of a restricted form. When the fairness assumption F contains only weak constraints, the CTL model-checking problem $((K, F), \phi)$ can be solved in in time linear in the number of fairness constraints.

Theorem 11.3 [Model checking of CTL for weak-fair structures] *Let $\mathcal{K} = (K, F)$ be a weak-fair structure, and let ϕ be an* CTL *formula. Suppose K has n states and m transitions, and F has ℓ weak-fairness constraints. The model-checking problem $(\mathcal{K}, \phi)$ can be solved in $O((n+m)\cdot\ell\cdot|\phi|)$ time.*

In particular, the CTL model-checking problem (K, ϕ) can be solved in $O((n+m)\cdot|\phi|)$ time. Thus, the additional complexity of CTL model checking over STL model checking is not due to the introduction of $\exists\Box$ connective in the logic, but due to the introduction of fairness constraints in the model.

To solve the Ctl-verification problem $(\mathcal{P}, \phi)$, for a finite fair module $\mathcal{P}$ and a Ctl specification ϕ, we can first construct the fair structure $\mathcal{K}_{\mathcal{P}}$, and then employ the model checking algorithm. As usual, since the structure $\mathcal{K}_{\mathcal{P}}$ may be exponentially larger than the module description, this results in an exponential algorithm. As in case of Stl, the Ctl verification problem of determining whether a fair module satisfies a Ctl-formula is Pspace-complete.

11.2.3 Compositionality and Ctl

As in Stl, satisfaction of existential Ctl-formulas is not preserved under parallel composition.

Exercise 11.6 {T3} [Non-compositionality of Ctl] Give an example of a fair module $\mathcal{P} \parallel \mathcal{Q}$ and an observation predicate p such that the answer to the verification problem $(\mathcal{P}, \exists\Box p)$ is Yes, while the answer to $(\mathcal{P} \parallel \mathcal{Q}, \exists\Box p)$ is No. ■

As in case on Stl, if we restrict ourselves only to the universal formulas, then the compositionality principle holds. Let ∀Ctl be the fragment of Ctl generated by the grammar

$$\phi \ ::= \ p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall\bigcirc \phi \mid \phi \forall\mathcal{U} \phi \mid \forall\Box\phi$$

The logic ∀Ctl is not closed under negation. The parallel composition operation on fair modules ensures that the projection of a fair trajectory of a compound module onto the variables of a component is a fair trajectory of that component. This implies that the compositionality principle holds for ∀Ctl.

Proposition 11.3 [Compositionality for ∀Ctl] *If the fair module $\mathcal{P}$ satisfies the ∀Ctl-formula ϕ, then for every fair module $\mathcal{Q}$ that is compatible with $\mathcal{P}$, the compound fair module $\mathcal{P} \parallel \mathcal{Q}$ satisfies ϕ.*

Exercise 11.7 {T3} [Compositionality of ∀Ctl] Prove Proposition 11.3. ■

11.3 The μ-Calculus

We now introduce a state logic, called *μ-calculus*, that is more expressive than Ctl. Before we present syntax and semantics of μ-calculus, two points must be noted. First, comprehending μ-calculus formulas requires considerable expertise, and hence, it is not a convenient specification language for writing requirements. On the other hand, its semantics immediately suggests a symbolic procedure for model checking. The role of of μ-calculus, then, is as an intermediate language which can be analyzed by symbolic algorithms. Second, the syntax of μ-calculus is expressive enough to specify fairness constraints. Consequently, we consider μ-calculus as a state logic, and interpret its formulas over states of observation structures.

11.3.1 Syntax and semantics

In μ-calculus, properties are expressed as fixpoints of functions that map regions to regions. As an example, consider the STL-formula $\exists\Diamond p$. The characteristic region $[\![\exists\Diamond p]\!]_K$ consists of all states of the observation structure K from which a state satisfying p is reachable. Consider the function $\mathcal{F}_{\exists\Diamond p}$ that maps regions of K to regions of K:

$$\mathcal{F}_{\exists\Diamond p}(\sigma) \;=\; [\![p]\!]_K \cup \mathit{pre}_K(\sigma).$$

Then, the region $[\![\exists\Diamond\, p]\!]_K$ is the least fixpoint of the function $\mathcal{F}_{\exists\Diamond p}$: it is the smallest region σ that contains $[\![p]\!]_K$ as well as $\mathit{pre}_K(\sigma)$. The μ-calculus formula corresponding to $\exists\Diamond p$ is $\mu\mathbf{x}.\,(p \vee \exists\bigcirc\mathbf{x})$. Here, the variable $\mathbf{x}$ ranges over regions, $\mu\mathbf{x}.$ is called the *least fixpoint operator*, and given a region σ, $\exists\bigcirc\,\sigma$ denotes the region containing states that have at least one successor in σ.

The dual of the least fixpoint operator is the *greatest fixpoint operator* $\nu\mathbf{x}$. As an example, the characteristic region $[\![\forall\Box p]\!]_K$ is the greatest fixpoint of the function $\mathcal{F}_{\forall\Box p}$ that maps regions of K to regions of K:

$$\mathcal{F}_{\forall\Box p}(\sigma) \;=\; [\![p]\!]_K \cap \{s \mid \mathit{post}_K(s) \;\subseteq\; \sigma\}.$$

The μ-calculus formula corresponding to $\forall\Box p$ is $\nu\mathbf{x}.\,(p \,\wedge\, \forall\bigcirc\,\mathbf{x})$.

> μ-CALCULUS SYNTAX
>
> Let **Var** be a set of *region variables*. The formulas of the *μ-calculus* (CTμ) are defined inductively by the grammar
>
> $$\phi \;::=\; p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists\bigcirc\,\phi \mid \forall\bigcirc\,\phi \mid \mu\mathbf{x}.\,\phi \mid \nu\mathbf{x}.\,\phi \mid \mathbf{x},$$
>
> where p is an atomic formula and $\mathbf{x} \in \mathbf{Var}$ is a region variable.

A CTμ formula of the form $\mu\mathbf{x}.\,\phi$ is called a *μ-formula*, and a CTμ formula of the form $\nu\mathbf{x}.\,\phi$ is called a *ν-formula*. A μ-formula or a ν-formula is also called a *fixpoint-formula*. The fixpoint operator is like a quantifier in first-order logic. Every occurrence of a region variable $\mathbf{x}$ in a formula is either free or bound, and if bound, has a unique fixpoint operator that binds it. The CTμ formula ϕ is *closed* if for all region variables $\mathbf{x} \in \mathbf{Var}$, each occurrence of $\mathbf{x}$ in ϕ is bound by a fixpoint operator. The CTμ formula ϕ is *open* if it contains a free occurrence of a region variable.

The logic CTμ is a state logic, and its formulas are interpreted over states of observation structures. As in state logics, for a formula ϕ of CTμ, a ϕ-structure is an observation structure whose observations give interpretation to the atomic formulas appearing in ϕ.

μ-CALCULUS SEMANTICS

Let $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ be an observation structure. A *region environment* $\mathbf{E}$ assigns to each region variable $\mathbf{x} \in \mathbf{Var}$ a region $\sigma \subseteq \Sigma$. Given a state $s \in \Sigma$ and a region environment $\mathbf{E}$,

$$
\begin{array}{lll}
s \models_{K,\mathbf{E}} p & \text{iff} & \langle\!\langle s\rangle\!\rangle \models p; \\
s \models_{K,\mathbf{E}} \neg p & \text{iff} & \langle\!\langle s\rangle\!\rangle \models \neg p; \\
s \models_{K,\mathbf{E}} \phi_1 \wedge \phi_2 & \text{iff} & s \models_{K,\mathbf{E}} \phi_1 \text{ and } s \models_{K,\mathbf{E}} \phi_2; \\
s \models_{K,\mathbf{E}} \phi_1 \vee \phi_2 & \text{iff} & s \models_{K,\mathbf{E}} \phi_1 \text{ or } s \models_{K,\mathbf{E}} \phi_2; \\
s \models_{K,\mathbf{E}} \exists\bigcirc\, \phi & \text{iff} & \text{for some state } t \in \mathit{post}_K(s),\ t \models_{K,\mathbf{E}} \phi; \\
s \models_{K,\mathbf{E}} \forall\bigcirc\, \phi & \text{iff} & \text{for all states } t \in \mathit{post}_K(s),\ t \models_{K,\mathbf{E}} \phi; \\
s \models_{K,\mathbf{E}} \mu\mathbf{x}.\, \phi & \text{iff} & \text{for all fixpoints } \sigma \text{ of } \mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}},\ s \in \sigma \\
s \models_{K,\mathbf{E}} \nu\mathbf{x}.\, \phi & \text{iff} & \text{for some fixpoint } \sigma \text{ of } \mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}},\ s \in \sigma \\
s \models_{K,\mathbf{E}} \mathbf{x} & \text{iff} & s \in \mathbf{E}(\mathbf{x}).
\end{array}
$$

The function $\mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}}$ maps regions to regions: for all regions $\sigma \subseteq \Sigma$ and all states $s \in \Sigma$,

$$s \in \mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}}(\sigma) \text{ iff } s \models_{K,\mathbf{E}[\mathbf{x}:=\sigma]} \phi.$$

From the following proposition it follows by the Knaster-Tarski fixpoint theorem that the function $\mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}}$ has a least fixpoint as well as a greatest fixpoint.

Proposition 11.4 [Monotonicity in μ-calculus] *Let ϕ be a* C<span style="font-variant:small-caps">t</span>μ *formula and let* $\mathbf{E}$ *be a region environment. The function $\mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}}$ is monotonic; that is, $\sigma \subseteq \tau$ implies $\mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}}(\sigma) \subseteq \mathcal{F}^{\mathbf{x},\phi}_{K,\mathbf{E}}(\tau)$.*

Exercise 11.8 {T3} [Monotonicity in μ-calculus] Prove Proposition 11.4. ■

Remark 11.7 [Region environments in μ-calculus] Let ϕ be a CTμ formula. If two region environments $\mathbf{E}$ and $\mathbf{E}'$ agree on the values of the region variables that are free in ϕ, then $s \models_{K,\mathbf{E}} \phi$ iff $s \models_{K,\mathbf{E}'} \phi$. In particular, for a closed formula ϕ, in the definition of the satisfaction relation $\models_{K,\mathbf{E}}$, the value of $\mathbf{E}$ is not important. ■

A state $s \in \Sigma$ *satisfies* the closed CTμ formula ϕ, written $s \models_K \phi$, if $s \models_{K,\mathbf{E}} \phi$ for all region environments $\mathbf{E}$. For notational convenience, we admit regions as formulas of state logics: for all regions $\sigma \subseteq \Sigma$ and all states $s \in \Sigma$, $s \models_K \sigma$ iff $s \in \sigma$. Given a region environment $\mathbf{E}$, a CTμ formula of the form $\mu\mathbf{x}.\,\phi$, then, defines the least fixpoint of the function $\mathcal{F}^{\phi}_{K,\mathbf{E}} \colon \mathbf{2}^\Sigma \rightarrow \mathbf{2}^\Sigma$ that maps each region $\sigma \subseteq \Sigma$ to the region $[\![\phi[\mathbf{x} := \sigma]]\!]_{K,\mathbf{E}}$; that is,

$$[\![\mu\mathbf{x}.\,\phi]\!]_{K,\mathbf{E}} = \bigcup_{\kappa\in} (\mathcal{F}^{\phi}_{K,\mathbf{E}})^{\kappa}(\emptyset).$$

Exercise 11.9 {T4} [Continuity in μ-calculus] A transition relation → is finitely branching iff every state has finitely many successors. Let K be an observation structure. (1) Prove that the function pre_K that maps regions of K to regions of K is $\bigcap$-continuous iff the transition relation of K is finitely branching. (2) Prove that the function $\mathcal{F}^{\mathbf{X},\phi}_{K,\mathbf{E}}$ is both $\bigcup$-continuous and $\bigcap$-continuous if the function pre_K is $\bigcap$-continuous. It follows that finite branching of the transition relation is both a sufficient and necessary condition for continuity of the functions $\mathcal{F}^{\mathbf{X},\phi}_{K,\mathbf{E}}$. ■

If the transition relation of K is finitely branching, then the function $\mathcal{F}^{\phi}_{K,\mathbf{E}}$ is $\bigcup$-continuous (Exercise 11.9) and, by the Kleene fixpoint theorem,

$$[\![\mu\mathbf{x}.\,\phi]\!]_{K,\mathbf{E}} \;=\; \bigcup_{i\in\mathbb{N}} (\mathcal{F}^{\phi}_{K,\mathbf{E}})^i(\emptyset);$$

that is, the characteristic region $[\![\mu\mathbf{x}.\,\phi]\!]_{K,\mathbf{E}}$ is the limit of the infinite approximation sequence $\emptyset$, $\mathcal{F}^{\phi}_{K,\mathbf{E}}(\emptyset)$, $\mathcal{F}^{\phi}_{K,\mathbf{E}}(\mathcal{F}^{\phi}_{K,\mathbf{E}}(\emptyset))$, etc. We will use this observation to compute the characteristic regions of C<span style="font-variant:small-caps">t</span>μ formulas. For example,

$$[\![\mu\mathbf{x}.\,(p\,\vee\,\exists\bigcirc\mathbf{x})]\!] \;=\; [\![\mathit{false}]\!]\cup[\![p]\!]\cup[\![\exists\bigcirc p]\!]\cup[\![\exists\bigcirc\exists\bigcirc p]\!]\cup\cdots$$

Similarly, a CTμ formula of the form $\nu\mathbf{x}.\,\phi$, then, defines the greatest fixpoint of the function $\mathcal{F}^{\phi}_{K,\mathbf{E}}$:

$$[\![\nu\mathbf{x}.\,\phi]\!]_{K,\mathbf{E}} \;=\; \bigcap_{\kappa\in} (\mathcal{F}^{\phi}_{K,\mathbf{E}})^{\kappa}(\Sigma).$$

If the transition relation of K is finitely branching, then the function $\mathcal{F}^{\phi}_{K,\mathbf{E}}$ is $\bigcap$-continuous (Exercise 11.9) and the characteristic region $[\![\nu\mathbf{x}.\,\phi]\!]_{K,\mathbf{E}}$ is the limit of the infinite approximation sequence Σ, $\mathcal{F}^{\phi}_{K,\mathbf{E}}(\Sigma)$, $\mathcal{F}^{\phi}_{K,\mathbf{E}}(\mathcal{F}^{\phi}_{K,\mathbf{E}}(\Sigma))$, etc. For example,

$$[\![\nu\mathbf{x}.\,(p\,\wedge\,\forall\bigcirc\mathbf{x})]\!] \;=\; [\![\mathit{true}]\!]\cap[\![p]\!]\cap[\![\forall\bigcirc p]\!]\cap[\![\forall\bigcirc\forall\bigcirc p]\!]\cap\cdots$$

11.3.2 Expressive Power

Alternation depth

For a CTμ formula ϕ, its *nesting depth* is the the length of the longest chain of fixpoint-subformulas of ϕ that are nested in one another. The *alternation depth*, on the other hand, is computed by counting the number of alternations between μ-formulas and ν-formulas along chains of nested fixpoint-subformulas. The alternation depth is a better measure of the complexity of CTμ formulas.

ALTERNATION DEPTH

The *alternation depth* $ad(\phi)$ of a CTμ formula ϕ is defined inductively: If ϕ is not a fixpoint-formula then,

$$ad(\phi) = max\{ad(\psi) \mid \psi \text{ is a fixpoint-subformula of } \phi\};$$

else if $\phi = \mu\mathbf{x}.\,\psi$ then

$$ad(\phi) = max\{1, ad(\psi), 1 + max\{ad(\chi) \mid \chi \text{ is open } \nu\text{-subformula of } \psi\}\};$$

else if $\phi = \nu\mathbf{x}.\,\psi$ then

$$ad(\phi) = max\{1, ad(\psi), 1 + max\{ad(\chi) \mid \chi \text{ is open } \mu\text{-subformula of } \psi\}\}.$$

For every integer $k \geq 0$, the logic CTμ^k consists of all CTμ formulas ϕ with $ad(\phi) \leq k$. The CTμ formula ϕ is said to be *alternation-free* if $ad(\phi) \leq 1$, and the logic CTμ^1 is called alternation-free μ-calculus.

Remark 11.8 [Alternation depth] Alternation depth of a CTμ formula ϕ is the maximum integer $k \geq 0$ such that there exists a sequence $\phi_1\phi_2\ldots\phi_k$ of fixpoint-formulas such that (1) ϕ_1 is a subformula of ϕ, (2) for each $1 \leq j < k$, the formula ϕ_{j+1} is a subformula of ϕ_j, (3) for $2 \leq j \leq k$, the fixpoint-formula ϕ_j is open, and (4) for each $1 \leq j < k$, the types of ϕ_j and ϕ_{j+1} are different: ϕ_j is a μ-formula iff ϕ_{j+1} is a ν-formula. ■

Example 11.2 [Alternation depth] The definition of the alternation-depth is illustrated by the following examples

$$\begin{array}{lcl}
ad(\mu\mathbf{x}.\,p \vee \exists\bigcirc\mathbf{x}) & = & 1\\
ad(\mu\mathbf{x}.\,((\nu\mathbf{y}.\,p \wedge \forall\bigcirc\mathbf{y}) \vee \exists\bigcirc\mathbf{x})) & = & 1\\
ad(\nu\mathbf{x}.\,(p \wedge \exists\bigcirc\nu\mathbf{y}.\,(q \wedge \forall\bigcirc\mathbf{y} \vee \exists\bigcirc\mathbf{x})) & = & 1\\
ad(\nu\mathbf{x}.\,\mu\mathbf{y}.\,((p \wedge \mathbf{x}) \vee \exists\bigcirc\mathbf{y})) & = & 2
\end{array}$$

Note that the nesting depth of the first formula is 1, but for all the rest, the nesting depth is 2. ■

Closure under negation

While the syntax of the logic CTμ does not admit negation, it is effectively closed under negation because every operator has its dual within the logic.

Exercise 11.10 {T3} [Duality of least and greatest fixpoint operators] Let B be a boolean algebra and let $\mathcal{F}\colon B \to B$ be a monotonic function. We write $\mu\mathcal{F}$ for the least fixpoint of $\mathcal{F}$ and $\mathcal{F}^{\neg\neg}$ for the function that maps each $x \in B$ to $\neg\mathcal{F}(\neg x)$. Prove that $\neg\mu\mathcal{F}^{\neg\neg}$ is the greatest fixpoint of $\mathcal{F}$. ■

Proposition 11.5 [Closure under negation] *Let K be an observation structure, and let ϕ be a closed* CTμ *formula. Then, there exists a* CTμ *formula ψ such that for every state s of K, $s \models \phi$ iff $s \not\models \psi$. Furthermore, $ad(\phi) = ad(\psi)$.*

Proof. The proof uses the fact that the logical connectives $\wedge$ and $\vee$ are duals of each other, the temporal connectives $\exists\bigcirc$ and $\forall\bigcirc$ are duals of each other, and the fixpoint operators μ and ν are duals of each other. Specifically, define the function f that maps every CTμ formula to another CTμ formula. The function f is defined inductively:

$$f(p) = \neg p;\; f(\neg p) = p;$$
$$f(\phi_1 \wedge \phi_2) = f(\phi_1) \vee f(\phi_2);\; f(\phi_1 \vee \phi_2) = f(\phi_1) \wedge f(\phi_2);$$
$$f(\exists\bigcirc\,\phi) = \forall\bigcirc\, f(\phi);\; f(\forall\bigcirc\,\phi) = \exists\bigcirc\, f(\phi);$$
$$f(\mu\mathbf{x}.\,\phi) = \nu\mathbf{x}.\,f(\phi);\; f(\nu\mathbf{x}.\,\phi) = \mu\mathbf{x}.\,f(\phi);\; f(\mathbf{x}) = \mathbf{x}.$$

We prove that for every state s of an observation structure K, and a region environment $\mathbf{E}$, $s \models_{K,\mathbf{E}} \phi$ iff $s \not\models_{K,\mathbf{E}} f(\phi)$. This is proved by induction on the structure of ϕ. ■

Remark 11.9 [Alternative definition of CTμ syntax] The syntax of CTμ can alternatively be defined by the following clauses: (1) every atomic formula is a CTμ formula, (2) every region variable is a CTμ formula, (3) if ϕ is a CTμ formula, then so are $\neg\phi$ and $\exists\bigcirc\,\phi$, (4) if ϕ_1 and ϕ_2 are CTμ formulas then so is $\phi_1 \vee \phi_2$, and (5) if ϕ is a CTμ formula, and $\mathbf{x}$ is a region variable that is within the scope of an even number of negations in ϕ, then $\mu\mathbf{x}.\,\phi$ is a CTμ formula. ■

Alternation-free μ-calculus is as expressive as CTL

We establish that every CTL formula is equivalent to an alternation-free CTμ formula over observation structures.

Proposition 11.6 [Fixpoint characterization of $\exists\Diamond$] *Let K be an observation structure, and let p be an observation predicate of K. Then, the characteristic regions $[\![\exists\Diamond p]\!]_K$ and $[\![\mu\mathbf{x}.\,(p \vee \exists\bigcirc\,\mathbf{x})]\!]_K$ are identical.*

Proof. Consider the function $\mathcal{F}_{\exists\Diamond p}$ that maps regions of K to regions of K:

$$\mathcal{F}_{\exists\Diamond p}(\sigma) \;=\; [\![p]\!]_K \cup pre_K(\sigma).$$

Observe that the operator $\exists\bigcirc$ of CTμ is same as the function *pre*, and hence, $[\![\mu\mathbf{x}.\,(p \vee \exists\bigcirc\,\mathbf{x})]\!]_K$ is the least fixpoint of the function $\mathcal{F}_{\exists\Diamond p}$. First, we show that the characteristic region $[\![\exists\Diamond p]\!]_K$ is a fixpoint of the function $\mathcal{F}_{\exists\Diamond p}$:

$$[\![\exists\Diamond p]\!] \;\Leftrightarrow\; [\![p]\!] \cup pre([\![\exists\Diamond p]\!]).$$

This is established from the definition of the CTL operator $\exists\Diamond$. Second, we show that the region $[\![\exists\Diamond\sigma]\!]$ is contained in all fixpoints of $\mathcal{F}_{\exists\Diamond p}$: for all regions $\sigma \subseteq \Sigma$ and all states $s \in \Sigma$, if $\sigma = [\![p]\!] \cup \mathit{pre}(\sigma)$ and $s \models \exists\Diamond p$, then $s \in \sigma$.

So assume that $\sigma = [\![p]\!] \cup \mathit{pre}(\sigma)$ and that there is a source-s trajectory $\overline{s}_{0..n}$ of K such that $s_n \models p$. Then $s_n \in \sigma$, and by backward induction on $\overline{s}_{0..n}$, $s_i \in \sigma$ for all $0 \leq i \leq n$. ■

Exercise 11.11 {T2} [μ-calculus vs. CTL] Which CTL formula is equivalent to the CTμ formula $\mu\mathbf{x}.\, \exists\bigcirc\, (\mathbf{x} \vee p)$? ■

Remark 11.10 [Fixpoint characterization of $\exists\Diamond$] Let ϕ be a CTμ formula and ψ be a CTL formula. If ϕ and ψ are equivalent, then so are the formulas $\mu\mathbf{x}.\, (\phi \vee \exists\bigcirc\, \mathbf{x})$ and $\exists\Diamond\psi$. ■

To obtain fixpoint characterization of the possibly-until connective $\exists\mathcal{U}$, observe the following equivalence:

$$(\phi\, \exists\mathcal{U}\, \psi) \;\leftrightarrow\; \psi \vee (\phi \wedge (\phi\, \exists\mathcal{U}\, \psi).$$

A state satisfies $(\phi\, \exists\mathcal{U}\, \psi)$ if either it satisfies ψ, or it satisfies ϕ and has a successor that is already known to satisfy $(\phi\, \exists\mathcal{U}\, \psi)$. This suggests that $\exists\mathcal{U}$ can be defined as a μ-formula:

Proposition 11.7 [Fixpoint characterization of $\exists\mathcal{U}$] *Let ϕ_1 and ϕ_2 be a* CTμ *formulas, and let ψ_1 and ψ_2 be* CTL *formulas. If the formulas ϕ_1 and ψ_1 are equivalent, and the formulas ϕ_2 and ψ_2 are equivalent, then so are the formulas $\mu\mathbf{x}.\, (\phi_2 \vee (\phi_1 \wedge \exists\bigcirc\, \mathbf{x}))$ and $\psi_1 \exists\mathcal{U} \psi_2$.*

Finally, let us consider the possibly-always connective $\exists\Box$. A state all of whose successors do not satisfy p cannot satisfy $\exists\Box p$. A state all of whose successors are known not to satisfy $\exists\Box p$ cannot satisfy $\exists\Box p$. This suggests a characterization of $\exists\Box p$ as a greatest fixpoint: $[\![\exists\Box p]\!]$ is the maximal region each of whose states satisfies p and has at least one successor satisfying p.

Proposition 11.8 [Fixpoint characterization of $\exists\Box$] *Let ϕ be a* CTμ *formula and ψ be a* CTL *formula. If ϕ and ψ are equivalent, then so are the formulas $\nu\mathbf{x}.\, (\phi \wedge \exists\bigcirc\, \mathbf{x})$ and $\exists\Box\psi$.*

Theorem 11.4 [From CTL to CTμ] *Every* CTL *formula ϕ is equivalent to an alternation-free* CTμ *formula of length $O(|\phi|)$.*

Exercise 11.12 {T4} [Correctness of translation from CTL to CTμ] Prove Propositions 11.7 and 11.8, and then, prove Theorem 11.4 using Propositions 11.5, 11.7 and 11.8. ■

We can define temporal operators in CTμ:

$$\begin{array}{lll}
\exists\Diamond\phi & \text{for} & \mu\mathbf{x}.\,(\phi \vee \exists\bigcirc\mathbf{x}); \\
\phi_1\exists\mathcal{U}\phi_2 & \text{for} & \mu\mathbf{x}.\,(\phi_2 \vee (\phi_1 \wedge \exists\bigcirc\mathbf{x})); \\
\exists\Box\phi & \text{for} & \nu\mathbf{x}.\,(\phi \wedge \exists\bigcirc\mathbf{x}); \\
\forall\Diamond\phi & \text{for} & \mu\mathbf{x}.\,(\phi \vee \forall\bigcirc\mathbf{x}); \\
\forall\Box\phi & \text{for} & \nu\mathbf{x}.\,(\phi \wedge \forall\bigcirc\mathbf{x}); \\
\phi_1\forall\mathcal{U}\phi_2 & \text{for} & \mu\mathbf{x}.\,(\phi_2 \vee (\phi_1 \wedge \forall\bigcirc\mathbf{x})).
\end{array}$$

Notice that the CTμ formula ($\nu\mathbf{x}.\,\phi \vee \exists\bigcirc\mathbf{x}$) is equivalent to *true*, and the CTμ formula ($\mu\mathbf{x}.\,\phi \wedge \exists\bigcirc\mathbf{x}$) is equivalent to *false*.

Distinguishing power of CTμ

In Section 5.4 we established that bisimilarity is a fully abstract semantics for STL; that is, two bisimilar states satisfy the same set of STL formulas, and if two states are not bisimilar then some STL formula distinguishes between them. Since (alternation-free) μ-calculus is as expressive as STL, it follows that it can distinguish between states that are not bisimilar. Furthermore, μ-calculus cannot distinguish between bisimilar states.

Proposition 11.9 [CTμ abstraction] *Bisimilarity is an abstract semantics for* CTμ*.*

Exercise 11.13 {`T4`} [CTμ abstraction] Prove Proposition 11.9. ■

Thus, the distinguishing powers of a variety of state logics, such as STL$^{\bigcirc}$, STL, CTL, CTμ, CTμ^1, coincide, and all these logics are more distinguishing than the structure logic SAL.

Alternation-free CTμ is more expressive than CTL

The alternation-free μ-calculus is more expressive than CTL. There are at least two types of properties that can be specified in CTμ^1, but not in CTL. The first type concerns the inability of CTL to count, while the second one concerns inability of CTL to specify game-like properties.

Proposition 11.10 [CTL vs. CTμ^1] *Let p be a proposition. No* CTL *formula is equivalent to the* CTμ^1 *formula $\nu\mathbf{x}.\,(p \wedge \forall\bigcirc\forall\bigcirc\mathbf{x})$.*

Proof. The formula $\nu\mathbf{x}.\,(p \wedge \forall\bigcirc\forall\bigcirc\mathbf{x})$ is satisfied by a state s of an observation structure K iff for every source-s ω-trajectory $\underline{s}$, $s_i \models p$ for all even numbers i. Thus, the formula $\nu\mathbf{x}.\,(p \wedge \forall\bigcirc\forall\bigcirc\mathbf{x})$ is equivalent to the SAL formula ϕ_{even} (see proof of Theorem 6.2). We already know that no STL formula is equivalent to ϕ_{even}. The same proof can be extended to establish that no CTL formula is equivalent to ϕ_{even}. ■

Consider an observation structure K with three observations a, b, and c. Consider the following two-player game between a protagonist and an adversary. The positions of the game is described by a state of K. If the current position s has observation c, then the protagonist wins the game. Otherwise, the position of the game is updated to some successor of s. If the observation of s is a, then the protagonist chooses the successor position, and if the observation of s is b, then the adversary chooses the successor position. Given an initial position, the protagonist wins if it has a strategy to force the game to a state with observation c. Thus, the described game is a standard AND-OR game, where states with observations c are winning positions, states with observation a are OR-positions and states with observation b are AND-positions. Let σ be the set of winning initial positions for the protagonist. To get a fixpoint characterization of σ, observe that (1) all states with observation c belong to σ, (2) for a state s with observation a, if some successor of s is already known to be winning, then the protagonist can win from s also, and (3) for a state s with observation b, if all successors of s are already known to be winning, then the protagonist can win from s also. Thus, σ is the smallest region that contains $[\![c]\!]$, $[\![a \wedge \exists\bigcirc\sigma]\!]$, and $[\![b \wedge \forall\bigcirc\sigma]\!]$. Thus, the set of winning positions is described by the alternation-free formula $\mu\mathbf{x}.\,(c \vee (a \wedge \exists\bigcirc\mathbf{x}) \vee (b \wedge \forall\bigcirc\mathbf{x}))$. It turns out that the set of winning positions cannot be characterized using a CTL formula.

Proposition 11.11 [CTL vs. CTμ^1] *Let p and q be propositions. No* CTL *formula is equivalent to the* CTμ^1 *formula* $\mu\mathbf{x}.\,(q \vee (p \wedge \exists\bigcirc\mathbf{x}) \vee (\neg p \wedge \forall\bigcirc\mathbf{x}))$.

Fair region for a single Büchi

We turn our attention to characterization of fair regions using μ-calculus. Let (K, F) be a fair structure. The fair region σ^F of K consists of states from which there exists a F-fair ω-trajectory. For now, let us assume that F contains a single Büchi constraint specified by the state predicate p. Thus, a state s of K belongs to σ^F iff there exits a source-s ω-trajectory that contains infinitely many states that satisfy p. We use the operator $\Box\Diamond$ to denote infinite repetition:

$s \models_K \exists\Box\Diamond p$ iff there exists a source-s ω-trajectory $\underline{s}$ of K such that
$s_i \models_K p$ for infinitely many positions i.

The formula $\exists\Box\Diamond p$ can be expressed in CTμ using nested fixpoints: it is equivalent to the formula $\nu\mathbf{x}.\,\mu\mathbf{y}.\,\exists\bigcirc\,((\mathbf{x} \wedge p) \vee \mathbf{y})$, which can also be written as $\nu\mathbf{x}.\,\exists\Diamond^+(\mathbf{x} \wedge p)$. That is, $[\![\exists\Box\Diamond p]\!]$ is the maximal region σ such that from every state in σ, some state in $\sigma \cap [\![p]\!]$ is reachable in one or more steps. The i-th approximation in the computation of $\nu\mathbf{x}.\,\exists\Diamond^+(\mathbf{x} \wedge p)$ contains all states from which there exists a trajectory containing i states satisfying p:

$$[\![\nu\mathbf{x}.\,\exists\Diamond^+(\mathbf{x} \wedge p)]\!] \;=\; [\![\mathit{true}]\!] \cap [\![\exists\Diamond^+ p]\!] \cap [\![\exists\Diamond^+(p \wedge \exists\Diamond^+ p)]\!] \cap \cdots$$

Proposition 11.12 [Fixpoint characterization of $\exists\Box\Diamond$] *The* Ctμ *formula* $\nu\mathbf{x}.\,\mu\mathbf{y}.\,\exists\bigcirc((\mathbf{x} \wedge p) \vee \mathbf{y})$ *is equivalent to* $\exists\Box\Diamond p$.

Proof. Let K be an observation structure. Consider the function $\mathcal{F}_{\exists\Box\Diamond p}$ that maps regions of K to regions of K:

$$\mathcal{F}_{\exists\Box\Diamond p}(\sigma) \;=\; \mathit{pre}^{+}(\sigma \cap [\![p]\!]).$$

It suffices to show that the region $[\![\exists\Box\Diamond p]\!]$ is the maximal fixpoint of the function $\mathcal{F}_{\exists\Box\Diamond p}$. First, we show that $[\![\exists\Box\Diamond p]\!]$ is a fixpoint of $\mathcal{F}_{\exists\Box\Diamond p}$:

$$[\![\exists\Box\Diamond p]\!] \;\Leftrightarrow\; \mathit{pre}^{+}([\![\exists\Box\Diamond p]\!] \cap [\![p]\!]).$$

To establish this, for all states s, there is a source-s p-fair trajectory iff there exists a state t such that (i) t is reachable from s in one or more steps (i.e. $s \in \mathit{pre}^{+}(t)$), (ii) t satisfies p, and (iii) there is a source-t p-fair trajectory. Second, we need to establish that every fixpoint of $\mathcal{F}_{\exists\Box\Diamond p}$ is contained in $[\![\exists\Box\Diamond p]\!]$: for all regions σ and all states s,

$$\text{if } \sigma = \mathit{pre}^{+}(\sigma \cap [\![p]\!]) \text{ and } s \in \sigma \text{ then } s \models \exists\Box\Diamond p.$$

So assume that $\sigma = \mathit{pre}^{+}(\sigma \cap [\![p]\!])$ and $s \in \sigma$. We construct an infinite sequence of states $s_0 s_1 \ldots$ as follows. Let $s_0 = s$. Given $s_i \in \sigma$, choose s_{i+1} such that $s_{i+1} \in \sigma$ and $s_{i+1} \models p$ and $s_{i+1} \in \mathit{post}^{+}(s_i)$ (such a state exists since $\sigma = \mathit{pre}^{+}(\sigma \cap [\![p]\!])$). It follows that there exists a source-s ω-trajectory containing infinitely many states satisfying p. ■

Exercise 11.14 {T2} [$\exists\Box\Diamond$ in μ-calculus] Is the formula $\nu\mathbf{x}.\,\exists\Diamond(p \wedge \mathbf{x})$ equivalent to $\exists\Box\Diamond p$? Is the formula $\nu\mathbf{x}.\,\exists\Diamond(p \wedge \exists\bigcirc\mathbf{x})$ equivalent to $\exists\Box\Diamond p$? ■

Exercise 11.15 {T2} [Fixpoint characterization of $\exists\Box p$ in Büchi structures] Let $\mathcal{K} = (K, F)$ be a fair structure such that F contains a single Büchi constraint specified by the predicate q. Write a Ctμ formula ϕ such that $[\![\phi]\!]_K$ equals $[\![\exists\Box p]\!]_{\mathcal{K}}$. That is, $s \models_K \phi$ iff there is a source-s F-fair ω-trajectory $\underline{s}$ with $s_i \models p$ for all $i \geq 0$. ■

Exercise 11.16 {T3} [$\exists\Diamond\Box$ in μ-calculus] Given a state s of an observation structure K, and a state predicate p, define $s \models_K \exists\Diamond\Box p$ iff there exist a source-s ω-trajectory $\underline{s}$ and an integer $i \geq 0$ such that $s_j \models_K p$ for all $j \geq i$. Write Ctμ formula that is equivalent to $\exists\Diamond\Box p$. ■

Now let us consider the case when the fairness assumption contains a single weak-fairness constraint specified by an action α. Suppose the action α is specified by the action predicate $p \wedge q'$; that is, $s \xrightarrow{\alpha} t$ iff $s \models p$ and $t \models q$. We wish to characterize the fair region by a Ctμ formula. A state s of K satisfies the Ctμ fromula $\nu\mathbf{x}.\,\exists\Diamond(p \wedge \exists\bigcirc(q \wedge \mathbf{x}))$ iff there is a source-s ω-trajectory $\underline{s}$ such that for infinitely many positions i, $s_i \models p$ and $s_{i+1} \models q$, that is, iff there is a source-s α-fair trajectory. This leads to the characterization of fair regions when fairness contains a single weak constraint. In general, the action α will be specified using a disjunction $\vee 0 \leq i \leq k.\, p_i \wedge q'_i$: $s \overset{\alpha}{\to} t$ iff for some $0 \leq i \leq k$, $s \models p_i$ and $t \models q_i$.

Proposition 11.13 [Single weak constraint in C<span>T</span>μ] *Let $\mathcal{K} = (K, F)$ be a fair structure where F contains a single weak constraint α. Let $p_0, \ldots p_k$ and $q_0 \ldots q_k$ be state predicates of K such that $\alpha = [\![\vee 0 \leq i \leq k.\, p_i \wedge q'_i]\!]_K$. Then, the fair region of $\mathcal{K}$ equals*

$$[\![\nu\mathbf{x}.\, \exists\Diamond \ \vee\ 0 \leq i \leq k.\, (p_i \wedge \exists\bigcirc (q_i \wedge \mathbf{x}))]\!]_K.$$

Exercise 11.17 {T3} [Single weak constraint in μ-calculus] Prove Proposition 11.13. ■

Exercise 11.18 {T3} [Multiple Büchi constraints] Consider a Büchi structure (K, F) where F contains k Büchi constraints specified by predicates $p_1, \ldots p_k$. Show that the fair region is characterized by the C<span>T</span>μ formula

$$\nu\mathbf{x}.\, \exists\Diamond^{+}(p_1 \wedge \exists\Diamond^{+}(p_2 \wedge \cdots \wedge \exists\Diamond^{+}(p_k \wedge \mathbf{x}) \cdots)).$$

■

Consider a module P, and let a be an update choice of an atom U of P. The availability action $avail_a$ of the choice a is be described by a predicate q_{avail_a} over $\mathsf{read}X_U \cup \mathsf{await}X'_U$. The execution action $exec_a$ of the choice a is described by a predicate q_{exec_a} over $\mathsf{read}X_U \cup \mathsf{await}X'_U \cup \mathsf{ctr}X'_U$. The weak-fairness constraint of α is, then, described by the predicate $q_{exec_a} \vee \neg q_{avail_a}$. This predicate can be rewritten to a form required by Proposition 11.13.

Example 11.3 [Fairness constraints for mutual exclusion] Recall the fair module *FairPete* from Figure 8.5. The module has four weak-fairness constraints specified by the choices α_1, β_1, α_2, and β_2. Let us just consider the choice α_1 The weak-fairness constraint corresponding to the update choice α_1 is specified by the action $exec_{\alpha_1} \cup (\rightarrow \backslash avail_{\alpha_1})$. The execution action $exec_{\alpha_1}$ is specified by the predicate $pc_1 = inC \wedge pc'_1 = outC$, and the availability action $avail_{\alpha_1}$ is specified by the predicate $pc_1 = inC$. It follows that the fairness constraint corresponding to the choice α_1 is the disjunction

$$(pc_1 = inC \wedge pc'_1 = outC) \vee (pc_1 \neq inC).$$

The corresponding fair region, then, is expressed by the C<span>T</span>μ formula

$$\nu\mathbf{x}.\, \exists\Diamond\, [\, (pc_1 = inC \wedge \exists\bigcirc (pc_1 = outC \wedge \mathbf{x})) \vee (pc_1 \neq inC \wedge \exists\bigcirc \mathbf{x})\,].$$

■

While the operator $\exists\Box\Diamond$ is specifiable in CTμ^2, it is not specifiable in CTL.

Proposition 11.14 [CTL cannot express $\exists\Box\Diamond$] *There is no* CTL *formula that is equivalent to* $\exists\Box\Diamond p$.

Proof. Suppose there is a CTL formula ϕ such that for every structure K, $[\![\exists\Box\Diamond p]\!]_K$ equals $[\![\phi]\!]_K$. Suppose the length of ϕ is k. Consider the observation structure of Figure 11.1. States that satisfy the atomic formula p are labeled with p. We first prove the following lemma.

Lemma A. For every CTL formula ψ, for all integers $|\psi| - 1 \leq i \leq j \leq k$, $s_i \models \psi$ iff $s_j \models \psi$ and $t_i \models \psi$ iff $t_j \models \psi$.

Proof of Lemma A. The proof is by induction on the structure of the formula ψ. For $0 \leq i \leq k$, all the states s_i satisfy the same atomic formulas, and so do all the states t_i. Hence, the lemma holds if ψ is an atomic formula. When $\psi = \neg\chi$, or when $\psi = \chi_1 \vee \chi_2$, the lemma follows from induction.

Case $\psi = \exists\bigcirc\chi$. For $1 \leq i \leq k$, $s_i \models \psi$ iff $t_{i-1} \models \chi$, and $t_i \models \psi$ iff $t_i \models \chi$ or $s_{i-1} \models \chi$. For $|\psi| - 1 \leq i \leq j \leq k$, $i \geq 1$ and $|\chi| \leq i - 1 \leq j - 1 \leq k$. By induction, $t_{i-1} \models \chi$ iff $t_{j-1} \models \chi$; $s_{i-1} \models \chi$ iff $s_{j-1} \models \chi$; and $t_i \models \chi$ iff $t_j \models \chi$.

Case $\psi = \exists\Box\chi$. For $1 \leq i \leq k$, $t_i \models \psi$ iff $t_i \models \chi$, and $s_i \models \psi$ iff $s_i \models \chi$ and $t_{i-1} \models \chi$. Now we can proceed as in the previous case.

Case $\psi = \chi_1 \exists\mathcal{U} \chi_2$. Left as an exercise.

Corollary B. For every subformula ψ of ϕ, $s_k \models \psi$ iff $s_{k-1} \models \psi$.

The next lemma implies that $s_k \models \phi$ iff $u \models \phi$. This yields a contradiction, because $s_k \not\models \exists\Box\Diamond p$, but $u \models \exists\Box\Diamond p$.

Lemma C. For every subformula ψ of ϕ, $s_k \models \psi$ iff $u \models \psi$, and $t_k \models \psi$ iff $v \models \psi$.

Proof of Lemma C. The proof is by induction on the structure of the formula ψ. When ψ is an atomic formula, the lemma is immediate as the states s_k and u, and states t_k and v have identical observations. When $\psi = \neg\chi$, or when $\psi = \chi_1 \vee \chi_2$, the lemma follows from induction.

Case $\psi = \forall\bigcirc\chi$. $s_k \models \psi$ iff $t_k \models \chi$ iff, by induction, $v \models \chi$ iff $u \models \psi$. $t_k \models \psi$ iff both t_k and s_{k-1} satisfy χ iff, by Corollary B, all of s_k, t_k, and s_{k-1} satisfy χ iff, by induction, all of u, v, and s_{k-1} satisfy χ iff $v \models \psi$.

Case $\psi = \exists\Box\chi$. $s_k \models \psi$ iff both s_k and t_k satisfy χ iff, by induction, both u and v satisfy χ iff $u \models \psi$. $t_k \models \psi$ iff $t_k \models \chi$ iff, by induction, $v \models \chi$ iff $v \models \psi$.

Case $\psi = \chi_1 \exists\mathcal{U} \chi_2$. Left as an exercise. ■

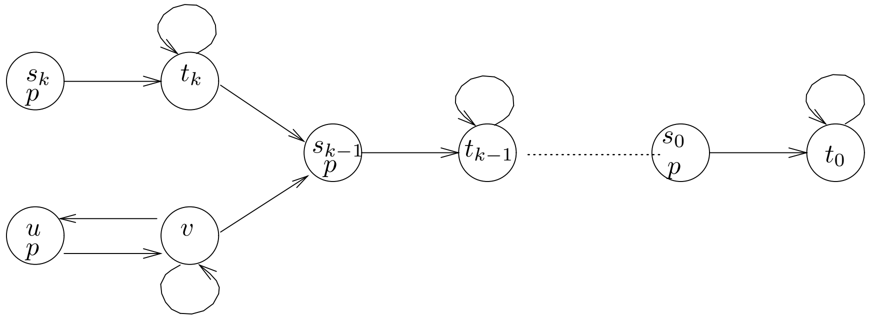


Figure 11.1: $\exists\Box\Diamond$ is not expressible in Ctl

Exercise 11.19 {T4} [Alternation-free μ-calculus cannot express $\exists\Box\Diamond$] Prove that no formula of Ctμ^1 is equivalent to $\exists\Box\Diamond p$. ■

Remark 11.11 [Hierarchy of expressiveness] For every integer $i \geq 0$, the fragment Ctμ^{i+1} is more expressive than the fragment Ctμ^i. Thus, the expressiveness of Ctμ strictly increases with increasing alternation depth. ■

Specifying fair regions

Now we turn our attention to strong fairness constraints. Let F be a fairness assumption for an observation structure K. Suppose each fairness constraint $f \in F$ is a Streett constraint defined by state predicates p and q: an ω-trajectory $\underline{s}$ is f-fair iff if it is q-fair or not p-fair.

Exercise 11.20 {T3} [Fixpoint characterization of single Streett constraint] Consider an observation structure K and two state predicates p and q of K. Show that a state s of K satisfies the Ctμ formula $\exists\Diamond(\exists\Box\neg p \vee \exists\Box\Diamond q)$ iff there exists a source-s (p,q)-fair trajectory of K. ■

Exercise 11.20 suggests characterization of fair regions when the fairness assumption has a single fairness constraint. It can be generalized to multiple Streett constraints. Let F be a Streett assumption for an observation structure K. Then, a state s belongs to the fair region of K iff there exists a state $t \in \mathit{post}^*(s)$, a subset F' of F, and a source-t ω-trajectory (1) that is q-fair for every $(p,q) \in F'$, and (2) all of whose states satisfy $\neg p$ for every $(p,q) \in F\backslash F'$. This suggests a Ctμ formula whose length is exponential in the number of Streett constraints in F. However, a polynomial translation is possible.

Proposition 11.15 [Emerson-Lei Fixpoint characterization of Streett assumption] *Let K be an observation structure, and let F be a Streett assumption for K. Then, the fair region of (K,F) is the characteristic region of the formula*

$$\exists\Diamond\, \nu\mathbf{x}.\ \bigwedge(p,q) \in F.\,[\,\exists\bigcirc\,(\mathbf{x}\exists\mathcal{U}(q \,\wedge\, \mathbf{x}))\,\vee\,(\neg p \,\wedge\, \exists\bigcirc\,\mathbf{x})\,].$$

Proof. Let K be an observation structure. Let $F = \{(\neg p_1, q_1), \ldots (\neg p_k, q_k)\}$ be a Streett assumption with k Streett constraints. An ω-trajectory $\underline{s}$ is F-fair iff for $1 \leq i \leq k$, either $\underline{s}$ is q_i-fair or it has a suffix containing only p_i-states. This requirement on the ω-trajectory is expressed by the formula

$$\phi \;=\; \bigwedge 1 \leq i \leq k.\,(\Diamond\Box p_i \;\vee\; \Box\Diamond q_i).$$

The fair region is characterized by the formula $\exists\phi$. Define the formula

$$\phi' \;=\; \bigwedge 1 \leq i \leq k.\,(\Box p_i \;\vee\; \Box\Diamond q_i).$$

An ω-trajectory $\underline{s}$ satisfies ϕ' iff for $1 \leq i \leq k$, either $\underline{s}$ is q_i-fair or contains only p_i-states. A state s satisfies $\exists\phi'$ iff there is source-s ω-trajectory satisfying ϕ'. The next two lemmas follow from the definitions of the formulas ϕ and ϕ'.

Lemma A. $[\![\exists\Diamond\exists\phi']\!] = [\![\exists\phi]\!]$.

Lemma B. $[\![\exists\Diamond\exists\phi]\!] = [\![\exists\phi]\!]$.

Now consider the function $\mathcal{F}$ that maps regions of K to regions of K:

$$\mathcal{F}(\sigma) \;=\; \bigwedge 1 \leq i \leq k.\,[\,\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma)) \,\vee\, (p_i \,\wedge\, \exists\bigcirc\,\sigma)\,].$$

Lemma C. If σ is a fixpoint of $\mathcal{F}$ then $\sigma \,\subseteq\, [\![\exists\phi]\!]$.

Proof of Lemma C. Let σ be a fixpoint of $\mathcal{F}$. Consider $s \in \sigma$. We will construct a source-s ω-trajectory that satisfies ϕ. For every $j \geq 0$, we define a state s_j, and a finite trajectory from s_j to s_{j+1} containing only σ-states. Let $s_0 = s \in \sigma$. Consider s_j in σ. Let i be j *mod* k. Since $\sigma = \mathcal{F}(\sigma)$, s_j satisfies $\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma))$ or $p_i \,\wedge\, \exists\bigcirc\,\sigma$. If s_j satisfies $\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma))$, then there exists a source-s_j trajectory $\overline{t}_{0\ldots n}$ with $n > 0$ containing only σ-states such that $t_n \models q_i$. Choose $s_{j+1} = t_n$. If s_j does not satisfy $\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma))$, then it must satisfy $p_i \,\wedge\, \exists\bigcirc\,\sigma$, and choose s_{j+1} to be a successor of s_j in σ.

Let $\underline{t}$ be the source-s ω-trajectory obtained by concatenating the finite trajectories from s_j to s_{j+1} defined above. Every state in $\underline{s}$ belongs to σ. We wish to establish that $\underline{t}$ satisfies ϕ. Consider $1 \leq i \leq k$. For every $n \geq 0$, if s_{i+kn} satisfies $\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma))$ then s_{i+kn+1} satisfies q_i. Suppose that there are infinitely many n such that s_{i+kn} satisfies $\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma))$. Then, by construction, $\underline{t}$ is q_i-fair. Otherwise, there exists $n \geq 0$ such that $s_{i+kn'}$ does not satisfy $\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma))$ for $n' \geq n$. Since every state in $\underline{t}$ satisfies σ, it follows that there exists $n \geq 0$ such that $t_{n'}$ does not satisfy $\exists\bigcirc\,(\sigma\,\exists\mathcal{U}(q_i \,\wedge\, \sigma))$ for $n' \geq n$. Since σ is a fixpoint of $\mathcal{F}$, it follows that $t_{n'}$ satisfies $p_i \,\wedge\, \exists\bigcirc\,\sigma$, and hence, $\underline{t}$ satisfies $\Diamond\Box p_i$.

Lemma D. $[\![\exists\phi']\!] \,\subseteq\, \mathcal{F}([\![\exists\phi']\!])$.

Proof of Lemma D. Consider a state $s \in [\![\exists\phi']\!]$. There exists a source-$s$ ω-trajectory $\underline{s}$ such that for $1 \leq i \leq k$, either $\underline{s}$ is q_i-fair or contains only p_i-states.

Every suffix of $\underline{s}$ satisfies ϕ', and hence, $s_j \models \exists\phi'$ for all $j \geq 0$. We wish to establish that s satisfies $\mathcal{F}([\![\exists\phi']\!])$. Consider $1 \leq i \leq k$. We need to prove that s satisfies either $\exists\bigcirc([\![\exists\phi']\!]\exists\mathcal{U}(q_i \wedge [\![\exists\phi']\!]))$ or $p_i \wedge \exists\bigcirc[\![\exists\phi']\!]$. If $\underline{s}$ is q_i-fair, then s_1 satisfies $[\![\exists\phi']\!]\exists\mathcal{U}(q_i \wedge [\![\exists\phi']\!])$; otherwise $\underline{s}$ contains only p_i-states, and s satisfies $p_i \wedge \exists\bigcirc[\![\exists\phi']\!]$.

Now we proceed to show that $\exists\phi$ is equivalent to $\exists\Diamond\,\nu\mathbf{x}.\,\mathcal{F}(\mathbf{x})$. Suppose $s \models \exists\Diamond\nu\mathbf{x}.\,\mathcal{F}(\mathbf{x})$. By Lemma C, if a state satisfies $\nu\mathbf{x}.\,\mathcal{F}(\mathbf{x})$ then it also satisfies $\exists\phi$. Hence, $s \models \exists\Diamond\exists\phi$. By Lemma A, $s \models \exists\phi$. Conversely, suppose $s \models \exists\phi$. By Lemma B, $s \models \exists\Diamond\exists\phi'$. By Lemma D, $[\![\exists\phi']\!]$ is contained in the maximla fixpoint of $\mathcal{F}$. Hence, $s \models \exists\Diamond\nu\mathbf{x}.\,\mathcal{F}(\mathbf{x})$. ■

Exercise 11.21 {T3} [Fixpoint characterization of fairness assumption] Consider a fair graph (K, F). Every constraint is F is a pair of actions, and suppose every action α is represented by state predicates $p_0, \ldots p_k$ and $q_0 \ldots q_k$ of K such that $\alpha = [\![\vee\, 0 \leq i \leq k.\, p_i \wedge q_i']\!]_K$. Given this representation of actions, write a Ctμ formula that characterizes the faire region of (K, F). ■

Thus, the fair region of a fair graph can be characterized in μ-calculus using formulas of alternation depth 2. To characterize the region $[\![\exists\Box p]\!]_{\mathcal{K}}$ of fair structure, only a slight modification is required. For instance, for a Streett assumption F, the characteristic region $[\![\exists\Box p]\!]$ equals

$$p\,\exists\mathcal{U}\,\nu\mathbf{x}.\,p \wedge \bigwedge(q, r) \in F.\,[\,\exists\bigcirc(\mathbf{x}\exists\mathcal{U}(r \wedge \mathbf{x})) \vee (\neg q \wedge \exists\bigcirc\mathbf{x})\,].$$

Theorem 11.5 [From Ctl over fair structures to Ctμ] *For every* Ctl *formula ϕ and a fair structure $\mathcal{K} = (K, F)$, there exists a formula ψ of* Ctμ^2 *such that $[\![\phi]\!]_{\mathcal{K}} = [\![\psi]\!]_K$ and $|\psi| = O(|\phi| \cdot |F|)$.*

Let a be an update choice of a module P. The strong-fairness constraint of a is the pair $(avail_a, exec_a)$ of actions. After writing the two actions $avail_a$ and $exec_a$ in the form stipulated by Exercise 11.21, we can write a Ctμ formula that characterizes the fair region of the fair module.

11.3.3 Model checking

We are given a closed Ctμ formula ϕ and ϕ-structure K, we are required to check if all the initial states of K satisfy ϕ. For this purpose, we compute the characteristic region $[\![\phi]\!]_K$. Assume that the fomula ϕ has no name-conflicts in the use of region variables: every variable $\mathbf{x}$ is quantified by a unique fixpoint operator.

The characteristic region $[\![\phi]\!]_K$ can be computed using a recursive function *Eval*. The table $\mathbf{E}$ stores, for every region variable $\mathbf{x}$, a region $\mathbf{E}(\mathbf{x})$ of K. The function *Eval* takes a formula ψ as an argument, and returns the set of states satisfying

ψ using the table $\mathbf{E}$ to evaluate free variables. If ψ is an atomic formula, the computation of $\mathit{Eval}(\psi)$ is immediate. If ψ is a conjunction of formulas, then *Eval* calls itself recursively on the conjuncts, and returns the intersection of the results. The case of disjunction is similar. When ψ equals $\exists\bigcirc\,\chi$, *Eval* calls itself recursively on χ, and returns the set of predecessors of the result. The evaluation of $\forall\bigcirc\,\chi$ uses the fact that $\exists\bigcirc$ and $\forall\bigcirc$ are duals of each other: $\forall\bigcirc = \neg\exists\bigcirc\,\neg$.

To evaluate a subformula $\mu\mathbf{x}.\,\chi$, the minimal fixpoint is computed by evaluating χ repeatedly. In the first iteration, $\mathbf{E}(\mathbf{x})$ is chosen to be the empty set, and in each successive iteration, $\mathbf{E}(\mathbf{x})$ is chosen to be the value of $\mathit{Eval}(\chi)$ from the previous iteration. The fixpoint is reached when two consecutive iterations yield the same result. The number of iterations is bounded by the number of states in the observation structure. The evaluation of $\nu\mathbf{x}.\,\chi$ is similar, but in this case, in the first iteration, $\mathbf{E}(\mathbf{x})$ is chosen to be the set of all states. A naive implementation of this recursive scheme would make the depth of recursion equal to the nesting depth of the formula, resulting in an algorithm with time complexity $O(n^k)$, where k is the nesting depth of the formula. Two improvements are possible.

First, every closed formula needs to be evaluated just once. For example, consider the formula $\mu\mathbf{x}.\,\psi$, where χ is a closed fixpoint subformula of ψ. The invocation $\mathit{Eval}(\mu\mathbf{x}.\,\psi)$ results in repeated calls to $\mathit{Eval}(\psi)$, and hence to $\mathit{Eval}(\chi)$, each time with a different value of $\mathbf{E}(\mathbf{x})$. However, χ is a closed formula, and its value does not depend on $\mathbf{E}(\mathbf{x})$. Consequently, it needs to be evaluated only once. For this purpose, we use a hash-table *Done* that stores the results of evaluating closed formulas. Upon invocation, *Eval* checks if its input formula is closed, and if so, whether it has already been evaluated by consulting the hash-table.

Second, consider the formula $\mu\mathbf{x}.\,\phi$, where $\psi = \mu\mathbf{y}.\,\chi$ is a disjunct of ϕ. Let σ_0 be the empty set. The first iteration in $\mathit{Eval}(\mu\mathbf{x}.\,\phi)$ calls $\mathit{Eval}(\phi)$ with $\mathbf{E}(\mathbf{x}) = \sigma_0$. This involves evaluation of the fixpoint formula ψ, which itself involves an iterative computation of χ during which the region $\mathbf{E}(\mathbf{y})$ keeps growing. Let $\tau_0 = [\![\psi]\!]$ and $\sigma_1 = [\![\phi]\!]$ with $\mathbf{E}(\mathbf{x}) = \sigma_0$. If σ_0 is a strict subset of σ_1, the second iteration in $\mathit{Eval}(\phi)$ calls $\mathit{Eval}(\psi)$ with $\mathbf{E}(\mathbf{x}) = \sigma_1$. This would result in repeated evaluation of χ starting with $\mathbf{E}(\mathbf{y})$ to be the empty set until the value of $\mathbf{E}(\mathbf{y})$ becomes stable. Let $\tau_1 = [\![\psi]\!]$ with $\mathbf{E}(\mathbf{x}) = \sigma_1$. However, due to the monotonicity property, $\tau_0 \subseteq \tau_1$. This implies that, instead of computing τ_1 as a fixpoint starting with $\mathbf{E}(\mathbf{y})$ as empty set, we can speed up the convergence by choosing $\mathbf{E}(\mathbf{y})$ to be τ_0 in the first iteration. That is, there is no need to reinitialize $\mathbf{E}(\mathbf{y})$ from τ_0 to the empty set when $\mathbf{E}(\mathbf{x})$ is updated from σ_0 to σ_1. With this improved policy, $\mathit{Eval}(\chi)$ is called only n times, rather than n^2 times. The validity of this optimization is captured by the following proposition.

Proposition 11.16 [Optimization in C<span style="font-variant:small-caps">t</span>μ model checking] *Let K be an observation structure with finitely branching transition relation, and ϕ be a* C<span style="font-variant:small-caps">t</span>μ *formula. Let $\mathbf{E}$ and $\mathbf{E}'$ be region environments such that for every region variable $\mathbf{y}$ that is free in $\mu\mathbf{x}.\,\phi$, $\mathbf{E}(\mathbf{y}) \subseteq \mathbf{E}'(\mathbf{y})$. Then,*

$$[\![\mu\mathbf{x}.\,\phi]\!]_{\mathbf{E}'} \; = \bigcup i \in \quad .(\mathcal{F}^{\phi}_{\mathbf{E}'})^i([\![\mu\mathbf{x}.\,\phi]\!]_{\mathbf{E}}),$$

and

$$[\![\nu\mathbf{x}.\,\phi]\!]_{\mathbf{E}} \; = \bigcap i \in \quad .(\mathcal{F}^{\phi}_{\mathbf{E}})^i([\![\nu\mathbf{x}.\,\phi]\!]_{\mathbf{E}'}).$$

Proof. We consider the case corresponding to the least fixpoints. Whenever a function $\mathcal{F}$ is $\bigcup$-continuous, by Kleene fixpoint theorem, its least fixpoint can be computed by repeatedly applying $\mathcal{F}$ to the minimal element–the empty set: $\mu\mathcal{F} = \bigcup i \in \quad .\,\mathcal{F}^i(\emptyset)$. A slight generalization of the Kleene fixpoint theorem states that the least fixpoint of $\mathcal{F}$ can be computed by repeatedly applying $\mathcal{F}$ to any element that is smaller than the least fixpoint; that is, for any $\sigma \subseteq \mu\mathcal{F}$, $\mu\mathcal{F} = \bigcup i \in \quad .\,\mathcal{F}^i(\sigma)$.

If K has a finitely branching transition relation, $\mathcal{F}^{\phi}_{\mathbf{E}'}$ is $\bigcup$-continuous. hence, $[\![\mu\mathbf{x}.\,\phi]\!]_{\mathbf{E}'}$ equals $\bigcup i \in \quad .\,(\mathcal{F}^{\phi}_{\mathbf{E}'})^i(\sigma)$ for any region $\sigma \subseteq [\![\mu\mathbf{x}.\,\phi]\!]_{\mathbf{E}'}$. It suffices to show that $[\![\mu\mathbf{x}.\,\phi]\!]_{\mathbf{E}} \subseteq [\![\mu\mathbf{x}.\,\phi]\!]_{\mathbf{E}'}$. This can be proved, by induction on the structure of ϕ, using the assumption that for every region variable $\mathbf{y}$ that is free in $\mu\mathbf{x}.\,\phi$, $\mathbf{E}(\mathbf{y}) \subseteq \mathbf{E}'(\mathbf{y})$. ■

The reinitialization is necessary only when there is a switch in the fixpoint quantifiers. The resulting algorithm is shown in Figure 11.2. When *Eval* is invoked on a fixpoint subformula $\mu\mathbf{x}.\,\phi$, the if the enclosing fixpoint subformula is a ν-formula, then $\mathbf{E}(\mathbf{x})$, together with the variables corresponding to μ-subformulas of ϕ that have no enclosing ν-subformula within ϕ, are initialized to the empty set. Otherwise, $\mathbf{E}(\mathbf{x})$ is left unchanged, and equals the value returned by the previous invocation of *Eval*(ϕ).

The algorithm uses the following new operations:

Closed? : **form** $\mapsto \mathbb{B}$. Given a C<span style="font-variant:small-caps">t</span>μ formula ψ, *Closed*?(ψ) returns *true* if ψ is closed.

Switch? : **form** $\times$ **form** $\mapsto \mathbb{B}$. For C<span style="font-variant:small-caps">t</span>μ formulas ψ and ϕ, *Switch*?(ψ, ϕ) returns *true* iff there exists a formula χ different from ψ such that (1) ψ is a fixpoint subformula of χ, (2) χ is a fixpoint subformula of ϕ, (3) there is no formula χ' such that χ' is a fixpoint subformula of χ and ψ is a subformula of χ', and (4) the fixpoint-types of ψ and χ are different.

AtomEval. Given a atomic formula p and an observation structure K, *AtomEval*(p, K) returns the characteristic region $[\![p]\!]_K$.

Algorithm 11.1 [Symbolic CTμ model checking]

```
Input: a closed CT$\mu$ formula $\phi$, and a $\phi$-structure $K$ with a finitely-
    branching transition relation.
Output: the answer to the model-checking problem $(K, \phi)$.

local Done: table of form × region;  E: table of var × region
$\Sigma := AtomEval(true, K)$;
$Done := EmptyTable$;  $\mathbf{E} := EmptyTable$;
if $InitReg(K) \subseteq Eval(\phi)$ then return YES else return NO.

function Eval
input $\psi$: form
  if $Closed?(\psi)$ and $Done[\psi] \neq \bot$ then return $Done[\psi]$ fi;
  case $\psi = p$ for an atomic formula $p$: $\sigma := AtomEval(p, K)$
  case $\psi = \neg p$ for an atomic formula $p$: $\sigma := \Sigma \setminus AtomEval(p, K)$
  case $\psi = \chi_1 \vee \chi_2$: $\sigma := Eval(\chi_1) \cup Eval(\chi_2)$
  case $\psi = \chi_1 \wedge \chi_2$: $\sigma := Eval(\chi_1) \cap Eval(\chi_2)$
  case $\psi = \exists\bigcirc \chi$: $\sigma := PreReg(Eval(\chi), K)$
  case $\psi = \forall\bigcirc \chi$: $\sigma := \Sigma \setminus PreReg(\Sigma \setminus Eval(\chi), K)$
  case $\psi = \mu\mathbf{x}.\,\chi$:
    if $Switch?(\psi, \phi)$ or $Closed?(\psi)$ then $Initialize(\psi, mu)$ fi;
    repeat $\sigma := \mathbf{E}(\mathbf{x})$; $\mathbf{E}(\mathbf{x}) := Eval(\chi)$ until $\sigma = \mathbf{E}(\mathbf{x})$;
  case $\psi = \nu\mathbf{x}.\,\chi$:
    if $Switch?(\psi, \phi)$ or $Closed?(\psi)$ then $Initialize(\psi, nu)$ fi;
    repeat $\sigma := \mathbf{E}(\mathbf{x})$; $\mathbf{E}(\mathbf{x}) := Eval(\chi)$ until $\sigma = \mathbf{E}(\mathbf{x})$;
  case $\psi = \mathbf{x}$: $\sigma := \mathbf{E}(\mathbf{x})$;
  end case
  if $Closed?(\psi)$ then $Done[\psi] := \sigma$;
  return $\sigma$
  end.

function Initialize
input $\psi$: form;  $m$: $\{mu, nu\}$
  case $\psi = p$ for an atomic formula $p$:
  case $\psi = \neg p$ for an atomic formula $p$:
  case $\psi = \chi_1 \vee \chi_2$: $Initialize(\chi_1, m)$;  $Initialize(\chi_2, m)$
  case $\psi = \chi_1 \wedge \chi_2$: $Initialize(\chi_1, m)$;  $Initialize(\chi_2, m)$
  case $\psi = \exists\bigcirc \chi$: $Initialize(\chi, m)$
  case $\psi = \forall\bigcirc \chi$: $Initialize(\chi, m)$
  case $\psi = \mu\mathbf{x}.\,\chi$:
    if $m = mu$ then $\mathbf{E}(\mathbf{x}) := EmptySet$;  $Initialize(\chi, m)$ fi
  case $\psi = \nu\mathbf{x}.\,\chi$:
    if $m = nu$ then $\mathbf{E}(\mathbf{x}) := \Sigma$;  $Initialize(\chi, m)$ fi
  case $\psi = \mathbf{x}$:
  end case
  end.
```

Figure 11.2: Symbolic CTμ model checking

Theorem 11.6 [Correctness of CTμ model checking] *Given an observation structure K with finite bisimulation, and a closed* CTμ *formula ϕ, Algorithm 11.1 terminates with the correct answer to the model checking problem (K, ϕ).*

Theorem 11.7 [Complexity of CTμ model checking] *Let K be a finite observation structure with n states and m transitions, and let ϕ be a closed* CTμ *formula with length ℓ and alternation-depth k. Algorithm 11.1 solves the model checking problem (K, ϕ) in time $O((\ell \cdot (m + n))^{k+1})$.*

If the input structure for Algorithm 11.1 is finite, then all state predicates that are computed by the algorithm can be viewed as propositional formulas. An implementation of symbolic CTμ model checking for finite observation structures, then, may use BDDs. By Theorem 11.5, we can reduce the verification problem for CTL over fair modules to the CTμ verification problem. Consequently, we have symbolic procedure for CTL verification.

Contents

Chapter 12

Automata-theoretic Liveness Verification

In this chapter, we extend the automata-theoretic approach studied in Chapter 8 for safety requirements to liveness requirements. In the automata-theoretic liveness verification, a fair module is viewed as a generator of an ω-language, namely, the set of its fair traces, the requirement is specified by an ω-automaton that accepts only the desired ω-traces, and the verification problem corresponds to inclusion between these two ω-languages.

12.1 ω-Automata

A fair structure $\mathcal{K}$ consists of an observation structure K and a fairness assumption F. Each fair structure defines the ω-language $\mathcal{L}_{\mathcal{K}}$ over the set of its observations consisting of the set of ω-traces corresponding to initialized F-fair ω-trajectories. Fair structures can be used to specify requirements also. In their role as a specification language, fairness constraints are usually specified using regions rather than actions. In this role, fairness constraints should be viewed as *accepting conditions* that classify ω-trajectories into accepting and rejecting rather than assumptions about fair resolution of choice. We will concentrate on two types of accepting conditions: Büchi acceptance and Streett acceptance.

Büchi automata

Finite structures with a single weak-fairness constraint specified by a region are called Büchi automata.

BÜCHI AUTOMATON

A *Büchi automaton* $\mathcal{M}$ consists of (1) a finite observation structure K and (2) [the *repeating region*] a region σ^A of K. An initialized ω-trajectory $\underline{s}$ of K is *accepted* by the Büchi automaton $\mathcal{M}$ if $s_i \in \sigma^A$ for infinitely many positions $i \geq 0$. The *ω-language* $\mathcal{L}_\mathcal{M}$ of the Büchi automaton $\mathcal{M}$ is the set of traces corresponding to initialized accepted trajectories of $\mathcal{M}$. The Büchi automaton (K, σ^A) is *deterministic* if the observation structure K is deterministic.

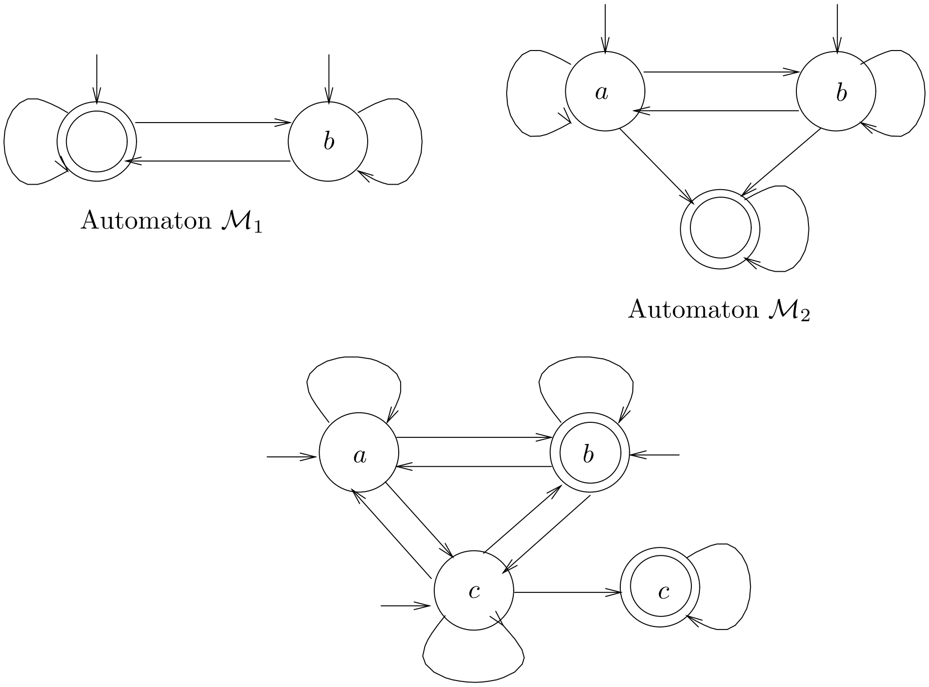
Note that syntactically a Büchi automaton is identical to an ordinary automaton. In an ordinary automaton, a (finite) trajectory is accepted if it terminates in an accepting state; in a Büchi automaton, an ω-trajectory is accepted if its visits a repeating state infinitely often.

Example 12.1 [Büchi languages] Let $A = \{a, b\}$. The Büchi automaton $\mathcal{M}_1$ of Figure 12.1 accepts the response language $(b^* a)^\omega$ consisting of ω-words that contain infinitely many a symbols. The Büchi automaton $\mathcal{M}_2$ of Figure 12.1 accepts the persistence language $A^* a^\omega$ consisting of ω-words with a suffix containing only a symbols. Note that the automaton $\mathcal{M}_2$ is nondeterministic (it guesses the beginning of the suffix containing only a symbols).

Let $A = \{a, b, c\}$. The nondeterministic Büchi automaton $\mathcal{M}_3$ of Figure 12.1 accepts the reactivity language consisting of ω-words that contain either only finitely many a symbols or infinitely many b symbols. Note that "finitely many a or infinitely many b" is equivalent to "infinitely many b or eventually always c." ■

Remark 12.1 [Multi-Büchi automaton] A *multi-Büchi automaton* $\mathcal{M}$ consists of (1) a finite observation structure K, and (2) a finite set F of *repeating regions* of K. An initialized ω-trajectory $\underline{s}$ of K is accepted by the multi-Büchi automaton $\mathcal{M}$ if for every repeating region $\sigma \in F$, $s_i \in \sigma$ for infinitely many positions $i \geq 0$. Thus, a multi-Büchi automaton is a weak-fair structure all of whose weak-fairness constraints are specified by regions. ■

Exercise 12.1 {`P2`} [CoBüchi automata] A *CoBüchi automaton* $\mathcal{M}$ consists of (1) a finite observation structure K and (2) [the *stable region*] a region σ^A of K. An initialized ω-trajectory $\underline{s}$ of K is accepted by the CoBüchi automaton $\mathcal{M}$ if it has a suffix all of whose states are in the stable region: there exists $i \geq 0$ such that $s_j \in \sigma^A$ for all $j \geq i$. Note that syntactically a CoBüchi automaton is



Automaton $\mathcal{M}_1$

Automaton $\mathcal{M}_2$

Automaton $\mathcal{M}_3$

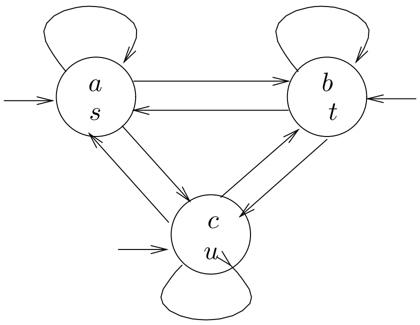
Figure 12.1: Sample Büchi automata

like a Büchi automaton or an ordinary automaton. Semantically, the CoBüchi automaton (K, σ^A) is like the fair structure $(K, \{(\Sigma \backslash \sigma^A, \emptyset)\})$.

Consider the alphabet $A = \{a, b\}$. (1) Find a CoBüchi automaton whose ω-language is the persistence language $A^* a^\omega$ consisting ω-words with a suffix containing only a symbols. (2) Consider the response language $(b^* a)^\omega$ consisting of ω-words with infinitely many a symbols. Can you draw a CoBüchi automaton that accepts this langauge? ■

Streett Automata

Finite structures with a fairness constraints specified by regions are called Streett automata.



Streett Automaton $\mathcal{M}$

Fairness constraint =(s, t)

Figure 12.2: Sample Streett automaton

STREETT AUTOMATON

A *Streett automaton* $\mathcal{M}$ consists of (1) a finite observation structure K and (2) [the *Streett constraints*] a finite set F of pairs of regions. An initialized ω-trajectory $\underline{s}$ of K is *accepted* by the Streett automaton $\mathcal{M}$ if for every Streett constraint $(\sigma, \tau) \in F$, if $\underline{s}$ is σ-fair then $\underline{s}$ is τ-fair. The *ω-language* $\mathcal{L}_{\mathcal{M}}$ of the Streett automaton $\mathcal{M}$ is the set of ω-traces corresponding to initialized accepted ω-trajectories of $\mathcal{M}$.

Remark 12.2 [Büchi as a special case of Streett] A Büchi automaton (K, σ^A) can be viewed as the Streett automaton $(K, \{(\Sigma, \sigma^A)\})$ with a single Streett constraint. ■

Example 12.2 [Streett language] Let $A = \{a, b, c\}$. The nondeterministic Büchi automaton $\mathcal{M}_3$ of Figure 12.1 accepts the reactivity language consisting of ω-words that contain either only finitely many a symbols or infinitely many b symbols. The same ω-language is accepted by the deterministic Streett automaton $\mathcal{M}$ of Figure 12.2. There is a single Streett constraint $(\{s\}, \{t\})$. ■

Exercise 12.2 {`P2`} [Rabin automata] A *Rabin automaton* $\mathcal{M}$ is syntactically identical to a Streett automaton, and consists of (1) a finite observation structure K and (2) [the *Rabin constraints*] a finite set F of pairs of regions. An initialized ω-trajectory $\underline{s}$ of K is *accepted* by the Rabin automaton $\mathcal{M}$ if there exists a Rabin constraint $(\sigma, \tau) \in F$ such that $\underline{s}$ is σ-fair and but not τ-fair. Thus, semantically a Rabin automaton is the dual of the Streett automaton: Streett acceptance has the form

$$\bigwedge(\sigma, \tau) \in F.\ [\neg(\sigma\text{-fair}) \vee \tau\text{-fair}\ ],$$

while Rabin acceptance requires

$$\bigvee(\sigma, \tau) \in F.\ [\sigma\text{-fair} \wedge \neg(\tau\text{-fair})\ ].$$

Let $A = \{a, b, c\}$. Find a deterministic Rabin automaton $\mathcal{M}$ accepting the reactivity language consisting of ω-words that contain either only finitely many a symbols or infinitely many b symbols. ■

Exercise 12.3 $\{\texttt{T3}\}$ [Muller automata] A *Muller automaton* $\mathcal{M}$ is syntactically like a multi-Büchi automaton, and consists of (1) a finite observation structure K and (2) [the *Muller acceptance*] a finite set F of regions of K. An initialized ω-trajectory $\underline{s}$ of K is accepted by the Muller automaton $\mathcal{M}$ if the set $\{s \in \Sigma \mid s_i = s \text{ for infinitely many } i \geq 0\}$ of states repeating infinitely often along $\underline{s}$ is in F. Show that Streett automata as well as Rabin automata are special cases of Muller automata. ■

The ω-language-inclusion problem

The ω-language-inclusion problem asks whether every ω-trace accepted by one ω-automaton is also accepted by another ω-automaton.

> THE ω-LANGUAGE-INCLUSION PROBLEM
>
> An instance $(\mathcal{M}_1, \mathcal{M}_2)$ of the *ω-language-inclusion problem* consists of two ω-automata $\mathcal{M}_1$ and $\mathcal{M}_2$ over the same observation alphabet A. The answer to the ω-language-inclusion problem $(\mathcal{M}_1, \mathcal{M}_2)$ is YES if $\mathcal{L}_{\mathcal{M}_1} \subseteq \mathcal{L}_{\mathcal{M}_2}$, and otherwise NO.

Note that in an instance $(\mathcal{M}_1, \mathcal{M}_2)$ of the ω-language-inclusion problem, each of the ω-automata $\mathcal{M}_1$ and $\mathcal{M}_2$ may be either a fair structure, or a Büchi automaton, or a Streett automaton.

Automata as specifications

ω-automata can be used for specifying requirements of fair modules. As in case of the logic SAL, the observations of the requirements automaton are boolean expressions over the observable variables of modules. We define the fair state logic LAL whose formulas are Büchi and Streett automata.

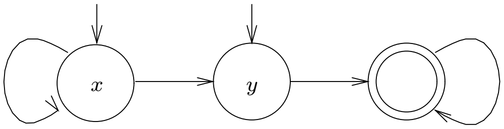


Figure 12.3: The Lal formula $\mathcal{M}_{\forall\mathcal{U}}$

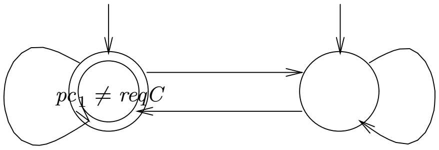


Figure 12.4: Starvation freedom in live automaton logic

Live automaton logic

A formula of the fair state logic *live automaton logic* (Lal) is a Büchi or a Streett automaton $\mathcal{M}$ whose observations are boolean expressions.

Given a formula $\mathcal{M}$ of Lal, a fair structure $\mathcal{K}$ is a $\mathcal{M}$-structure if each observation of $\mathcal{K}$ is a valuation for a superset of the variables appearing in the observations of $\mathcal{M}$.

The satisfaction relation for Lal is defined by:

$$\begin{array}{lll} s \models_{\mathcal{K}} \mathcal{M} & \text{iff} & \text{for every source-}s\text{ fair }\omega\text{-trajectory }\underline{s}\text{ of }\mathcal{K} \\ & & \text{there is an accepting }\omega\text{-trace }\underline{a} \in \mathcal{L}_{\mathcal{M}}\text{ such that} \\ & & \text{for all }i \geq 0,\ s_i \models a_i. \end{array}$$

In other words, a state s of $\mathcal{K}$ satisfies the requirement specified by the ω-automaton $\mathcal{M}$ if for every source-s fair ω-trace $\underline{a}$ of $\mathcal{K}$, we can find an initialized accepting ω-trace $\underline{b}$ of M such that every observation in $\underline{a}$ is consistent with the corresponding expression in $\underline{b}$.

Example 12.3 [Lal] The Lal formula $\mathcal{M}_{\forall\mathcal{U}}$ shown in Figure 12.3 asserts that, given a state s, every source-s fair ω-trajectory contains a state satisfying y which is preceded only by states satisfying x. The formula $\mathcal{M}_{\forall\mathcal{U}}$ can be interpreted at states of a fair structure whose observations assign values to x and y. It follows that the Lal formula $\mathcal{M}_{\forall\mathcal{U}}$ is equivalent to the Ctl formula $x\forall\mathcal{U}y$. Contrast the specification $\mathcal{M}_{\forall\mathcal{U}}$ with the Sal specification $M_{\mathcal{W}}$ corresponding to the Stl formula $x\forall\mathcal{W}y$ (see Example 8.2). ■

Example 12.4 [Starvation freedom in live automaton logic] Recall the starvation freedom requirement for mutual exclusion protocols. The requirement that $pc_1 \neq reqC$ be a recurrent is expressed in Lal by the Büchi automaton of Figure 12.4. ■

Lal model checking

The model-checking problem for Lal can be reduced to the ω-language-inclusion problem. As in case of Sal, we expand each ω-automaton $\mathcal{M}$ of Lal to a larger automaton $E\mathcal{M}$ whose observations are valuations to the variables appearing in the observations of $\mathcal{M}$. Recall the definition of the expansion operator E from Chapter 8. To obtain expansion of an ω-automaton, we apply the expansion operation to the underlying observation structure, and modify the accepting condition appropriately.

Expansion of an Lal automaton

For a Büchi automaton $\mathcal{M} = (K, \sigma^A)$ given as a Lal formula, the *expansion* $E\mathcal{M}$ is another Büchi automaton: (1) the observation structure of $E\mathcal{M}$ is EK, and (2) the repeating region of $E\mathcal{M}$ is $\sigma^A \Uparrow$.

For a Streett automaton $\mathcal{M} = (K, F)$ given as a Lal formula, the *expansion* $E\mathcal{M}$ is another Streett automaton: (1) the observation structure of $E\mathcal{M}$ is EK, and (2) for every $(\sigma, \tau) \in F$, the automaton $E\mathcal{M}$ has the Streett constraint $(\sigma \Uparrow, \tau \Uparrow)$.

Exercise 12.4 {P1} [Lal expansion] Draw the expanded Büchi automaton corresponding to the Lal specification $\mathcal{M}_{\forall \mathcal{U}}$ of Figure 12.3. ■

It follows that checking whether a fair structure satisfies an Lal automaton $\mathcal{M}$ is equivalent to checking whether $\mathcal{K}$ satsifies the expanded ω-automaton $E\mathcal{M}$, which in turn corresponds to checking whether the fair language of $\mathcal{K}$ is contained in the fair language of $E\mathcal{M}$.

Proposition 12.1 [Lal model checking] *The* Lal *model-checking problem* $(\mathcal{K}, \mathcal{M})$ *and the ω-language-inclusion problem* $(\mathcal{K}, E\mathcal{M})$ *have the same answer.*

12.2 Operations on ω-automata

To solve the ω-language inclusion problem $(\mathcal{M}_1, \mathcal{M}_2)$, we first obtain an ω-automaton that accepts the complement of the ω-language accepted by $\mathcal{M}_2$, then construct its product with $\mathcal{M}_1$, and solve the fair emptiness problem on the resulting ω-automaton.

12.2.1 Product

Given two ω-automata $\mathcal{M}_1$ and $\mathcal{M}_2$, we wish to define another ω-automaton that accepts the intersection of the ω-languages of $\mathcal{M}_1$ and $\mathcal{M}_2$. For this purpose, we resort to the product construction described in Section 8.3.2 over observation structures. Consider two observation structures K_1 and K_2, and let $K_1 \times K_2$ be their product. Let $\underline{s}$ be an ω-trajectory of the product. Then, by the definition of the product, there exists an ω-trajectory $\underline{t}$ of K_1 and an ω-trajectory $\underline{u}$ of K_2 such that $\underline{s} = (t_0, u_0)(t_1, u_1)\cdots$. Observe that, for a region σ of K_1, the ω-trajectory $\underline{t}$ of K_1 is σ-fair iff the ω-trajectory $\underline{s}$ of the product is $(\sigma \Uparrow)$-fair. Similarly, for a region σ of K_2, the ω-trajectory $\underline{u}$ of K_2 is σ-fair iff the ω-trajectory $\underline{s}$ of the product is $(\sigma \Uparrow)$-fair. In other words, fairness with respect to a region σ in a component translates to fairness with respect to the lifted region $\sigma \Uparrow$ in the product. Similarly, fairness with respect to an action α in a component translates to fairness with respect to the lifted action $\alpha \Uparrow$ in the product. This leads to a natural definition of product for ω-automata.

Product of ω-automata

Let $\mathcal{M}_1 = (K_1, F_1)$ and $\mathcal{M}_2 = (K_2, F_2)$ be two Streett automata. The product $\mathcal{M}_1 \times \mathcal{M}_2$ is the Streett automaton $(K_1 \times K_2, \{(\sigma \Uparrow, \tau \Uparrow) \mid (\sigma, \tau) \in F_1 \cup F_2\})$.

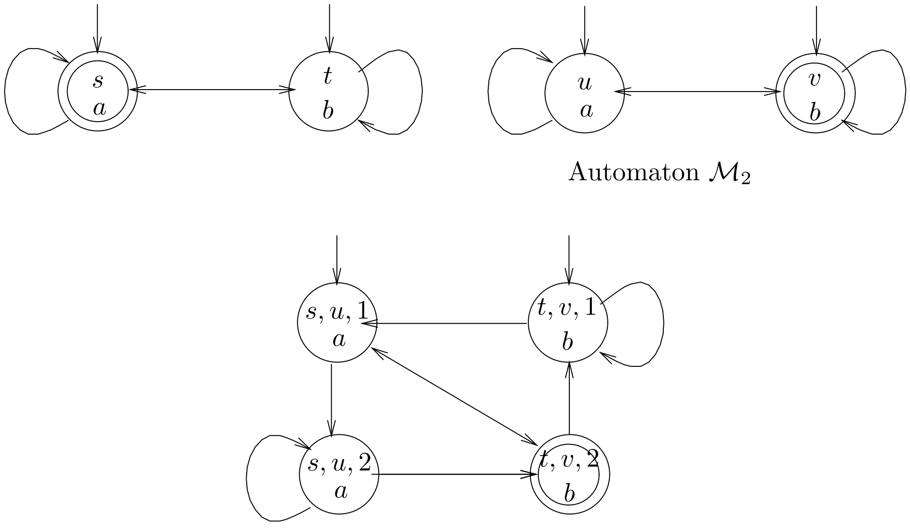
Let $\mathcal{K}_1 = (K_1, F_1)$ and $\mathcal{K}_2 = (K_2, F_2)$ be two fair structures. The product $\mathcal{K}_1 \times \mathcal{K}_2$ is the fair structure $(K_1 \times K_2, \{(\alpha \Uparrow, \beta \Uparrow) \mid (\alpha, \beta) \in F_1 \cup F_2\})$.

Proposition 12.2 [Product of ω-automata] *If $\mathcal{M}_1$ and $\mathcal{M}_2$ are two Streett automata, then $\mathcal{L}_{\mathcal{M}_1 \times \mathcal{M}_2} = \mathcal{L}_{\mathcal{M}_1} \cap \mathcal{L}_{\mathcal{M}_2}$. If $\mathcal{K}_1$ and $\mathcal{K}_2$ are two fair structures, then $\mathcal{L}_{\mathcal{K}_1 \times \mathcal{K}_2} = \mathcal{L}_{\mathcal{K}_1} \cap L_{\mathcal{K}_2}$.*

Remark 12.3 [Cost of product] Let $\mathcal{M}_1$ be a Streett automaton with n_1 states, m_1 transitions, and ℓ_1 Streett constraints. Let $\mathcal{M}_2$ be a Streett automaton with n_2 states, m_2 transitions, and ℓ_2 Streett constraints. Then, the product $\mathcal{M}_1 \times \mathcal{M}_2$ has at most $n_1 \cdot n_2$ states, at most $m_1 \cdot m_2$ transitions, and $\ell_1 + \ell_2$ Streett constraints. ■

Product of Büchi automata

The product of two Büchi automata (K_1, σ_1^A) and (K_2, σ_2^A) can be defined to be the multi-Büchi automaton $(K_1 \times K_2, \{\sigma_1^A \Uparrow, \sigma_2^A \Uparrow\})$. However, by introducing a counter, as described in Section 11.3.2, we can define product of Büchi automata to be a Büchi automaton. The states of the product are, then, of the form (s, t, i), where s is a state of K_1, t is a state of K_2, and i is a counter that can be either 1 or 2. The counter is updated from 1 to 2 when an accepting state of K_1 is visited, and from 2 to 1 when an accepting state of K_2 is visited. The



Automaton $\mathcal{M}_2$

Product Automaton $\mathcal{M}_1 \times \mathcal{M}_2$

Figure 12.5: Product construction for Büchi automata

accepting condition of the product requires infinitely many updates from 2 to 1.

Product of Büchi automata

Let $\mathcal{M}_1 = (\Sigma_1, \sigma_1^I, \rightarrow_1, A, \langle\!\langle\cdot\rangle\!\rangle_1, \sigma_1^A)$ and $\mathcal{M}_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2, A, \langle\!\langle\cdot\rangle\!\rangle_2, \sigma_2^A)$ be two Büchi automata. The *product* $\mathcal{M}_1 \times \mathcal{M}_2$ is the Büchi automaton $(\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle, \sigma^A)$:

- $\Sigma = \{(s_1, s_2, i) \mid s_1 \in \Sigma_1,\ s_2 \in \Sigma_2,\ \langle\!\langle s_1\rangle\!\rangle_1 = \langle\!\langle s_2\rangle\!\rangle_2,\ \text{and } i \in \{1, 2\}\}$;
- $(s_1, s_2, i) \in \sigma^I$ iff $s_1 \in \sigma_1^I$, $s_2 \in \sigma_2^I$, and $i = 1$;
- $(s_1, s_2, i) \rightarrow (t_1, t_2, j)$ iff $s_1 \rightarrow_1 t_1$, $s_2 \rightarrow_2 t_2$, if $i = 1$ then if $s_1 \in \sigma_1^A$ then $j = 2$ else $j = 1$, and if $i = 2$ then if $s_2 \in \sigma_2^A$ then $j = 1$ else $j = 2$;
- $\langle\!\langle (s_1, s_2, i)\rangle\!\rangle = \langle\!\langle s_1\rangle\!\rangle_1 = \langle\!\langle s_2\rangle\!\rangle_2$;
- $(s_1, s_2, i) \in \sigma^A$ if $i = 2$ and $s_2 \in \sigma_2^A$.

Proposition 12.3 [Product of Büchi automata] *If* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *are two Büchi automata, then* $\mathcal{L}_{\mathcal{M}_1 \times \mathcal{M}_2} = \mathcal{L}_{\mathcal{M}_1} \cap \mathcal{L}_{\mathcal{M}_2}$.

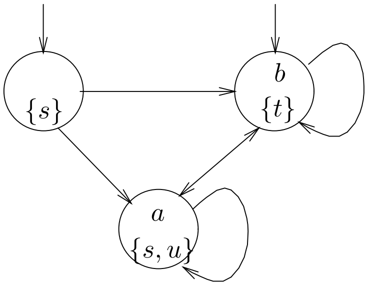


Figure 12.6: Subset construction does not work for Büchi acceptance

Example 12.5 [Product of Büchi automata] Consider the two Büchi automata $\mathcal{M}_1$ and $\mathcal{M}_2$ of Figure 12.5. The automaton $\mathcal{M}_1$ accepts all ω-words that contain infinitely many a symbols, while $\mathcal{M}_2$ accepts all ω-words that contain infinitely many b symbols. The result of applying the product construction contains 4 states, of which the only accepting state is $(t, v, 2)$. Verify that the product accepts precise those ω-words that contain infinitely many a symbols as well as infinitely many b symbols. ■

Remark 12.4 [Product of deterministic Büchi automata] If $\mathcal{M}_1$ and $\mathcal{M}_2$ are deterministic Büchi automata, then so is their product $\mathcal{M}_1 \times \mathcal{M}_2$. Thus, the class of ω-languages definable by deterministic Büchi automata is closed under intersection. ■

Exercise 12.5 {`T2`} [Product of CoBüchi automata] Given two CoBüchi automata $\mathcal{M}_1$ and $\mathcal{M}_2$, define a CoBüchi automaton $\mathcal{M}_1 \times \mathcal{M}_2$ that accepts the intersection of the ω-languages of $\mathcal{M}_1$ and $\mathcal{M}_2$. ■

12.2.2 Complementation

We turn our attention to the problem of complementing a Büchi automaton. Recall that for an ordinary automaton, its complement is constructed by first determinizing the observation structure using the subset construction, followed by completion by adding dummy states, followed by inversion of the accepting condition. Given a Büchi automaton $\mathcal{M} = (K, \sigma^A)$, can we add accepting conditions to the determinized structure ΔK without changing the ω-language accepted by $\mathcal{M}$? The obstacle in such an approach is illustrated by the following example.

Example 12.6 [Subset construction and Büchi automata] Recall the Büchi automaton $\mathcal{M}_2$ from Figure 12.1 that accepts the persistence language A^*a^ω. The determinized structure obtained by subset construction is shown in Figure 12.6. Declaring the state corresponding to the subset $\{s, u\}$ to be repeating does not preserve the ω-language of $\mathcal{M}_2$. ■

The problem with the subset construction is that states of the determinized structure may contain both repeating and nonrepeating states. Complementing a nondeterministic Büchi automaton turns out to be a nontrivial problem. Consequently, existing model checkers do not support nondeterministic ω-automata as specifications. However, understanding the complementation procedure provides insights into the structure of ω-automata.

We begin by some preliminary definitions. Let A be a finite alphabet. An equivalence relation $\sim \subseteq A^* \times A^*$ over words over A is said to be a *congruence* (with respect to concatenation) if for all words $\overline{a}$, $\overline{b}$, and $\overline{c}$, if $\overline{a} \sim \overline{b}$ then $\overline{a} \cdot \overline{c} \sim \overline{b} \cdot \overline{c}$ and $\overline{c} \cdot \overline{a} \sim \overline{c} \cdot \overline{b}$. By a *finite* equivalence relation, we mean an equivalence relation with finitely many equivalence classes.

Let $\mathcal{M} = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle \cdot \rangle\!\rangle, \sigma^A)$ be a Büchi automaton. We are going to establish that both $\mathcal{L}_{\mathcal{M}}$ and and its complement can be expressed as finite unions of ω-languages of the form $L_1 \cdot L_2^\omega$, where L_1 and L_2 are blocks of a certain finite congruence on A^*.

> For two state s and t of $\mathcal{M}$, and a word $\overline{a}_{0 \ldots m}$ over A, define $s \, \alpha^T \overline{a} \, t$ if there is a trajectory $\overline{s}_{0 \ldots m}$ of $\mathcal{M}$ such that $s_0 = s$, $s_m = t$, and $\langle\!\langle s_i \rangle\!\rangle = a_i$ for all $0 \leq i \leq m$.

That is, $s \, \alpha^T \overline{a} \, t$ means that the trace $\overline{a}$ can lead the automaton from the initial state s to the final state t.

> For two state s and t of $\mathcal{M}$, and a word $\overline{a}_{0 \ldots m}$ over A, define $s \, \alpha^T \overline{a}' \, t$ if there is a trajectory $\overline{s}_{0 \ldots m}$ of $\mathcal{M}$ such that $s_0 = s$, $s_m = t$, $\langle\!\langle s_i \rangle\!\rangle = a_i$ for all $0 \leq i \leq m$, and $s_j \in \sigma^A$ for some $0 \leq j \leq m$.

That is, $s \, \alpha^T \overline{a}' \, t$ means that the trace $\overline{a}$ can lead the automaton from the initial state s to the final state t via a trajectory that visits some repeating state. Now we are ready to define the desired equivalence relation on A^* induced by $\mathcal{M}$:

> For two words $\overline{a}$ and $\overline{b}$ over A, $\overline{a} \sim_{\mathcal{M}} \overline{b}$ iff for all states s and t of $\mathcal{M}$, (1) $s \, \alpha^T \overline{a} \, t$ iff $s \, \alpha^T \overline{b} \, t$, and (2) $s \, \alpha^T \overline{a}' \, t$ iff $s \, \alpha^T \overline{b}' \, t$.

First, we note that the equivalence $\sim_{\mathcal{M}}$ is a finite congruence:

Lemma 12.1 [Congruence] *The equivalence relation $\sim_{\mathcal{M}}$ over A^* is a congruence with respect to concatenation.*

Proof. Left as an exercise. ■

Lemma 12.2 [Finiteness] *The equivalence relation $\sim_{\mathcal{M}}$ over A^* is finite, and has at most $\mathbf{2}^{2n^2}$ classes if $\mathcal{M}$ has n states.*

Proof. For every two states s and t of $\mathcal{M}$, let $L_{s,t}$ be the language containing words $\overline{a}$ such that $s\,\alpha^T\overline{a}\,t$, and let $L'_{s,t}$ be the language containing words $\overline{a}$ such that $s\,\alpha^T\overline{a}'\,t$. Let Π be the set of these $2n^2$ languages. Now an equivalence class of $\sim_{\mathcal{M}}$ corresponds to a subset of Π: given a subset $\Pi' \subseteq \Pi$, the intersection

$$[\bigcap L \in \Pi'.\, L\,] \;\cap\; [\bigcap L \notin \Pi'.\, A^*\backslash L\,]$$

defines an equivalence class of $\sim_{\mathcal{M}}$. It follows that the number of subsets of Π is an upper bound for the number of equivalence classes of $\sim_{\mathcal{M}}$. ■

The next lemma asserts a saturation property of the $\sim_{\mathcal{M}}$-equivalence classes with respect to the ω-language accepted by $\mathcal{M}$:

Lemma 12.3 [Saturation] *Let L_1 and L_2 be two equivalence classes of the congruence $\sim_{\mathcal{M}}$. Then, if $L_1 \cdot L_2^\omega \cap \mathcal{L}_{\mathcal{M}}$ is nonempty then $L_1 \cdot L_2^\omega \subseteq \mathcal{L}_{\mathcal{M}}$.*

Proof. Let L_1 and L_2 be two equivalence classes of $\sim_{\mathcal{M}}$. Suppose $L_1 \cdot L_2^\omega \cap \mathcal{L}_{\mathcal{M}}$ is nonempty, and contains the ω-word $\underline{a}$. Since $\underline{a} \in L_1 \cdot L_2^\omega$, it is of the form $\overline{b}_0 \cdot \overline{b}_1 \cdot \overline{b}_2 \cdots$, where the word $\overline{b}_0$ is in L_1 and for $i \geq 1$, the word $\overline{b}_i$ is in L_2. Since $\underline{a}$ is accepted by $\mathcal{M}$, there exists an initialized accepting ω-trajectory corresponding to $\underline{a}$. Thus, there exist states $s_0, s_1, \ldots$ such that

$$s_0\ \alpha^T\overline{b}_0\ s_1\ \alpha^T\overline{b}_1\ s_1\ \alpha^T\overline{b}_2\ \cdots$$

Furthermore, for infinitely many indices i, $s_i\ \alpha^T\overline{b}_i{}'\ s_{i+1}$.

Now consider another word $\underline{c} \in L_1 \cdot L_2^\omega$. We need to establish that $\mathcal{M}$ accepts $\underline{c}$ also. The ω-word $\underline{c}$ is of the form $\overline{d}_0 \cdot \overline{d}_1 \cdot \overline{d}_2 \cdots$ such that the word $\overline{d}_0$ is in L_1 and for $i \geq 1$, the word $\overline{d}_i$ is in L_2. Since L_1 and L_2 are equivalence classes of $\sim_{\mathcal{M}}$, $\overline{b}_i \sim_{\mathcal{M}} \overline{d}_i$ for all $i \geq 0$. It follows that

$$s_0\ \alpha^T\overline{d}_0\ s_1\ \alpha^T\overline{d}_1\ s_1\ \alpha^T\overline{d}_2\ \cdots$$

and for infinitely many indices i, $s_i\ \alpha^T\overline{d}_i{}'\ s_{i+1}$. We conclude that there is an initialized accepting trajectory corresponding to the ω-word $\underline{c}$. ■

The next lemma asserts that ω-languages of the form $L_1 \cdot L_2^\omega$ cover the set of all ω-words, provided L_1, L_2 range over equivalence classes of a finite congruence.

Lemma 12.4 [Coverage] *Let $\sim$ be a finite congruence over A^*, and let $\underline{a}$ be an ω-word over A. Then, there exist equivalence classes L_1 and L_2 of $\sim$ such that $\underline{a} \in L_1 \cdot L_2^\omega$.*

Proof. Let $\sim$ be a finite congruence relation over A^*, and let $\underline{a}$ be an infinite word over A. We say that two indices i and j merge at an index $k > i, j$ if $\overline{a}_{i\ldots k} \sim \overline{a}_{j\ldots k}$. For two indices i and j, define $i \cong j$ if they merge at some index. Verify that $\cong$ is an equivalence relation over the set of nonnegative integers. Furthermore, given a finite subset D of nonnegative integers such that D has more elements than the number of equivalence classes of $\sim$, if we choose k such that $k > i$ for all $i \in D$, then the set $\{\overline{a}_{i\ldots k} \mid i \in D\}$ must contain two $\sim$-equivalent words. It follows that the equivalence relation $\cong$ itself is finite (the number of equivalence classes of $\cong$ is bounded by the number of equivalence classes of $\sim$).

Finiteness of $\cong$ implies that there exists an infinite sequence $i_0 < i_1 < i_2 < \cdots$ of indices that are $\cong$-equivalent to each other. Note that for every $j \geq 1$, all the indices $i_0, i_1, \ldots i_j$ merge at some $k > i_j$. Without loss of generality, we may assume that for every $j \geq 1$, all the indices $i_0, i_1, \ldots i_j$ merge at i_{j+1} (this is because we can delete indices from the original sequence, and if $i_0, i_1, \ldots i_j$ merge at k then they merge at every $k' > k$ as $\sim$ is a congruence). It follows that there is an infinite sequence $i_0, i_1, i_2, \ldots$ of indices such that

1. all the words in $\{\overline{a}_{i_0\ldots i_j} \mid j \geq 1\}$ belong to the same equivalence class of $\sim$, let this class be L_2,
2. for all $j < j' < k$, the indices i_j and $i_{j'}$ merge at i_k.

From (1), $\overline{a}_{i_0\ldots i_1}$ is in L_2. For all $j \geq 1$, $\overline{a}_{i_0\ldots i_{j+1}}$ is in L_2 by (1), and $\overline{a}_{i_0\ldots i_{j+1}}$ is $\sim$-equivalent to $\overline{a}_{i_j\ldots i_{j+1}}$ by (2). It follows that for all $j \geq 0$, $\overline{a}_{i_j\ldots i_{j+1}}$ is in L_2.

It follows that the suffix $\overline{a}_{i_0\ldots}$ is in L_2^ω. This completes the proof. ■

Since $\sim_\mathcal{M}$ is a finite congruence, it follows that the set

$$\{L_1 \cdot L_2^\omega \mid L_1, L_2 \text{ are equivalence classes of } \sim_\mathcal{M}\}$$

covers A^ω, and then, by the saturation property, the ω-language accepted by $\mathcal{M}$ corresponds to a subset of this set, and the complement defines the complementary language.

Corollary 12.1 [Structure of Büchi language] *The ω-language $\mathcal{L}_\mathcal{M}$ accepted by the Büchi automaton $\mathcal{M}$ equals*

$$\bigcup\{L_1 \cdot L_2^\omega \mid\ L_1, L_2 \textit{ are equivalence classes of } \sim_\mathcal{M} \textit{ and } L_1 \cdot L_2^\omega \cap \mathcal{L}_\mathcal{M} \neq \emptyset\},$$

and the complementary ω-language $A^\omega \backslash \mathcal{L}_\mathcal{M}$ equals

$$\bigcup\{L_1 \cdot L_2^\omega \mid\ L_1, L_2 \textit{ are equivalence classes of } \sim_\mathcal{M} \textit{ and } L_1 \cdot L_2^\omega \cap \mathcal{L}_\mathcal{M} = \emptyset\}.$$

Proof. Follows from Lemmas 12.1, 12.2, 12.3, and 12.4. ■

The next proposition asserts that if L_1 and L_2 are two regular languages then the ω-language $L_1 \cdot L_2^\omega$ is accepted by a Büchi automaton.

Proposition 12.4 [Regular concatenation] *If L_1 and L_2 are two regular languages over A then the ω-language $L_1 \cdot L_2^\omega$ is accepted by some Büchi automaton.*

Proof. Let L_1 be a regular language accepted by the automaton $M_1 = (\Sigma_1, \sigma_1^I, \rightarrow_1, A, \langle\!\langle\cdot\rangle\!\rangle_1, \sigma_1^A)$, and let L_2 be a regular language accepted by the automaton $M_2 = (\Sigma_2, \sigma_2^I, \rightarrow_2, A, \langle\!\langle\cdot\rangle\!\rangle_2, \sigma_2^A)$. The ω-automaton accepting $L_1 \cdot L_2^\omega$ is obtained by taking disjoint union of the two automata M_1 and M_2, and adding transitions from accepting states of M_1 to the initial states of M_2, and from accepting states of M_2 to the initial states of M_2. Specifically, define the Büchi automaton $\mathcal{M}$ over the alphabet $\mathcal{M}$: (1) the state-space of $\mathcal{M}$ is $\Sigma_1 \cup \Sigma_2$ (assuming Σ_1 and Σ_2 are disjoint sets), (2) the initial region of $\mathcal{M}$ is σ_1^I, (3) the set of transitions of $\mathcal{M}$ equals $\rightarrow_1 \cup \rightarrow_2 \cup (\sigma_1^A \times \sigma_2^I) \cup (\sigma_2^A \times \sigma_2^I)$, (4) the observation of a state s of $\mathcal{M}$ is $\langle\!\langle s\rangle\!\rangle_1$ if $s \in \Sigma_1$ and $\langle\!\langle s\rangle\!\rangle_2$ otherwise, (5) the repeating region of $\mathcal{M}$ is σ_2^A. Verify that an ω-word $\underline{a}$ is accepted by the Büchi automaton $\mathcal{M}$ precisely when it belongs to $L_1 \cdot L_2^\omega$. ■

Remark 12.5 [Regular concatenation] If the regular language L_1 is accepted by an automaton with n_1 states, and the regular language L_2 is accepted by an automaton with n_2 states, then $L_1 \cdot L_2^\omega$ is accepted by a Büchi automaton with $n_1 + n_2$ states. ■

Theorem 12.1 [Büchi Theorem on Complementation Closure] *Given a Büchi automaton $\mathcal{M}$, there exists a Büchi automaton $\mathcal{M}'$ such that $\mathcal{L}_{\mathcal{M}'} = A^\omega \backslash \mathcal{L}_{\mathcal{M}}$.*

Proof. Let $\mathcal{M}$ be a Büchi automaton. The languages $L_{s,t}$ and $L'_{s,t}$ defined in the proof of Lemma 12.2 are regular languages. Since regular languages are closed under complement and intersection, from the proof of Lemma 12.2 it follows that every equivalence class of $\sim_{\mathcal{M}}$ is a regular language. By Proposition 12.4, for two equivalence classes L_1 and L_2 of $\sim_{\mathcal{M}}$, the ω-language $L_1 \cdot L_2^\omega$ is accepted by a Büchi automaton. Since Büchi automata are closed under union, from Corollary 12.1, it follows that $A^\omega \backslash \mathcal{L}_{\mathcal{M}}$ is accepted by some Büchi automaton. ■

Complexity of Complementation

Let us now analyze the bound on the size of the Büchi automaton accepting the complement of $\mathcal{L}_{\mathcal{M}}$ obtained by our construction. Suppose $\mathcal{M}$ has n states. Recall that $L_{s,t}$, for two states s and t of $\mathcal{M}$, is the language containing words $\overline{a}$ such that $s\,\alpha^T \overline{a}\, t$. It follows that there an automaton accepting $L_{s,t}$ with n states (use the same states and transitions of $\mathcal{M}$, but declare s to be initial and t to be accepting). The language $L'_{s,t}$ containing words $\overline{a}$ such that $s\,\alpha^T \overline{a}'\, t$ can be accepted by an automaton with $2n$ states (the state-space of $\mathcal{M}$ is doubled to keep track of whether an accepting state has been visited or not). It follows that the language $A^* \backslash L_{s,t}$ is accepted by an automaton with $\mathbf{2}^n$ states, and the language $A^*\backslash L'_{s,t}$ is accepted by an automaton with 4^n states (complementation may require determinization). Recall that an equivalence class of $\sim_{\mathcal{M}}$ corresponds to a subset Π' of the set Π containing $2n^2$ languages $L_{s,t}$, $L'_{s,t}$. Such an equivalence class Π' is defined by the product of the automata accepting L for $L \in \Pi'$ and the automata accepting $A^*\backslash L$ for $L \notin \Pi'$. Consequently, the bound on the size of the automaton accepting an equivalence class of $\sim_{\mathcal{M}}$ is $(\mathbf{2}^n)^{n^2} \cdot (4^n)^{n^2}$, which equals $\mathbf{2}^{3n^3}$.

The number of states of the Büchi automaton accepting $L_1 \cdot L_2^\omega$ equals the sum of the number of states of the automata accepting L_1 and L_2. Thus, for two equivalence classes L_1 and L_2 of $\sim_{\mathcal{M}}$, there is a Büchi automaton with $\mathbf{2}^{3n^3+1}$ states accepting $L_1 \cdot L_2^\omega$.

The number of states of the Büchi automaton accepting the union of ω-languages equals the sum of the number of states of the Büchi automata accepting the disjuncts. Since the number of pairs of equivalence classes of $\sim_{\mathcal{M}}$ is $\mathbf{2}^{4n^2}$, from Corollary 12.1, the next theorem follows.

Theorem 12.2 [Complexity of Büchi complementation] *Given a Büchi automaton $\mathcal{M}$ with n states, there exists a Büchi automaton $\mathcal{M}'$ with $\mathbf{2}^{3n^3+4n^2+1}$ states such that $\mathcal{L}_{\mathcal{M}'} = A^\omega\backslash\mathcal{L}_{\mathcal{M}}$.*

Note that to construct the desired complement automaton, we need to construct, for every pair L_1 and L_2 of equivalence classes of $\sim_{\mathcal{M}}$, the Büchi automaton accepting $L_1 \cdot L_2^\omega$, and check if it has a nonempty intersection with $\mathcal{L}_{\mathcal{M}}$. We have already outlined the product construction to obtain intersection of the languages accepted by two Büchi automata. Algorithms for checking fair cycles from Chapter 10 can be used to check for emptiness.

Remark 12.6 [Safra's Construction] The complementation construction presented here yields an automaton with $\mathbf{2}^{3n^3+4n^2+1}$ states. Better constructions are known. In particular, Safra's complementation algorithm produces an automaton with $\mathbf{2}^{O(n \cdot \log\ n)}$ states. This is essentially optimal: $\mathbf{2}^{n \cdot \log\ n}$ is a lower bound on the number of states necessary to define complements of a family of Büchi automata. ■

Complementing Streett automata

To establish that a Streett automaton can be complemented, we show that every Streett automaton has a language-equivalent Büchi automaton.

Proposition 12.5 [From Streett to Büchi] *Let $\mathcal{M}$ be a Streett automaton over A. There exists a Büchi automaton $\mathcal{M}'$ over A such that $\mathcal{L}_{\mathcal{M}} = \mathcal{L}_{\mathcal{M}'}$.*

Proof. Let $\mathcal{M} = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle, F)$ be a Streett automaton. Recall that an ω-trajectory $\underline{s}$ is F-fair iff there exists a subset $F' \subseteq F$ of Streett constraints and an index $i \geq 0$ such that (1) for every $(\sigma, \tau) \in F$, $\underline{s}$ is τ-fair, and (2) for every $(\sigma, \tau) \notin F$, $s_j \notin \sigma$ for all $j \geq i$.

Suppose F has ℓ Streett constraints. The Büchi automaton $\mathcal{M}'$ has $\mathbf{2}^\ell + 1$ copies of the observation structure of $\mathcal{M}$, an initial copy together with a copy for every subset $F' \subseteq F$ of Streett constraints of $\mathcal{M}$. The automaton starts in the initial copy, and at some point, guesses the set $F' \subseteq F$ of Streett constraints (σ, τ) such that the region τ is going to repeat infinitely many times, and switches to the copy corresponding to F'. The copy corresponding to the set F' ensures that, for every $(\sigma, \tau) \in F'$, τ is visited infinitely often, and for every $(\sigma, \tau) \notin F'$, σ is not visited. To enforce that, for every $(\sigma, \tau) \in F'$, τ is visited infinitely often, we introduce a counter as in the translation from multi-Büchi constraints to Büchi constraint. To enforce that, for every $(\sigma, \tau) \notin F'$, σ is not visited, we delete the states in σ. The formal definition of $\mathcal{M}'$ is left as an exercise. ■

Remark 12.7 [From Streett to Büchi] If $\mathcal{M}$ is a Streett automaton with n states and ℓ Streett constraints, the corresponding language-equivalent Büchi automaton constructed according to the proof of Proposition 12.5, has $n + n \cdot \ell \cdot \mathbf{2}^\ell$ states. Thus, simulating a set of Streett constraints by a single Büchi constraint causes a blow-up exponential in the number of constraints. Such a blow-up can be shown to be essential. ■

Given a Streett automaton, we can first construct the equivalent Büchi automaton, and then complement it using the complementation construction for Büchi automata.

Theorem 12.3 [Complementation of Streett automata] *Given a Streett automaton $\mathcal{M}$ with n states and ℓ Streett constraints, there exists a Büchi automaton $\mathcal{M}'$ with $\mathbf{2}^{O(n^3 \cdot 2^{3\ell})}$ states such that $\mathcal{L}_{\mathcal{M}'} = A^\omega \backslash \mathcal{L}_{\mathcal{M}}$.*

Exercise 12.6 {`T3`} [Complementing deterministic Streett automata] Let $\mathcal{M} = (K, F)$ be a Streett automaton such that K is a deterministic and complete observation structure. Show that if F is interpreted as a Rabin accepting condition, then the resulting language is the complement of $\mathcal{L}_{\mathcal{M}}$. ■

12.3 Expressiveness

12.3.1 ω-regular languages

In Chapter 10, we defined different ways of constructing ω-languages from languages of finite words. In particular, we defined the operators *safe*, *guar*, *recur*, and *persist*. If the languages to which these operators are applied are regular, then the resulting ω-languages are ω-regular.

ω-REGULAR LANGUAGES

The ω-language $\mathcal{L} \subseteq A^\omega$ is a *regular-safety language* if there is a regular language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{safe}(L)$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *regular-guarantee language* if there is a regular language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{guar}(L)$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *regular-response language* if there is a regular language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{recur}(L)$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *regular-persistence language* if there is a regular language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{persist}(L)$.

The ω-language $\mathcal{L} \subseteq A^\omega$ is *ω-regular* if it is a boolean combination of regular-response and regular-persistence languages.

Remark 12.8 [Normal form for ω-regular languages] Every ω-regular language is of the form

$$\bigcap 0 \leq i \leq k.\, \mathit{recur}(L_i) \cup \mathit{persist}(L'_i)$$

for regular languages L_i, L'_i. ■

Thus, the set of regular-safety languages is a subset of the the set of safety languages, etc. The set of ω-regular languages is a subset of the set of reactivity languages.

Example 12.7 [ω-regular languages] Let $A = \{a, b, c\}$. The ω-language $(Aa)^\omega$ is regular-safe; the ω-language A^*aA^ω is regular-guarantee; the ω-language $(A^*a)^\omega$ is regular-response; the ω-language A^*a^ω is regular-persistence; and the ω-language consisting of ω-words with infinitely many b symbols or only finitely many a symbols is ω-regular. The ω-language consisting of ω-words $\underline{a}$ such that for all $i \geq 0$, if i is a prime number, then $a_i = a$, is safe but not regular-safe. ■

Closure properties of regular-safety, regular-guarantee, regular-response, regular-persistence, and ω-regular languages coincide with the corresponding closure properties of safety, guarantee, response, persistence, and reactivity languages, respectively. In particular, regular-safety and regular-guarantee classes are duals of each other, and so are regular-response and regular-persistence classes. These properties are summarized in the following proposition, and its proof follows from the constructions of Section 10.1.3.

Proposition 12.6 [Closure properties of ω-regular languages] *Regular-safety languages, regular-guarantee languages, regular-response languages, and regular-persistence languages are closed under union and intersection, but not under complementation. The ω-regular languages are closed under all boolean operations.*

12.3.2 Expressiveness of ω-automata

The ω-language $\mathcal{L} \subseteq A^\omega$ is said to be a *Büchi language* if $\mathcal{L}$ is accepted by some Büchi automaton. First, let us note that Büchi languages are closed under all boolean operations:

Proposition 12.7 [Closure of Büchi languages] *The class of Büchi languages is closed under union, intersection, and complementation.*

Since Büchi acceptance is a special case of Streett acceptance, and by Proposition 12.5 is powerful enough to admit translation from Streett constraints, it follows that

Corollary 12.2 [Streett acceptance vs. Büchi acceptance] *The ω-language $\mathcal{L} \subseteq A^\omega$ is a a Büchi language iff $\mathcal{L}$ is accepted by some Streett automaton.*

Exercise 12.7 {`T3`} [Acceptance by fair structures] Show that the ω-language $\mathcal{L} \subseteq A^\omega$ is a Büchi language iff $\mathcal{L}$ is the fair language of some finite fair structure. ■

Recall that every ω-regular language is a boolean combination of regular-response languages. Every regular-response language is accepted by a deterministic Büchi automaton.

Proposition 12.8 [From regular-response to deterministic Büchi] *For every regular language L, there exists a Büchi automaton that accepts the response language $recur(L)$.*

Proof. Let L be a regular language. There exists a deterministic and complete automaton $M = (K, \sigma^A)$ such that $L_M = L$. Consider an ω-word $\underline{a}$. There exists precisely one initialized ω-trajectory $\underline{s}$ of K such that $\langle\langle \underline{s} \rangle\rangle = \underline{a}$. For every $i \geq 0$, the prefix $\overline{a}_{0\ldots i}$ belongs to L iff $s_i \in \sigma^A$. Hence, the ω-word $\underline{a}$ belongs to $recur(L)$ iff the ω-trajectory $\underline{s}$ is σ^A-fair. It follows that if we interpret M as a Büchi automaton it accepts the ω-language $recur(L)$. ■

Corollary 12.3 [Lower bound on Büchi expressiveness] *For every ω-regular language $\mathcal{L}$, there exists a Büchi automaton that accepts $\mathcal{L}$.*

Proof. Follows from the definition of ω-regular languages, Proposition 12.7, and Proposition 12.8. ■

Conversely, the language accepted by every ω-automaton is ω-regular:

Proposition 12.9 [Upper bound on Büchi expressiveness] *Every Büchi language is ω-regular.*

Proof. To be added. ■

Exercise 12.8 {T4} [Expressiveness of Muller automata] Show that an ω-language ω-regular iff it is accepted by a Muller automaton. ■

Exercise 12.9 {T4} [ω-regular expressions] We have been using ω-regular expressions (e.g. $(A^*a)^\omega$) to specify ω-languages. Let us now formally define the syntax of ω-regular expressions. Given a finite alphabet A, the set of ω-regular expressions is defined by the grammar

$$\varphi := a \mid \varphi \cdot \varphi \mid \varphi + \varphi \mid \varphi^* \mid \varphi^\omega$$

where $a \in A$. Show that an ω-language $\mathcal{L}$ is defined by an ω-regular expression iff $\mathcal{L}$ is ω-regular. ■

Exercise 12.10 {T3} [Expressiveness of CoBüchi automata] Does the class of languages accepted by CoBüchi automata coincide with ω-regular languages? ■

12.3.3 Deterministic ω-automata

We have established that the nondeterministic varieties of different types ω-automata accept the same class of languages, namely, ω-regular languages. We proceed to understand the expressive power of different types of deterministic ω-automata.

The next proposition shows that every ω-regular language is accepted by some deterministic Streett automaton, and both nondeterministic and deterministic varieties of Streett automata have the same expressive power, namely, ω-regular languages.

Proposition 12.10 [Expressiveness of deterministic Streett] *For every ω-regular language $\mathcal{L}$, there exists a deterministic Streett automaton that accepts $\mathcal{L}$.*

Proof. Let $\mathcal{L}$ be an ω-regular language. Suppose $\mathcal{L} = \bigcap 0 \le i \le k.\, (\mathit{recur}(L_i) \cup \mathit{persist}(L'_i))$ such that the languages L_i, L'_i are regular. For $0 \le i \le k$, let $M_i = (K_i, \sigma_i)$ be a complete deterministic automaton accepting the regular language L_i, and let $M'_i = (K'_i, \sigma'_i)$ be a complete deterministic automaton accepting the regular language L'_i. Let K be the product of the $2k+2$ observation structures K_i, K'_i, for $0 \le i \le k$. Since product construction preserves determinism, K is deterministic.

Let $\underline{a}$ be an ω-word. For every $0 \le i \le k$, the structure K_i has precisely one ω-trajectory $\underline{s}_i$ with the corresponding trace $\underline{a}$, and the word $\underline{a} \in \mathit{recur}(L_i)$ iff the ω-trajectory $\underline{s}_i$ is σ_i-fair. Similarly, for every $0 \le i \le k$, the structure K'_i has precisely one ω-trajectory $\underline{s}'_i$ with the corresponding trace $\underline{a}$, and the word

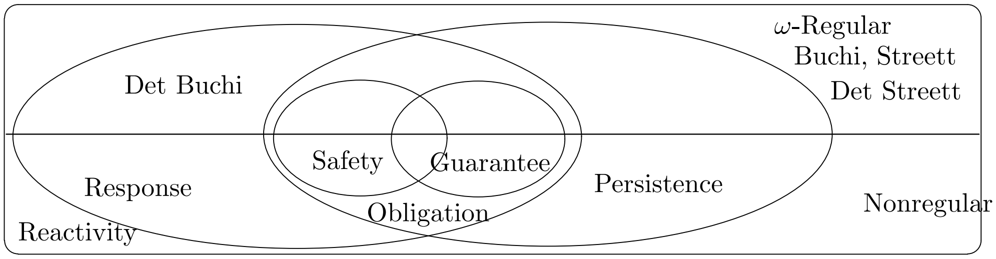


Figure 12.7: Classes of ω-regular languages

$\underline{a} \in \mathit{persist}(L'_i)$ iff the ω-trajectory $\underline{s}_i$ is not σ'_i-fair. The product structure K has precisely one ω-trajectory $\underline{s}$ with the trace $\underline{a}$. For all $0 \leq i \leq k$, the ω-trajectory $\underline{s}_i$ of K_i is σ_i-fair iff the ω-trajectory $\underline{s}$ of K is $(\sigma_i \Uparrow)$-fair (recall the definition of the lifting: $\sigma_i \Uparrow$ contains all product states whose component corresponding to the structure K_i is in σ_i). Similarly, for all $0 \leq i \leq k$, the ω-trajectory $\underline{s}'_i$ of K'_i is σ'_i-fair iff the ω-trajectory $\underline{s}$ of K is $(\sigma'_i \Uparrow)$-fair.

It follows that the word $\underline{a}$ belongs to $\mathcal{L}$ iff the ω-trajectory $\underline{s}$ of the product is, for all $0 \leq i \leq k$, either $(\sigma_i \Uparrow)$-fair or not $(\sigma'_i \Uparrow)$-fair. Hence, the deterministic Streett automaton $(K, \{(\sigma'_i \Uparrow, \sigma_i \Uparrow) \mid 0 \leq i \leq k\})$ accepts the ω-language $\mathcal{L}$. ■

Exercise 12.11 {T3} [Expressiveness of deterministic Rabin automata] Show that an ω-language is ω-regular iff it is accepted by some deterministic Rabin automaton. ■

It turns out that Büchi accepting condition is not expressive to capture all ω-regular languages, if we restrict to deterministic observation structures. The class of languages accepted by deterministic Büchi automata coincides with the regular-response languages.

Proposition 12.11 [Expressiveness of deterministic Büchi] *The ω-language $\mathcal{L} \subseteq A^\omega$ is accepted by a deterministic Büchi automaton iff it is a regular-response language.*

Proof. By Proposition 12.10, we know that every regular-response language is accepted by a deterministic Büchi automaton. For converse, consider a deterministic Büchi automaton $\mathcal{M} = (K, \sigma^A)$. Let L be the regular language accepted by the automaton with observation strcuture K and accepting region σ^A. Then, the Büchi automaton $\mathcal{M}$ accepts the regular-response language $\mathit{recur}(L)$. ■

It follows that deterministic Büchi automata are not closed under complementation. For instance, the response language "infinitely many a symbols" is accepted by a deterministic Büchi automaton, but its complement "only finitely many a symbols" is a persistence language, and is not accepted by any deterministic Büchi automaton. Intuitively, to define the persistence language consisting of ω-words with a suffix containing only b symbols using Büchi acceptance, the automaton must "guess" when the suffix containing only b symbols has commenced. The relationship between different classes of ω-languages is summarized in Figure 12.7

Exercise 12.12 {`T3`} [Union closure of deterministic Büchi automata] Since response languages are closed under union, from Proposition 12.11, it follows that deterministic Büchi automata are closed under union. Closure under union can, alternatively, be established by a direct construction. Give an algorithm to construct, given two deterministic Büchi automata $\mathcal{M}_1$ and $\mathcal{M}_2$, a deterministic Büchi automaton that accepts the union $\mathcal{L}_{\mathcal{M}_1} \cup \mathcal{L}_{\mathcal{M}_2}$. ■

Appendix: Notation

For two sets A and B, if σ is a subset of A then $\sigma \Uparrow$ denotes the subset $\{(a, b) \mid a \in \sigma\}$ of the product-set $A \times B$; if σ is a subset of B then $\sigma \Uparrow$ denotes the subset $\{(a, b) \mid b \in \sigma\}$ of the product-set $A \times B$. The lifting operator $\Uparrow$ can similarly be applied to binary relations over A to obtain binary relations over the product-set. The lifting operation generalizes to products of multiples sets also.

Contents

Chapter 13

Linear Temporal Logic

13.1 Linear Temporal Logic

In the last chapter, we studied how to use ω-automata to specify liveness requirements regarding infinite behaviors of fair modules. Such requirements can be alternatively, and more succinctly, specified using the temporal logic.

13.1.1 Syntax and Semantics

While the formulas of branching-time logics such as CTL are interpreted over trees, the formulas of *linear temporal logic* (LTL) are interpreted over ω-words over observations. As in case of CTL, the logic LTL employs temporal modalities such as *next*, *always*, *eventually*, and *until.* While in CTL every temporal connective has two types; existential and universal (e.g. possibly-next $\exists\bigcirc$ vs. inevitably-next $\forall\bigcirc$), such a distinction is not necessary in LTL whose formulas are interpreted over a fixed ω-word. On the other hand, while LTL admits nesting of temporal connectives freely, operators such $\Box\Diamond$ are not expressible in CTL.

LINEAR TEMPORAL LOGIC: SYNTAX

The formulas of *Linear Temporal Logic* (LTL) are defined inductively by the grammar

$$\varphi \ ::= \ p \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi\mathcal{U}\varphi$$

for observation predicates p.

The LTL formulas are interpreted over the positions of an infinite sequence of observations. Consider an ω-word $\underline{a}$ over the alphabet A whose symbols give interpretation to the predicates appearing in an LTL formula. The truth of an atomic predicate at the position i of $\underline{a}$ is evaluated according to the corresponding observation a_i of $\underline{a}$. The *next-formula* $\bigcirc\varphi$ holds at the position i of $\underline{a}$ iff the formula φ holds at the next-position $i+1$ of $\underline{a}$. The *until-formula* $\varphi_1\mathcal{U}\varphi_2$ holds at the position i of $\underline{a}$ iff theres exists a later position $j \geq i$ such that the formula φ_2 holds at the position j and the formula φ_1 holds at all positions k such that $i \leq k < j$.

For an LTL formula φ, the set A_φ contains the set of all possible valuations to the atomic predicates appearing in φ.

Linear temporal logic: Semantics

An Ltl formula φ is interpreted at the positions of ω-words over the set A_φ of observations. For all ω-words $\underline{a}$ and all $i \geq 0$,

$$\begin{array}{lll}
i \models_{\underline{a}} p & \text{iff} & a_i \models p; \\
i \models_{\underline{a}} \varphi_1 \vee \varphi_2 & \text{iff} & i \models_{\underline{a}} \varphi_1 \text{ or } i \models_{\underline{a}} \varphi_2; \\
i \models_{\underline{a}} \neg\varphi & \text{iff} & i \not\models_{\underline{a}} \varphi; \\
i \models_{\underline{a}} \bigcirc\varphi & \text{iff} & i+1 \models_{\underline{a}} \varphi; \\
i \models_{\underline{a}} \varphi_1 \mathcal{U} \varphi_2 & \text{iff} & \text{there is a natural number } j \geq i \text{ such that} \\
 & & j \models_{\underline{a}} \varphi_2 \text{ and for all } i \leq k < j,\ k \models_{\underline{a}} \varphi_1.
\end{array}$$

The ω-word $\underline{a}$ *satisfies* the Ltl formula φ, written $\underline{a} \models \varphi$, if $0 \models_{\underline{a}} \varphi$. The *$\omega$-language* $\mathcal{L}_\varphi$ defined by φ is the set $\{\underline{a} \in A_\varphi^\omega \mid \underline{a} \models \varphi\}$ of ω-words that satisfy φ. The Ltl formula φ is *satisfiable* if the ω-language $\mathcal{L}_\varphi$ is nonempty, and *valid* if the ω-language $\mathcal{L}_\varphi$ equals A_φ^ω.

The following temporal operators are defined in Ltl:

$$\begin{array}{lll}
\Diamond\varphi & \text{for} & \mathit{true}\,\mathcal{U}\,\varphi; \\
\Box\varphi & \text{for} & \neg\Diamond\neg\varphi; \\
\varphi_1 \mathcal{W} \varphi_2 & \text{for} & \varphi_1 \mathcal{U} \varphi_2 \vee \Box\varphi_1.
\end{array}$$

The temporal operators $\bigcirc$, $\Box$, $\Diamond$, $\mathcal{U}$, and $\mathcal{W}$ are called *next*, *always*, *eventually*, *until*, and *wait-for*, respectively. The eventually-formula $\Diamond\varphi$ holds at the position i of an ω-word $\underline{a}$ if the formula φ holds at some position $j \geq i$ of $\underline{a}$; the always-formula $\Box\varphi$ holds at the position i of an ω-word $\underline{a}$ if the formula φ holds at all positions $j \geq i$ of $\underline{a}$.

Remark 13.1 [Propositional Ltl] The Ltl formula φ is *propositional* if every atomic predicate in φ is a propositional formula. For a propositional Ltl formula φ, an observation in A_φ is a valuation for the set X_φ of boolean variables appearing in φ, and the set A_φ equals the power-set $\mathbf{2}^{X_\varphi}$. ■

Example 13.1 [Ltl languages] The Ltl formula $\Box p$ defines the safety language containing ω-words all of whose observations satisfy p. The Ltl formula $\Diamond p$ defines the guarantee language containing ω-words that contain an observation satisfying p. The Ltl formula $\Box\Diamond p$ defines the response language containing ω-words that contain infinitely many observations satisfying p. The Ltl formula $\Diamond\Box p$ defines the persistence language containing ω-words with a suffix with only observations satisfying p. Thus, the Ltl operators $\Box$, $\Diamond$, $\Box\Diamond$, and $\Diamond\Box$ correspond to the operators *safe*, *guar*, *recur*, and *persist*, respectively. ■

13.1.2 Ltl as a specification logic

We can view Ltl as a fair state logic, by interpreting Ltl formulas over the states of a given fair structure $\mathcal{K}$.

> Ltl semantics over fair structures
>
> Let φ be an Ltl formula, and let $\mathcal{K}$ be a fair structure with observations A_φ. For a state s of $\mathcal{K}$, $s \models_{\mathcal{K}} \varphi$ if all source-s fair ω-traces of $\mathcal{K}$ satisfy φ. The fair structure $\mathcal{K}$ *satisfies* the Ltl formula φ if $s \models_{\mathcal{K}} \varphi$ for all initial states s of $\mathcal{K}$.

Thus, a fair structure $\mathcal{K}$ satisfies an Ltl formula φ if every fair trace of $\mathcal{K}$ satisfies φ: $\mathcal{L}_{\mathcal{K}} \subseteq \mathcal{L}_\varphi$. The model checking problem and verification problem for Ltl are defined as in other logics.

Remark 13.2 [Interpretation of atoms] Let φ be an Ltl formula whose atomic predicates are boolean expressions over the set X_φ of variables. Then, the set A_φ is the set Σ_{X_φ} of valuations for X_φ. We can interpret the formula φ over a fair structure whose observations are valuations for a superset of the variables X_φ. ■

Example 13.2 [Ltl specifications for mutual exclusion] Recall the mutual exclusion problem from Chapter 2. The mutual exclusion requirement is specified by the Ltl formula

$$\varphi_{me}\colon\ \Box\,\neg(pc_1 = inC \ \wedge\ pc_2 = inC).$$

The first-request-first-in requirement that if process P_1 requests an entry to the critical section while process P_2 is outside, then P_2 cannot overtake P_1 to enter the critical section, is expressed by the formula

$$\varphi_{fifo}\colon\ \Box\,[\,(pc_1 = reqC \ \wedge\ pc_2 = outC) \ \rightarrow\ (pc_2 \neq inC)\,\mathcal{W}\,(pc_1 = inC)\,].$$

Finally, the starvation freedom requirement for process P_1 is specified by the formula

$$\varphi_{sf}\colon\ \Box\,(\,pc_1 = reqC \ \rightarrow\ \Diamond pc_1 = inC).$$

■

Remark 13.3 [Ltl specifications] The observation predicate p is an invariant of the module P iff the module P satisfies the Ltl formula $\Box p$; the observation predicate p is a recurrent of the fair module $\mathcal{P}$ iff the fair module $\mathcal{P}$ satisfies the Ltl formula $\Box\Diamond p$; and the observation predicate p is a response to the observation predicate q for the fair module $\mathcal{P}$ iff the fair module $\mathcal{P}$ satisfies the Ltl formula $\Box(p \rightarrow \Diamond q)$. ■

Example 13.3 [Producer-consumer requirements] Recall the message-passing protocols from Section 2.3.3, and their fair versions from Section 9.5.3. Let us consider the requirement that if the producer produces a message, say with value m, then eventually the consumer consumes a message with value m. Recall that the producer signals the production of the message by issuing the event $done_P$, and the produced message appears in the variable msg_P. The consumer signals the consumption of the message by issuing the event $done_C$, and the consumed message appears in the variable msg_C. Let $done_P?$ denote the Ltl formula $(done_P \leftrightarrow \bigcirc\neg done_P)$, and for a message value m, let $done_P?m$ denote the Ltl formula $(done_P \leftrightarrow \bigcirc\neg done_P) \wedge \bigcirc(msg_P = m)$. The abbreviations $done_C?$ and $done_C?m$ are defined analogously. Then, the desired requirement is specified by the formula

$$\varphi_{resp} : \bigwedge m \in \quad . \, \Box \, [\, done_P?m \;\to\; \Diamond \, done_C?m \,].$$

Verify that the fair module *FairSyncMsg* satisfies the requirement φ_{resp}.

Let us now consider the copy-requirement that, in every round i, if $\overline{a}$ denotes the (finite) sequence of messages produced by the producer so far, then (1) the sequence of messages consumed by the consumer upto round i is a prefix of $\overline{a}$, and (2) there exists a later round j such that the sequence of messages consumed by the consumer upto round j equals $\overline{a}$. The former is a safety requirement, while the latter is a liveness requirement. The Ltl formula φ_{resp} is only an approximation to the liveness part of the copy-requirement. It turns out that the copy-requirement is not expressible in Ltl. We can approximate it by verifying φ_{resp}, and many additional weaker requirements such as

$$\bigwedge m \in \quad . \, [\, \Box\Diamond \, done_P?m \;\leftrightarrow\; \Box\Diamond \, done_C?m \,],$$

which requires that the producer produces infinitely many messages with value m iff the consumer consumes infinitely many messages with value m, and

$$\bigwedge m \in \quad . \, [\, (\neg done_C?m) \; \mathcal{W} \; done_P?m \,]$$

which requires that the consumer does not consume a message with value m unless at least one such message is produced by the producer.

A requirement stronger than the copy-requirement stipulates strict alternation between production and consumption starting with the production, and is expressed by the Ltl formula $\varphi_{alternate}$:

$$\begin{array}{ll} & (done_C? \; \mathcal{W} \; done_P?) \\ \wedge & \wedge\, m \in \quad . \, \Box[\, done_P?m \;\to\; \bigcirc((\neg done_P?) \, \mathcal{U} \, done_C?m) \,] \\ \wedge & \wedge\, m \in \quad . \, \Box[\, done_C? \;\to\; \bigcirc((\neg done_C?) \, \mathcal{W} \, done_P?) \,] \end{array}$$

The fair module *FairSyncMsg* does not satisfy the requirement $\varphi_{alternate}$ even though *FairSyncMsg* satisfies the copy-requirement. This is because the producer may produce two messages before the consumer has consumed any message. ■

Exercise 13.1 {P3} [Monitor] Design a monitor module *CopyMonitor* whose variables keep track of the produced and consumed messages such that the safety aspect of the copy requirement reduces to an invariant verification problem for the compound module *SyncMsg* $\|$ *CopyMonitor* and the liveness aspect of the copy requirement reduces to a response verification problem for the compound fair module *FairSyncMsg* $\|$ *CopyMonitor*. ■

Like μ-calculus, fairness requirements can be specified within LTL. Let K be an observation structure, and consider a Büchi constraint specified by the observation predicate p. Then, an ω-trajectory $\underline{s}$ is fair iff the ω-trace $\langle\!\langle \underline{s} \rangle\!\rangle$ satisfies the LTL formula $\Box\Diamond p$. Suppose the fairness assumption requires fairness with respect to the action α specified as $[\![p \wedge q']\!]$ for two observation predicates p and q. Then, α-fair ω-trajectories are precisely those satisfying the LTL formula $\Box\Diamond(p \wedge \bigcirc q)$.

Now consider the Streett constraint (p, q) specified by two observation predicates p and q. Fairness of an ω-trajectory with respect to such a Streett constraint is specified by the LTL formula $\Box\Diamond p \rightarrow \Box\Diamond q$. Fairness with respect to multiple Streett constraints corresponds to conjunction of LTL formulas corresponding to individual Streett constraints.

Proposition 13.1 [Fairness specification in LTL] *Given a fair module $\mathcal{P}$, there exists an* LTL *formula $\varphi_{\mathcal{P}}$ such that an ω-trajectory $\underline{s}$ of the module $\mathcal{P}$ is a fair trajectory iff $\underline{s} \models \varphi_{\mathcal{P}}$.*

To verify all the fair trajectories of a fair module $\mathcal{P}$ satisfy a requirement ψ, we can verify whether all trajectories of the underlying module satisfy the implication $\varphi_{\mathcal{P}} \rightarrow \psi$.

Example 13.4 [Specifying fairness assumption in LTL] Recall the fairness constraints of the module *FairSyncMutex* from Figure 9.4. Requirement of weak fairness with respect to the update choices α_1 and α_2 is specified by the LTL formula $\varphi_{FairSyncMsg}$:

$$\Box\Diamond\, pc_1 \neq inC \;\wedge\; \Box\Diamond\, pc_2 \neq inC$$

Consider the specification φ_{sf} of starvation freedom from Example 13.2. The module *SyncMutex* does not satisfy the specification φ_{sf}, but satisfies the formula $\varphi_{FairSyncMutex} \rightarrow \varphi_{sf}$. ■

Exercise 13.2 {P2} [Mutual exclusion] Recall the fairness constraints of the module *FairPete* from Figure 9.5. Write down the LTL formula $\varphi_{FairPete}$ that captures the weak-fairness constraints of *FairPete*, and verify that the module *Pete* satisfies the LTL formula $\varphi_{FairPete} \rightarrow \varphi_{sf}$. ■

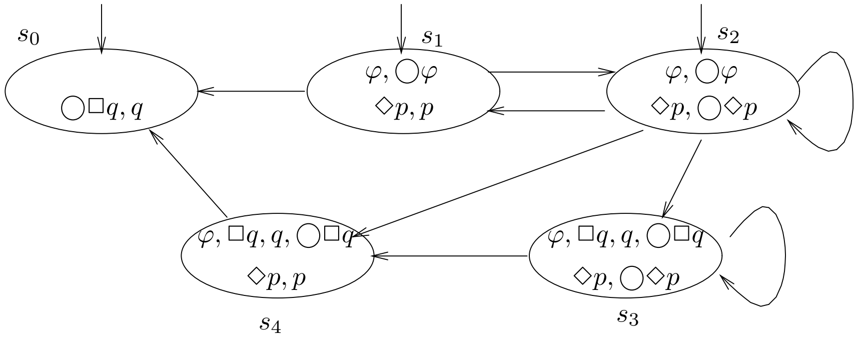

s_0
$\bigcirc\Box q, q$
s_1
$\varphi, \bigcirc\varphi$
$\Diamond p, p$
s_2
$\varphi, \bigcirc\varphi$
$\Diamond p, \bigcirc\Diamond p$
$\varphi, \Box q, q, \bigcirc\Box q$
$\Diamond p, p$
s_4
$\varphi, \Box q, q, \bigcirc\Box q$
$\Diamond p, \bigcirc\Diamond p$
s_3

Figure 13.1: Tableau construction for $(\Diamond p)\,\mathcal{U}\,(\Box q)$

13.2 Decision procedure

In this section, we give an algorithm for constructing, given an LTL formula φ, a Büchi automaton accepting the set $\mathcal{L}_\varphi$ of ω-words satisfying φ. This construction leads to a model checking algorithm for LTL.

13.2.1 Tableau Decision Procedure

Let φ be an LTL formula. We wish to construct a Büchi automaton $\mathcal{M}_\varphi$ over the alphabet A_φ such that an ω-word $\underline{a}$ is accepted by $\mathcal{M}_\varphi$ iff $\underline{a} \models \varphi$. States of the desired automaton are sets of subformulas of φ. Such an automaton is called a *tableau*.

Sample construction

To illustrate the principles of the tableau construction, let us consider the LTL formula $\varphi = (\Diamond p)\,\mathcal{U}\,(\Box q)$. The states of the tableau are collections of LTL formulas derived from φ. Each state s is to a set of formulas, and we would like to ensure that every formula contained in the state s is satisfied by every source-s ω-trajectory in the tableau. The initial states of the tableau are required to contain the given formula φ. From the semantics of the until-connective, an initial state satisfies φ if either (1) it satisfies $\Box q$, or (2) it satisfies both $\bigcirc\varphi$ and $\Diamond p$. In the former case, to satisfy $\Box q$, the initial state should also satisfy q as well as $\bigcirc\Box q$, and this gives the initial state $s_0 = \{\varphi, \Box q, \bigcirc\Box q, q\}$ (see Figure 13.1). In the latter case, $\Diamond p$ can be satisfied by either p, or by $\bigcirc\Diamond p$. The corresponding initial states are $s_1 = \{\varphi, \bigcirc\varphi, \Diamond p, p\}$ and $s_2 = \{\varphi, \bigcirc\varphi, \Diamond p, \bigcirc\Diamond p\}$.

To obtain successors of a state, we examine the next-formulas in the state. For every $\bigcirc\psi$ contained in the current state, the successor should contain ψ. Since s_0 contains $\bigcirc\Box q$, its successor is required to contain $\Box q$, and hence, we add a transition from s_0 to itself. The successors of s_1 are required to contain φ.

Since φ can be satisfied in three ways, all the initial states are successors of s_1. The successors of s_2 are required to contain both φ and $\Diamond p$. The formula φ can be satisfied in three ways, while $\Diamond p$ can be satisfied either by p or by $\bigcirc\Diamond p$. Continuing in this manner, we get the tableau of Figure 13.1 with five states.

We would like to ensure that if $\underline{t}$ is an infinite trajectory in the tableau, then for every position $i \geq 0$ and every formula $\psi \in t_i$, $i \models_{\underline{t}} \psi$. This is not quite true yet. For instance, in the ω-trajectory s_2^ω that corresponds to looping forever at the state s_2, every state contains $\Diamond p$, but no state satisfies p. Intuitively, along this ω-trajectory the choice to satisfy $\Diamond p$ is postponed forever. This can be avoided by adding Büchi constraints. In this example, we need two Büchi constraints since there are two eventualities. The first constraint requires that to satisfy φ, one must satisfy $\Box q$ eventually. This is expressed by the Büchi constraint $\sigma_\varphi = \{s_0, s_3, s_4\}$ containing states that contain $\Box q$. The second constraint requires that to satisfy $\Diamond p$, one must satisfy p eventually. This is expressed by the Büchi constraint $\sigma_{\Diamond p} = \{s_0, s_1, s_4\}$ containing states that either contain p or do not contain $\Diamond p$. Verify that, for the tableau of Figure 13.1 together with the multi-Büchi assumption $\{\sigma_\varphi, \sigma_{\Diamond p}\}$, the set of fair trajectories corresponds to the set of ω-words that satisfy φ.

In summary, in a tableau construction, states are subsets of formulas. Each formula stipulates requirements concerning other formulas that must be satisfied in the current state. The transition relation ensures propagation of the next-formulas from one state to its successor. Fairness constraints ensure eventual fulfillment of eventuality- or until-formulas. We proceed to formalize the tableau construction. In our example, we treated formulas that do not appear in a state as "don't care." For instance, in the initial state s_0, there is no mention of the formulas $\Diamond p$ and p. In the formal construction, each state assigns a truth to every subformula.

Closure

The *closure* $\mathit{Sub}(\varphi)$ of the Ltl formula φ is defined inductively as

$$\begin{array}{lcl}
\mathit{Sub}(p) & = & \{p\}; \\
\mathit{Sub}(\varphi_1 \vee \varphi_2) & = & \{\varphi_1 \vee \varphi_2\} \cup \mathit{Sub}(\varphi_1) \cup \mathit{Sub}(\varphi_2); \\
\mathit{Sub}(\neg\varphi) & = & \{\neg\varphi\} \cup \mathit{Sub}(\varphi); \\
\mathit{Sub}(\bigcirc\varphi) & = & \{\bigcirc\varphi\} \cup \mathit{Sub}(\varphi); \\
\mathit{Sub}(\varphi_1 \mathcal{U} \varphi_2) & = & \{\varphi_1 \mathcal{U} \varphi_2, \bigcirc(\varphi_1 \mathcal{U} \varphi_2)\} \cup \mathit{Sub}(\varphi_1) \cup \mathit{Sub}(\varphi_2).
\end{array}$$

Notice that the closure of a formula φ contains more than the syntactic subformulas of φ, namely, if an until-formula $\varphi_1 \mathcal{U} \varphi_2$ is in the closure, then so is the next-formula $\bigcirc(\varphi_1 \mathcal{U} \varphi_2)$.

Proposition 13.2 [Size of closure] *For every* Ltl *formula φ, $\varphi \in \mathit{Sub}(\varphi)$ and $|\mathit{Sub}(\varphi)| \leq 2|\varphi|$.*

Tableau

A subset $s \subseteq \mathit{Sub}(\varphi)$ of the closure of φ is *consistent* if the following conditions are satisfied:

$$\begin{array}{llll}
\text{if } (\chi_1 \vee \chi_2) \in \mathit{Sub}(\varphi) \text{ then} & (\chi_1 \vee \chi_2) \in s & \text{iff} & \chi_1 \in s \text{ or } \chi_2 \in s, \\
\text{if } \neg\chi \in \mathit{Sub}(\varphi) \text{ then} & \neg\chi \in s & \text{iff} & \chi \notin s, \\
\text{if } (\chi_1 \mathcal{U} \chi_2) \in \mathit{Sub}(\varphi) \text{ then} & (\chi_1 \mathcal{U} \chi_2) \in s & \text{iff} & \text{either } \chi_2 \in s \text{ or both} \\
 & & & \chi_1 \in s \text{ and } \bigcirc(\chi_1 \mathcal{U} \chi_2) \in s.
\end{array}$$

LTL TABLEAU

Given an LTL formula φ the *φ-tableau* is the multi-Büchi automaton $\mathcal{M}_\varphi$:

- The state-space of $\mathcal{M}_\varphi$ is the set Σ_φ of consistent subsets of $\mathit{Sub}(\varphi)$.
- The transition relation of $\mathcal{M}_\varphi$ is the relation $\rightarrow_\varphi$: for $s, t \in \Sigma_\varphi$, $s \rightarrow_\varphi t$ if for all formulas $\bigcirc\chi \in \mathit{Sub}(\varphi)$, $\bigcirc\chi \in s$ iff $\chi \in t$.
- A state $s \in \Sigma_\varphi$ is initial iff $\varphi \in s$.
- The set of observations is the set A_φ of valuations of the atomic predicates in φ.
- The observation of a state s is the set of atomic predicates in s (that is, $\langle\!\langle s \rangle\!\rangle \models p$ iff $p \in s$).
- For each until-formula $(\chi_1 \mathcal{U} \chi_2) \in \mathit{Sub}(\varphi)$, the fairness assumption of $\mathcal{M}_\varphi$ contains the Büchi constraint

$$\{s \in \Sigma_\varphi \mid \chi_2 \in s \text{ or } (\chi_1 \mathcal{U} \chi_2) \notin s\}.$$

Proposition 13.3 [Correctness of tableau construction] *For every* LTL *formula* φ, $\mathcal{L}_\varphi$ *equals* $\mathcal{L}_{\mathcal{M}_\varphi}$.

Proof. Let φ be an LTL formula, and let $\underline{a}$ be an ω-word over the set A_φ of observations. Suppose $\underline{a} \models \varphi$. For $i \geq 0$, let $s_i \subseteq \mathit{Sub}(\varphi)$ be the set $\{\chi \in \mathit{Sub}(\varphi) \mid i \models_{\underline{a}} \chi\}$ of formulas true at position i in $\underline{a}$. From the definitions, it follows that (1) for all i, the set s_i is consistent, (2) for all i, $s_i \rightarrow_\varphi s_{i+1}$, (3) the set s_0 is an initial state of $\mathcal{M}_\varphi$, (4) for all i, the observation of the state s_i in $\mathcal{M}_\varphi$ is a_i, and (5) for each $(\chi_1 \mathcal{U} \chi_2) \in \mathit{Sub}(\varphi)$, if $i \models_{\underline{a}} \chi_1 \mathcal{U} \chi_2$ for infinitely many positions i, then $j \models_{\underline{a}} \chi_2$ for infinitely many positions j. It follows that $\underline{s}$ is a fair initialized ω-trajectory of the tableau $\mathcal{M}_\varphi$, and $\underline{a}$ belongs to $\mathcal{L}_{\mathcal{M}_\varphi}$.

Now consider an initialized fair ω-trajectory $\underline{s}$ of $\mathcal{M}_\varphi$. Let $\underline{a}$ be the corresponding ω-trace. We want to establish that for all $\chi \in \mathit{Sub}(\varphi)$, for all $i \geq 0$, $\chi \in s_i$ iff $i \models_{\underline{a}} \chi$. The proof is by induction on the structure of χ, and is left as an exercise. It follows that $\underline{a} \models \varphi$. ■

The number of states of the tableau of an LTL formula is exponential in the length of the formula: for an LTL formula φ of length k, the automaton $\mathcal{M}_\varphi$ has at most 4^k states and k Büchi constraints. Checking satisfiability of the LTL formula φ corresponds to checking whether the fair language of the automaton $\mathcal{M}_\varphi$ is nonempty, and thus, can be solved in time $\mathbf{2}^{O(|\varphi|)}$. Checking validity of the LTL formula corresponds to checking satisfiability of the negated formula $\neg\varphi$, and thus, can also be solved in time $\mathbf{2}^{O(|\varphi|)}$.

Exercise 13.3 {T3} [On-the-fly tableau] In our definition of the tableau $\mathcal{M}_\varphi$, every state assigns a truth value to every formula in $Sub(\varphi)$. As indicated in our example (see Figure 13.1), not every formula needs to be evaluated in every state. Develop an algorithm to construct a tableau for the input formula that considers the formulas in the closure only as needed. ■

13.2.2 LTL Model Checking

Let $\mathcal{K}$ be a fair structure, and let φ be an LTL formula. The model checking problem $(\mathcal{K}, \varphi)$ corresponds to verifying that the ω-language $\mathcal{L}_\mathcal{K}$ is contained in the ω-language $\mathcal{L}_\varphi$, which, by the tableau-construction, corresponds to the ω-language inclusion problem $(\mathcal{K}, \mathcal{M}_\varphi)$. Observe that the tableau $\mathcal{M}_\varphi$ is a nondeterministic ω-automaton, and hence, solving the ω-language inclusion problem $(\mathcal{K}, \mathcal{M}_\varphi)$ is computationally hard, namely, exponential in the size of the structure $\mathcal{M}_\varphi$.xs However, we can avoid the complementation construction for ω-automata if, instead of constructing the tableau for φ, we construct the tableau $\mathcal{M}_{\neg\varphi}$ for the negated formula $\neg\varphi$. The ω-automaton $\mathcal{M}_{\neg\varphi}$ accepts all ω-words that do not satisfy the specification φ. Now, the fair structure $\mathcal{K}$ satisfies φ iff the intersection of the ω-languages of $\mathcal{K}$ and $\mathcal{M}_{\neg\varphi}$ is empty. This approach of negating the formula *before* applying the tableau-construction avoids the need for complementing the tableau.

Proposition 13.4 [LTL model checking] *The answer to the* LTL *model checking problem $(\mathcal{K}, \varphi)$ is* YES *iff the answer to the fair-emptiness problem $\mathcal{K} \times \mathcal{M}_{\neg\varphi}$ is* YES*.*

Since we already know how to obtain product of two ω-automata, and we know how to solve the fair emptiness problem for finite structures, we have an algorithm for LTL model checking.

Theorem 13.1 [LTL model checking] *If φ is an* LTL *formula of length k and $\mathcal{K}$ is a fair structure with n states, m transitions, and ℓ fairness constraints, then the* LTL *model checking problem $(\mathcal{K}, \varphi)$ can be solved in time $O((m+n) \cdot \ell^2 \cdot k \cdot \mathbf{2}^{O(k)})$.*

Remark 13.4 [Ltl model checking of weak-fair structures] If $\mathcal{K}$ is a weak-fair structure, then so is the product $\mathcal{K} \times \mathcal{M}_{\neg\varphi}$. If $\mathcal{K}$ has ℓ weak-fairness constraints, and $Sub(\varphi)$ has k until-formulas, then the product has $\ell + k$ weak-fairness constraints. It follows that the the Ltl model checking problem $(\mathcal{K}, \varphi)$, for a weak-fair structure, can be solved in time $O((m+n) \cdot (\ell + k) \cdot \mathbf{2}^{O(k)})$. ■

A variety of heuristics can be used to improve the computational requirements of Ltl model checking. In particular, an on-the-fly representation of the structure is used, and the product with the tableau is generated only during the search.

Remark 13.5 [Ltlvs. Ctl] Recall that the Ctl model-checking problem $(\mathcal{K}, \phi)$, for a Ctl formula ϕ of length k, can be solved in time $O((m+n) \cdot \ell^2 \cdot k)$. Thus, while Ctl model checking is linear in the size of the formula, Ltl model checking is exponential in the size of the formula. However, both model checking problems have identical model complexity. ■

If $\mathcal{P}$ is a propositional fair module with n boolean variables and φ is an Ltl formula of length k, then the Ltl verification problem $(\mathcal{P}, \varphi)$ can be solved in time exponential in $n + k$.

13.2.3 Complexity

The Ltl satisfiability problem is to determine whether a given Ltl formula φ is satisfiable. Checking satisfiability of φ corresponds to checking emptiness of the tableau $\mathcal{M}_{\varphi}$. Search for a reachable fair cycle in the tableau can be performed using space logarithmic in the number of states of the tableau, or equivalently, linear in the size of the formula φ. It follows that the Ltl satisfiability problem is in PSPACE. It turns out that checking satisfiability of Ltl formulas is PSPACE-hard.

Theorem 13.2 [Ltl complexity] *The satisfiability and the validity problems for* Ltl *are PSPACE-complete.*

Proof. An Ltl formula φ is satisfiable iff the formula $\neg\varphi$ is valid. It remains to be shown that the satisfiability problem is PSPACE-hard. Proof to be added. ■

Hardness of the satisfiability problem implies a lower bound for the model checking problem also.

Theorem 13.3 [Ltl model checking comlexity] *The* Ltl *model checking problem $(\mathcal{K}, \varphi)$, for a finite fair structure $\mathcal{K}$, is PSPACE-complete.*

Proof. The Ltl model checking problem $(\mathcal{K}, \varphi)$, for a finite fair structure $\mathcal{K}$, reduces to searching for a reachable fair cycle of the product $\mathcal{K} \times \mathcal{M}_{\neg\varphi}$. The search can be performed in space logarithmic in the number of states of the product, and PSPACE upper bound follows.

For lower bound, we reduce the LTL validity problem to LTL model checking. ■

Remark 13.6 [LTL model complexity] The model complexity of the LTL model checking problem is NLOGSPACE. The complexity of the LTL verification problem $(\mathcal{P}, \varphi)$, for a propositional fair module $\mathcal{P}$, is PSPACE. ■

Exercise 13.4 {T3} [LTL without next] Show that the decision and model checking problems for LTL without the next operator are still PSPACE-hard. ■

Exercise 13.5 {T3} [LTL$^\diamond$] Let LTL$^\diamond$ result from LTL by replacing the until operator $\mathcal{U}$ with the eventually operator $\diamond$. Prove that the decision problem for LTL$^\diamond$ is still PSPACE-hard. ■

Exercise 13.6 {T3} [LTL$^\diamond$ without next] Show that if an LTL$^\diamond$ formula φ without next operators is satisfiable, then it is satisfiable by an ω-word of the form $\overline{a}_1\overline{a}_2^\omega$ such that $|\overline{a}_1\overline{a}_2| \leq |\varphi|$. Then prove that the decision problem for LTL$^\diamond$ without the next operator is NP-complete. ■

Exercise 13.7 {T3} [LTL with past operators] The syntax of *Past* LTL is defined as

$$\varphi \ ::= \ p \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1\mathcal{U}\varphi_2 \mid \ominus\varphi \mid \varphi_1\mathcal{S}\varphi_2.$$

The semantics of the *previous* and *since* operators are defined as

$i \models_{\overline{a}} \ominus\varphi$ iff $i > 0$ and $i - 1 \models_{\overline{a}} \varphi$;
$i \models \varphi_1\mathcal{S}\varphi_2$ iff for some $j \leq i$, $j \models_{\overline{a}} \varphi_2$ and for all $j < k \leq i$, $k \models_{\overline{a}} \varphi_1$.

Give a tableau-based decision procedure for Past LTL and then prove that the decision problem for Past LTL is complete for PSPACE. ■

Exercise 13.8 {T3} [Counting LTL] The syntax of *Counting* LTL is defined as

$$\varphi \ ::= \ p \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \bigcirc^n\varphi \mid \varphi_1\mathcal{U}\varphi_2$$

for natural numbers n (represented in logarithmic notation, e.g., binary or decimal). The semantics of the *counting* operator is defined as

$i \models_{\overline{a}} \bigcirc^n\varphi$ iff $i + n \models_{\overline{a}} \varphi$.

Give a tableau-based decision procedure for Counting LTL and then prove that the decision problem for Counting LTL is complete for EXPSPACE. ■

13.3 Expressiveness

13.3.1 Linear-time versus branching-time

How do the expressive powers of the branching-time logic Ctl and the linear time logic Ltl compare? We have already seen that Ctl cannot express operators such as $\Box\Diamond$, and thus, cannot be more expressive than Ltl. For the converse, while the state equivalence induced by Ctl coincides with bisimilarity, the state equivalence induced by Ltl coincides with trace-equivalence:

Proposition 13.5 [State equivalence of Ltl] *Trace-equivalence is a fully abstract semantics for* Ltl *over observation structures.*

Since Ctl can distinguish between two states that are trace-equivalent, but not bisimilar, Ltl cannot be more expressive that Ctl.

Proposition 13.6 [Expressive power of Ltl vs. Ctl] *The expressive powers of the temporal logics* Ctl *and* Ltl *are incomparable. In particular, no* Ltl *formula is equivalent to the* Ctl *formula* $\forall\bigcirc\,\exists\bigcirc\,p$*, and no* Ctl *formula is equivalent to the* Ltl *formula* $\Box\Diamond p$*.*

The deficiency of Ltl compared to Ctl is the lack of existential quantification over trajectories, while the deficiency of Ctl compared to Ltl is the inability to nest temporal connectives to express requirements regarding a fixed trajectory. This motivates the definition of a temporal logic that is more expressive than both Ctl and Ltl.

Before we define the combination of Ctl and Ltl, let us understand the distinction between state formulas and trajectory formulas. While the formulas of a (fair) state logic—*state formulas*—are interpreted over the states of a (fair) observation structure, the formulas of a trajectory logic—*trajectory formulas*—are interpreted over the ω-trajectories of a (fair) observation structure. We can view Ctl as a two-sorted logic with state formulas ϕ and trajectory formulas φ:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists\varphi$$

$$\varphi ::= \bigcirc\phi \mid \phi_1 \mathcal{U} \phi_2 \mid \Box\phi$$

where p is an atomic formula. On the other hand, we can view Ltl as a two-sorted logic with state formulas ϕ and trajectory formulas φ:

$$\phi ::= \forall\varphi$$

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \phi_1 \mathcal{U} \phi_2$$

where p is an atomic formula. The logic Ctl* allows the state-formulas as in Ctl and trajectory formulas as in Ltl.

Temporal logic Ctl*: Syntax

The formulas of Ctl* are defined inductively by the two-sorted grammar with state formulas ϕ and trajectory formulas φ:

$$\phi \ ::= \ p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists\varphi$$

$$\varphi \ ::= \ \phi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

where p is an atomic formula.

Given a fair structure $\mathcal{K}$ whose observations are valuations to the atomic predicates, state-formulas of Ctl* are interpreted at states of $\mathcal{K}$, while the trajectory-formulas of Ctl* are interpreted at positions of the fair trajectories of $\mathcal{K}$.

Temporal logic Ctl*: Semantics

Let $\mathcal{K} = (K, F)$ be a fair structure. For each state s of $\mathcal{K}$,

$$\begin{array}{lll}
s \models_{\mathcal{K}} p & \text{iff} & \langle\!\langle s \rangle\!\rangle \models p; \\
s \models_{\mathcal{K}} \neg\phi & \text{iff} & s \not\models_{\mathcal{K}} \phi; \\
s \models_{\mathcal{K}} \phi_1 \vee \phi_2 & \text{iff} & s \models_{\mathcal{K}} \phi_1 \text{ or } s \models_{\mathcal{K}} \phi_2; \\
s \models_{\mathcal{K}} \exists\varphi & \text{iff} & \text{there is a source-}s\ F\text{-fair } \omega\text{-trajectory } \underline{s} \\
 & & \text{of } K \text{ such that } (\underline{s}, 0) \models_{\mathcal{K}} \varphi.
\end{array}$$

For each ω-trajectory $\underline{s}$ of $\mathcal{K}$ and each position $i \geq 0$,

$$\begin{array}{lll}
(\underline{s}, i) \models_{\mathcal{K}} \phi & \text{iff} & s_i \models_{\mathcal{K}} \phi; \\
(\underline{s}, i) \models_{\mathcal{K}} \neg\varphi & \text{iff} & (\underline{s}, i) \not\models_{\mathcal{K}} \varphi; \\
(\underline{s}, i) \models_{\mathcal{K}} \varphi_1 \vee \varphi_2 & \text{iff} & (\underline{s}, i) \models_{\mathcal{K}} \varphi_1 \text{ or } (\underline{s}, i) \models_{\mathcal{K}} \varphi_2; \\
(\underline{s}, i) \models_{\mathcal{K}} \bigcirc\varphi & \text{iff} & (\underline{s}, i+1) \models_{\mathcal{K}} \varphi; \\
(\underline{s}, i) \models_{\mathcal{K}} \varphi_1 \mathcal{U} \varphi_2 & \text{iff} & \text{there exist } j \geq i \text{ such that } (\underline{s}, j) \models_{\mathcal{K}} \varphi_2 \\
 & & \text{and for all } i \leq k < j,\ (\underline{s}, k) \models_{\mathcal{K}} \varphi_1.
\end{array}$$

The fair structure $\mathcal{K}$ satisfies the Ctl* formula ϕ if $s \models_{\mathcal{K}} \phi$ for every initial state s of $\mathcal{K}$.

The following operators are defined in Ctl*:

$$\begin{array}{lll}
\forall\varphi & \text{for} & \neg\exists\neg\phi; \\
\Diamond\varphi & \text{for} & \mathit{true}\mathcal{U}\varphi; \\
\Box\varphi & \text{for} & \neg\Diamond\neg\varphi; \\
\varphi_1 \mathcal{W} \varphi_2 & \text{for} & \varphi_1 \mathcal{U} \varphi_2 \ \vee \ \Box\varphi_1.
\end{array}$$

Remark 13.7 [Ctl* vs. Ltl and Ctl] Since every Ctl formula is a Ctl* formula, and for every Ltl formula φ, the equivalent Ctl* formula is $\forall\varphi$, it follows that Ctl* is more expressive power than both Ltl and Ctl. ■

While the expressive power of CTL* is more than CTL, the equivalence induced by CTL* coincides with bisimilarity, and thus, its distinguishing power coincides with CTL.

Proposition 13.7 [Equivalence induced by CTL*] *Bisimilarity is a fully abstract semantics of* CTL* *over observation structures.*

Exercise 13.9 {T3} [CTL$^+$] The formulas of CTL$^+$ are defined inductively by the two-sorted grammar

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists\varphi$$

$$\varphi ::= \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\phi \mid \phi_1 \mathcal{U} \phi_2$$

where p is an atomic formula. For example, for two propositions a and b, the CTL$^+$ formula $\exists(\Diamond a \wedge \Diamond b)$ is equivalent to the CTL formula

$$\exists\Diamond(a \wedge \exists\Diamond b) \vee \exists\Diamond(b \wedge \exists\Diamond a).$$

Give a systematic construction that yields for each CTL$^+$ formula ϕ an equivalent CTL formula ϕ^-. For your construction, what is the blowup in the size of the formula? (If ϕ has length k, give an asymptotic bound for the length of ϕ^- as a function of k.) ■

Exercise 13.10 {T3} [CTL$_\Diamond$] Let CTL$_\Diamond$ be obtained from CTL by replacing the binary operator $\exists\mathcal{U}$ with the unary operator $\exists\Diamond$, and let CTL$_\Diamond^+$ be obtained from CTL$^+$ by replacing the binary operator $\mathcal{U}$ with the unary operator $\Diamond$. Consider two propositions a and b. Prove that no CTL$_\Diamond$ formula is equivalent to the CTL$_\Diamond^+$ formula $\exists(\Box a \wedge \Diamond b)$ over finite observation structures, and that no CTL$_\Diamond^+$ formula is equivalent to the CTL formula $a\exists\mathcal{U}b$. We conclude that CTL$_\Diamond$ $\prec$ CTL$_\Diamond^+$ $\prec$ CTL $\equiv$ CTL$^+$ $\prec$ CTL*. ■

CTL* **model checking**

Theorem 13.4 [CTL* model checking] *If ϕ is a* CTL* *formula of length k and $\mathcal{K}$ is a fair structure with n states, m transitions, and ℓ fairness constraints, then the* CTL* *model checking problem $(\mathcal{K}, \phi)$ can be solved in time $O((m+n) \cdot \ell^2 \cdot \mathbf{2}^{O(k)})$.*

Remark 13.8 [Space complexity of CTL*] For finite fair structures, the CTL* model checking problem is PSPACE-complete. The structure complexity of CTL* model checking is NLOGSPACE. Thus, the space complexity of CTL* model checking coincides with the space complexity of LTL model checking. The CTL* verification problem $(\mathcal{P}, \phi)$ is PSPACE-complete. ■

13.3.2 Ltl versus ω-automata

The tableau construction establishes that, for every Ltl formula φ, the ω-language $\mathcal{L}_\varphi$ is ω-regular. The converse does not hold. In particular, the property that requires p to be true at every even position is not specifiable in Ltl, while it is speciable using automata as indicated in Figure 7.6.

Proposition 13.8 [Ltl cannot express even] *For $A = \{a, b\}$, the ω-regular language $\mathcal{L}_{even}$ that contains the ω-word $\underline{c}$ iff $c_i = a$ for all even numabers i, is not expressible in* Ltl*.*

Corollary 13.1 [Ltl vs. Lal] *The live automaton logic* Lal *is more expressive than* Ltl*.*

While Lal is more expressive than Ltl, the distinguishing powers of the two logics coincide, both logics induce trace-equivalence over observation structures.

Exercise 13.11 {T4} [Periodic Ltl] The syntax of *Periodic* Ltl is defined as

$$\varphi \ ::= \ p \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U}_n \varphi_2$$

for natural numbers n. The semantics of the *periodic-until* operator is defined as

$$i \models_{\overline{a}} \varphi_1 \mathcal{U}_n \varphi_2 \text{ iff for some } j \geq 0,\ i + jn \models_{\overline{a}} \varphi_2 \text{ and for all } 0 \leq k < j,\\ i + kn \models_{\overline{a}} \varphi_1.$$

(1) Express the property that "proposition x is true at every even position of an observation sequence" in Periodic Ltl. (2) Give a tableau-based decision procedure for Periodic Ltl. (3) What is the complexity of the decision problem for Periodic Ltl if numerals are represented in unary (binary, respectively)? ■

Exercise 13.12 {T4} [Linear-time μ-calculus] The formulas of the *Linear μ-calculus* Ltμ are defined as

$$\varphi \ ::= \ p \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \mu X.\, \varphi' \mid X$$

where p is an observation predicate, $X \in \mathcal{P}$ is a formula variable, and each free occurrence of X in φ' occurs within an even number of negations. The linear μ-calculus is interpreted over infinite observation sequences. Give a formal definition of the semantics of Ltμ such that (1) the Ltl formula $\varphi_1 \mathcal{U} \varphi_2$ is equivalent to the Ltμ formula $(\mu X.\, \varphi_2 \vee (\varphi_1 \wedge \bigcirc X))$ and (2) the Ltμ formula $(\nu X.\, y \wedge \bigcirc \bigcirc X)$ is satisfied by the ω-word $\underline{a}$ iff the proposition y is true at every even-numbered position of $\overline{a}$. When defining the semantics of Ltμ, you need to show that all required fixpoints exist (What is the underlying c.p.o.? Why are all definable functions monotonic?) and that (1) and (2) are indeed the case. Then prove that all definable functions are continuous. ■

Monadic second-order logic S1S

The fragment S1S of second-order logic is a classical notation to define ω-languages. The logic S1S allows first-order variables that range over nonnegative integers, and its terms are built from the first-order variables using the successor function "+1" that corresponds to adding 1. The formulas are built using logical connectives, first-order and second-order quantifiers, comparing terms using the ordering $<$, and second-order unary variables.

Formally, the set of terms of S1S is generated by the grammar

$$e \ := \ 0 \mid i \mid e+1,$$

where i is a first-order variable. The set of formulas of S1S is generated by the grammar

$$\varphi \ := \ p(e) \mid e < e \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists i.\, \varphi \mid \exists p.\, \varphi,$$

where p is a second-order variable.

Formulas of S1S are evaluated with respect to environments that map first-order variables to nonnegative integers and second-order variables to sets of nonnegative integers. Let $\mathcal{E}$ be an environment that maps first-order variables to and second-order variables to $\mathbf{2}^{\mathbb{N}}$. Then, $\mathcal{E}$ maps terms of S1S to : $\mathcal{E}(0) = 0$ and $\mathcal{E}(e+1) = \mathcal{E}(e)+1$. The satisfaction relation $\models_{\mathcal{E}}$ for the formulas of S1S is defined inductively:

$$\begin{array}{lll}
\models_{\mathcal{E}} p(e) & \text{iff} & \mathcal{E}(e) \in \mathcal{E}(p); \\
\models_{\mathcal{E}} e_1 < e_2 & \text{iff} & \mathcal{E}(e_1) < \mathcal{E}(e_2); \\
\models_{\mathcal{E}} \neg\varphi & \text{iff} & \not\models_{\mathcal{E}} \varphi; \\
\models_{\mathcal{E}} \varphi_1 \vee \varphi_2 & \text{iff} & \models_{\mathcal{E}} \varphi_1 \text{ or } \models_{\mathcal{E}} \varphi_2; \\
\models_{\mathcal{E}} \exists i.\, \varphi & \text{iff} & \text{for some } j \in \ , \models_{\mathcal{E}[i:=j]} \varphi; \\
\models_{\mathcal{E}} \exists p.\, \varphi & \text{iff} & \text{for some } \sigma \subseteq \ , \models_{\mathcal{E}[p:=\sigma]} \varphi.
\end{array}$$

The unary predicates in S1S formulas can be viewed as boolean variables. If X consists of boolean variables, and $\underline{a}$ is an ω-word over the valuations for X, then, for every $x \in X$, the ω-word $\underline{a}$ specifies the set $x[\underline{a}] = \{i \geq 0 \mid a_i \models x\}$ of positions. Consequently, a formula φ of S1S can be evaluated with respect to ω-words over observations that evaluate the second-order variables in φ. For instance, the S1S formula

$$\forall i.\, (p(i) \ \rightarrow \ \exists j.\, (i \leq j \ \wedge \ q(j)))$$

specifies the ω-language corresponding to the LTL formula

$$\Box(p \ \rightarrow \ \Diamond q).$$

Let φ be an S1S formula whose free second-order variables are in X, and let $\underline{a}$ be an ω-word over the alphabet Σ_X. Then, $\underline{a} \models \varphi$ iff $\models_{\mathcal{E}} \varphi$ for every environment $\mathcal{E}$ such that for all $p \in X$ and all $i \in$, $i \in \mathcal{E}(p)$ iff $a_i \models p$. The ω-language $\mathcal{L}_\varphi$ consists of ω-words $\underline{a}$ over Σ_X such that $\underline{a} \models \varphi$.

Exercise 13.13 {T2} [S1S] Define a mapping from Ltl formulas to S1S formulas that preserves the ω-language. ■

An outstanding theorem due to Büchi establishes that the expressive power of S1S and Büchi automata coincide:

Theorem 13.5 [Büchi Theorem on S1S vs. automata] *Let X be a finite set of boolean variables, and let $\mathcal{L}$ be an ω-language over the alphabet Σ_X. Then, $\mathcal{L}$ is ω-regular iff there exists an* S1S *formula φ such that $\mathcal{L}_\varphi = \mathcal{L}$.*

Remark 13.9 [Complexity of S1S] The satisfiability and the validity problems for S1S are of nonelementary complexity. ■

First-order fragment of S1S

The logic $\mathrm{S1S}^{fo}$ consists of the fragment of S1S that disallows quantification over second-order variables. Formally, the set of formulas of $\mathrm{S1S}^{fo}$ is generated by the grammar

$$\varphi \ := \ p(e) \mid e < e \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists i.\, \varphi,$$

where i is a first-order variable, p is a second-order variable and e is a term of S1S. It turns out that the first-order fragment of S1S precisely captures the expressiveness of Ltl.

Theorem 13.6 [Expressiveness of Ltl] *For an ω-language $\mathcal{L}$, there exists an* Ltl *formula φ with $\mathcal{L}_\varphi = \mathcal{L}$ iff there exists a formula ψ of* $\mathrm{S1S}^{fo}$ *with $\mathcal{L}_\psi = \mathcal{L}$.*

Chapter 99

Prerequisite Notions and Notations

In this appendix we define some of the mathematical concepts that are used throughout the book. The appendix is not intended as a tutorial on the pertinent topics in discrete mathematics, but only as a concise reference that formalizes our use of background terminology in order to avoid ambiguities. We make no attempt at completeness: many common and (we hope) unambiguous notions, such as the standard operators of naive set theory and boolean logic, are assumed to be familiar to the reader.

Types

A *type* is a set of values together with a set of operations on these values. A type is *finite* if the corresponding set of values is finite. When no confusion arises, we use the same symbol for a type and its set of values. We distinguish between primitive types and composite types. Our primitive types are the booleans $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ (finite), the nonnegative integers $\mathbb{N} = \{0, 1, 2, \ldots\}$ (infinite), and the positive integers $\mathbb{N}^{>0} = \mathbb{N}\backslash\{0\}$ (infinite), all with the usual operations. Given a type $\mathbb{T}$, the composite type $\mathbb{T}_\perp$ has the set $\mathbb{T} \cup \{\perp\}$ of values, which includes the "undefined" value $\perp \notin \mathbb{T}$, and the operations of $\mathbb{T}$, which behave strictly on $\perp$; that is, every operation that is performed on one or more undefined arguments returns an undefined result. Other composite types are **queue of** $\mathbb{T}$ and **stack of** $\mathbb{T}$. For these two composite types, the values are the finite (possibly empty) sequences of values of type $\mathbb{T}$; thus they are infinite types. The queue and stack types differ in their operations.

The type **queue of** $\mathbb{T}$ represents "first-in-first-out" sequences, called *queues*, of values in $\mathbb{T}$ and supports five operations. In the following, let a be a value in $\mathbb{T}$, let B be a queue of values in $\mathbb{T}$, and let x be a variable of type $\mathbb{T}$. The operation *EmptyQueue* returns the empty queue. The operation $\mathit{Enqueue}(a, B)$ returns the queue that results from adding the value a to the end of the queue B. The operation $\mathit{IsEmpty}(B)$ returns *true* if the queue B is empty, and otherwise returns *false*. If B is a nonempty queue, then the operation $\mathit{Front}(B)$ returns the first element of B and the operation $\mathit{Dequeue}(B)$ returns the queue that results from removing the first element from B. The implementation of queues as linked lists supports all five operations in constant time. We use the notation **foreach** x **in** B **do** to describe a loop whose body is executed once for each element of the queue B, and during successive executions of the loop body the variable x is bound to the successive elements of B, beginning with the first element.

The type **stack of** $\mathbb{T}$ represents "last-in-first-out" sequences, called *stacks*, of values in $\mathbb{T}$ and supports six operations. In the following, let a be a value in $\mathbb{T}$ and let C be a stack of values in $\mathbb{T}$. The operation *EmptyStack* returns the empty stack. The operation $\mathit{Push}(a, C)$ returns the stack that results from adding the value a to the beginning of the stack C. The operation $\mathit{IsEmpty}(C)$ returns *true* if the stack C is empty, and otherwise returns *false*. If C is a nonempty stack, then the operation $\mathit{Top}(C)$ returns the first element of C and the operation $\mathit{Pop}(C)$ returns the stack that results from removing the first element from C. The operation $\mathit{Reverse}(C)$ returns the stack that contains the elements of C in reverse order. The implementation of stacks as linked lists supports all six operations in constant time (doubly linked lists are necessary for stack reversal).

Functions and Relations

Functions and relations can be viewed as special kinds of sets. We take this view only for relations, and consider functions as primitives.

Functions

A *function* f from a set A to a set B maps each element $a \in A$ to a unique element $f(a) \in B$. The set A is called the *domain* of f, and B is the *range* of f. We write $[A \to B]$ for the set of functions with domain A and range B. The function f is *one-to-one* if for all $a, b \in A$, if $a \neq b$, then $f(a) \neq f(b)$, and f is *onto* if for all $b \in B$, there is an element $a \in A$ such that $f(a) = b$. A *bijection* between A and B is a function that is both one-to-one and onto. The bijections between A and A are called the *permutations* on A. A *partial function* g from A to B is a function from A to $B \cup \{\bot\}$, whose range includes the undefined value $\bot \notin B$; the partial function g is *undefined* on $a \in A$ iff $g(a) = \bot$. To emphasize that a function is not partial, it may be called *total*.

The *identity function* on a set A is the function $id \in [A \to A]$ such that $id(a) = a$ for all $a \in A$. The identity function is a bijection. The *inverse function* f^{-1} of a one-to-one function f from A to B is the partial function g from B to A such that for all $b \in B$, we have $g(b) = a$ if $f(a) = b$, and $g(b) = \bot$ if $f(a) \neq b$ for all $a \in A$. The inverse function of a bijection is again a bijection. Given a function $f \in [A \to B]$ and a function $g \in [B \to C]$, the *compound function* $g \circ f$ is the function h from A to C such that $h(a) = g(f(a))$ for all $a \in A$. The composition of two one-to-one (or onto) functions is again one-to-one (or onto). A function f on a set A is extended to subsets of A and to finite and infinite sequences over A in the natural way. Let $B \subseteq A$, and let $\overline{a} = a_0 a_1 \cdots$ be a sequence of elements a_i from A. Then $f(B) = \{f(b) \mid b \in B\}$, and $f(\overline{a}) = f(a_0) f(a_1) \cdots$. A function from A^n to A is called an *n-ary function on A*. Given a unary function f on A, by f^0 we denote the identity function on A, and for all $i \in \mathbb{N}$, by f^{i+1} we denote the compound function $f^i \circ f$, which is again a unary function on A.

Given a binary function $\star$ on A, and given $a, b \in A$, we usually write $a \star b$ instead of $\star(a, b)$. The function $\star$ is *commutative* if $a \star b = b \star a$ for all $a, b \in A$, and $\star$ is *associative* if $a \star (b \star c) = (a \star b) \star c$ for all $a, b, c \in A$. The element $a \in A$ is an *identity element* with respect to $\star$ if $a \star b = b \star a = b$ for all $b \in A$. An associative binary function on A with an identity element is a *monoid*, and A is called the *carrier* of the monoid. For example, the composition $\circ$ of the unary functions on a set B is a monoid (with carrier $[B \to B]$) whose identity element is the identity function on B. If a is an identity element with respect to $\star$, and $b \star c = c \star b = a$ for $b, c \in A$, then b is an *inverse element* of c with respect to $\star$. If each carrier element of a monoid has an inverse element, then the monoid is a *group*. For example, the composition $\circ$ of the permutations on B is a group (the inverse function of a permutation is an inverse element with respect to composition). For associative binary functions $\star$ we use the following notations. In expressions such as $a \star b \star c$, we may omit parentheses. If the arguments a, b, and c are themselves large expressions, we may use a vertical arrangement

$$\begin{aligned} &\star\ a \\ &\star\ b \\ &\star\ c \end{aligned}$$

of the arguments, each preceded by the function symbol. Provided that $\star$ has an identity, if the arguments a, b, and c can be parameterized —say, $a = f(0)$, $b = f(1)$, and $c = f(2)$— we may use the function symbol as a quantifier and write $(\star\ 0 \leq i \leq 2 \mid f(i))$. If the variable that is bound by the quantifier $\star$ ranges over an empty set, then the quantified expression denotes the identity of $\star$; for example, $(+\ i \in \emptyset \mid \ldots) = 0$ and $(\circ\ i \in \emptyset \mid \ldots) = id$.

Binary relations

A (*binary*) *relation* $\sim$ between two sets A and B is a subset of $A \times B$. The set $A \times B$ itself is the *universal relation* between A and B. For $a \in A$ and $b \in B$, we usually write $a \sim b$ instead of $(a, b) \in \sim$. Let $post_\sim(a) = \{b \in B \mid a \sim b\}$. The relation $\sim$ is *serial* if $post_\sim(a)$ is nonempty for all $a \in A$,[1] and $\sim$ is *finitely branching*, if $post_\sim(a)$ is finite for all $a \in A$. In the following, we assume that $A = B$ —in this case we refer to $\sim$ as a binary relation on A. Given $B \subseteq A$, we write $\sim [B]$ for the restriction $\{(a, b) \in B^2 \mid a \sim b\}$ of $\sim$ to B. The relation $\sim$ is *reflexive* if $a \sim a$ for all $a \in A$; *irreflexive* if $a \not\sim a$ for all $a \in A$; *transitive*, if for all $a, b, c \in A$, if $a \sim b$ and $b \sim c$, then $a \sim c$; *symmetric*, if for all $a, b \in A$, if $a \sim b$, then $b \sim a$; *asymmetric*, if for all $a, b \in A$, if $a \sim b$, then $b \not\sim a$; *antisymmetric*, if for all $a, b \in A$, if $a \sim b$ and $b \sim a$, then $a = b$; and *total*, if for all $a, b \in A$, if $a \neq b$, then $a \sim b$ or $b \sim a$. A reflexive and transitive relation is a *preorder*; a symmetric preorder is an *equivalence* (*relation*); an antisymmetric preorder is a *weak partial order*; an irreflexive, asymmetric, and transitive relation is a *strict partial order*; a total (weak or strict) partial order is a (*weak* or *strict*) *linear order*. For a partial order $\sim$, a linear order that is a superset of $\sim$ is called a *linearization* of $\sim$. Every partial order has at least one linearization, and possibly several.

The *identity relation* on A, written $=$, is the smallest reflexive relation on A. The *inverse relation* $\sim^{-1}$ of the relation $\sim$ is the binary relation $\approx$ on A such that for all $a, b \in A$, we have $a \approx b$ iff $b \sim a$. Given two binary relations $\sim_1$ and $\sim_2$ on A, the *compound relation* $\sim_1 \circ \sim_2$ is the binary relation $\approx$ on A such that for all $a, b \in A$, we have $a \approx b$ iff there is an element $c \in A$ such that $a \sim_1 c$

[1] A serial binary relation $\sim$ between A and B can be thought of as a *nondeterministic function* from A to B which maps each domain element $a \in A$ to the set $post_\sim(a) \subseteq B$ of range elements.

and $c \sim_2 b$. By $\sim^0$ we denote the identity relation on A, and for all $i \in \mathbb{N}$, by $\sim^{i+1}$ we denote the compound relation $\sim^i \circ \sim$. The *reflexive closure* $\sim^{\mathit{refl}}$ of the relation $\sim$ is the smallest reflexive superset of $\sim$; that is, $\sim^{\mathit{refl}} = (\sim^0 \cup \sim^1)$. Reflexive closure is a bijection between the strict partial (or linear) orders and the weak partial (or linear) orders, and therefore, we often do not distinguish between the weak and strict varieties of orders. The *transitive closure* $\sim^+$ is the smallest transitive superset of $\sim$; that is, $\sim^+ = (\cup\ i \in \mathbb{N}^{>0} \mid \sim^i)$. The *reflexive-transitive closure* $\sim^*$ is the smallest preorder that is a superset of $\sim$; that is, $\sim^* = (\cup\ i \in \mathbb{N} \mid \sim^i)$. The *symmetric closure* $\sim^{\mathit{symm}}$ is the smallest symmetric superset of $\sim$; that is, $\sim^{\mathit{symm}} = (\sim \cup \sim^{-1})$.

Syntactic Objects

We assume a global universe of typed variables in which each variable has a unique type. This universe is not fixed, but may change from one example to the next. For instance, in one example, the variable x may have the type $\mathbb{N}$, and in another example, x may have the type $\mathbb{B}$. However, we never combine or relate two syntactic objects (such as two reactive modules) from two different universes. In every universe we assume that each variable x has a primed twin x' of the same type. If X is a set of variables, we denote by $X' = \{x' \mid x \in X\}$ the set of all primed variables whose unprimed twins are contained in X.

Expressions and valuations

Let X be a finite set of typed variables. An *expression over* X is a typed expression e whose free variables are from X. We write $\mathit{free}(e)$ for the set of variables that occur freely in the expression e; then $\mathit{free}(e) \subseteq X$. If the variable x and the expression d are type-compatible, we write $e[x := d]$ for the expression over $X \cup \mathit{free}(d)$ which results from safely substituting d for all free occurrences of x in e.[2] A *valuation* for X is a function s that maps each variable $x \in X$ to a value $s(x)$ of the appropriate type. By Σ_X we denote the set of valuations for X.[3] The function $s \in \Sigma_X$ is extended to expressions over X in the standard way. If y is a variable that may or may not be contained in X, and a is a value in the type of y, then $s[y \mapsto a]$ is the valuation in $\Sigma_{X \cup \{y\}}$ which maps y to a, and maps each variable $x \in X$ different from y to the value $s(x)$. For a valuation $s \in \Sigma_X$ and a set $Y \subseteq X$ of variables, the valuation $s[Y] \in \Sigma_Y$ is the restriction of s to the domain of variables in Y. For two disjoint sets X and Y of variables, and two valuations $s \in \Sigma_X$ and $t \in \Sigma_Y$, the valuation $(s \cup t) \in \Sigma_{X \cup Y}$ maps

[2] Expressions may contain quantifiers. Safe substitution requires that the bound variables of e that occur freely in d are suitably renamed before the free occurrences of x are replaced with d. For example, if e is the boolean expression $(\exists y \mid y = x + 1)$ and d is the integer expression $2y$, then $e[x := d]$ denotes, up to renaming of the bound variable y', the boolean expression $(\exists y' \mid y' = 2y + 1)$.

[3] There is precisely one valuation for the empty set of variables.

each variable $x \in X$ to the value $s(x)$, and maps each variable $y \in Y$ to the value $t(y)$.

Let p be a boolean expression over X, and let s be a valuation for X. The valuation s *satisfies* the expression p, written $s \models p$, if $s(p) = \mathit{true}$; otherwise s *violates* p. If s satisfies p, then s is called a *model* of p. We write $[\![p]\!]$ for the set of models of p; that is, $[\![p]\!] = \{s \in \Sigma_X \mid s \models p\}$. The boolean expression p is *satisfiable* if there is a valuation for X that satisfies p; that is, $[\![p]\!] \neq \emptyset$. The expression p is *valid*, written $\models p$, if all valuations for X satisfy p; that is, $[\![p]\!] = \Sigma_X$. Let q be a second boolean expression over X. The boolean expression p *implies* the boolean expression q if every model of p is a model of q; that is, $[\![p]\!] \subseteq [\![q]\!]$. The two expressions p and q are *equivalent* if they have the same models; that is, $[\![p]\!] = [\![q]\!]$.

Guarded commands

Let X and Y be two finite sets of typed variables. A *guarded assignment* γ *from* X *to* Y consists of a *guard* p_γ and for each variable $y \in Y$, an *assignment* e^y_γ. The guard p_γ is a boolean expression over X. Each assignment e^y_γ is an expression over X that is type-compatible with y. Informally, the guarded assignment γ can be executed if the guard p_γ evaluates to true, and then each variable $y \in Y$ is updated to the value of the assignment e^y_γ. Formally, the guarded assignment γ defines a partial function $[\![\gamma]\!]$ from the valuations for X to the valuations for Y: given $s \in \Sigma_X$, if $s \models p_\gamma$, then $[\![\gamma]\!](s)$ maps each variable $y \in Y$ to the value $s(e^y_\gamma)$; otherwise $[\![\gamma]\!](s)$ is undefined. When writing guarded assignments, we may suppress assignments that leave the value of a variable unchanged: we specify the guarded assignment γ using the notation

$$p_\gamma \;\rightarrow\; y_1 := e^{y_1}_\gamma;\; \ldots;\; y_m := e^{y_m}_\gamma,$$

where $y_1, \ldots, y_m$ are pairwise distinct variables from Y (possibly $m = 0$) such that $e^y_\gamma = y$ for all variables $y \in Y$ that do not appear in the list $y_1, \ldots, y_m$.

A *guarded command* Γ *from* X *to* Y is a finite set $\{\gamma_i \mid 1 \leq i \leq n\}$ of guarded assignments from X to Y such that the disjunction $(\vee\, 1 \leq i \leq n \mid p_i)$ of the guards is valid[4] (this implies that $n > 0$). Informally, a guarded command nondeterministically chooses one of the guards that evaluates to true, and then executes the corresponding guarded assignment. Formally, the guarded command Γ defines a serial binary relation $[\![\Gamma]\!]$ between Σ_X and Σ_Y, namely, $(s, t) \in [\![\Gamma]\!]$ iff $[\![\gamma_i]\!](s) = t$ for some $1 \leq i \leq n$. The guarded assignment $\gamma_n \in \Gamma$ is called a *default assignment* if (1) $p_n = (\wedge\, 1 \leq i < n \mid \neg p_i)$ and (2) $e^y_n = y$ for all variables $y \in Y$; that is, if none of the other guards evaluates to true, then the values of all variables stay unchanged. When writing guarded commands, we

[4] Given a guarded assignment $\gamma_i \in \Gamma$, we write p_i short for the guard p_{γ_i}, and similarly for the assignments of γ_i.

may suppress default assignments: we specify the guarded command Γ using the notation

$$\gamma_1 \;[\!]\; \cdots \;[\!]\; \gamma_{n-1}$$

if γ_n is a default assignment; otherwise we write $\gamma_1 \,[\!]\, \cdots \,[\!]\, \gamma_n$. The guarded command Γ is *deterministic* if for all guarded assignments $\gamma_i, \gamma_j \in \Gamma$ and all valuations $s \in \Sigma_X$, if both $s \models p_i$ and $s \models p_j$, then $[\![\gamma_i]\!](s) = [\![\gamma_j]\!](s)$. If the guarded command Γ is deterministic, then $[\![\Gamma]\!]$ is a (total) function from Σ_X to Σ_Y. For finite sets $\mathbb{T} = \{a_1, \ldots, a_n\}$, we freely use abbreviations such as $p \to y := \mathbb{T}$ for the nondeterministic guarded command

$$p \to y := a_1 \;[\!]\; \cdots \;[\!]\; p \to y := a_n.$$

Variable renamings

Let X be a finite set of typed variables. A *renaming* ρ for X is a one-to-one function that maps each variable $x \in X$ to a type-compatible variable $x[\rho]$. Given a set $Y \subseteq X$ of variables, we write $Y[\rho]$ for the set $\{y[\rho] \mid y \in Y\}$ of renamed variables. Given an expression e over X, we write $e[\rho]$ for the expression over $X[\rho]$ that results from e by replacing all free occurrences of each variable $x \in X$ with the variable $x[\rho]$ using safe substitution. Renaming extends to guarded assignments and guarded commands in the natural way, by applying the renaming to all subexpressions: given a guarded command Γ from X to Y, and a renaming ρ for $X \cup Y$, we write $\Gamma[\rho]$ for the renamed guarded command from $X[\rho]$ to $Y[\rho]$. We specify the renaming ρ using the notation

$$x_1, \ldots, x_m \;:=\; x_1[\rho], \ldots, x_m[\rho],$$

where $x_1, \ldots, x_m$ are pairwise distinct variables from X (possibly $m = 0$) such that $x[\rho] = x$ for all variables $x \in X$ that do not appear in the list $x_1, \ldots, x_m$.

Words and Languages

Let A be a nonempty set of letters. A *word* $\overline{a} = a_1 \cdots a_m$ *over the alphabet* A is a finite sequence of letters a_i from A. We write $|\overline{a}| = m$ for the *length* of $\overline{a}$; that is, $|\overline{a}|$ denotes the number of letters in $\overline{a}$. By ϵ we denote the *empty word* (then $|\epsilon| = 0$). By $\overline{a} \cdot \overline{b}$ we denote the word that results from *concatenating* the two words $\overline{a}$ and $\overline{b}$ (then $|\overline{a} \cdot \overline{b}| = |\overline{a}| + |\overline{b}|$). By $\overline{a}_{i..j}$, for $1 \le i \le j \le m$, we denote the word $a_i \cdots a_j$ that results from $\overline{a}$ by removing $i - 1$ initial and $m - j$ final letters (then $|\overline{a}_{i..j}| = j - i + 1$). We write A^* for the set of words over A, and A^+ for the set of nonempty words. A *language* L *over the alphabet* A is a set of nonempty words over A; that is, $L \subseteq A^+$. The word $\overline{a}$ is a *prefix* of the word $\overline{b}$ if there exists a word $\overline{c}$ such that $\overline{b} = \overline{a} \cdot \overline{c}$; and $\overline{a}$ is a *suffix* of $\overline{b}$ if there exists a word $\overline{c}$ such that $\overline{b} = \overline{c} \cdot \overline{a}$. The language L is *prefix-closed* if for every word $\overline{a}$ in L, all prefixes of $\overline{a}$ are also in L; and L is *suffix-closed* if for every word $\overline{a}$

in L, all suffixes of $\overline{a}$ are also in L. The language L is *fusion-closed* if for all letters a, if $\overline{b} \cdot a \cdot \overline{c}$ and $\overline{b}' \cdot a \cdot \overline{c}'$ are in L, then so is $\overline{b} \cdot a \cdot \overline{c}'$. The language L is *upward stutter-closed* if for all letters a, if $\overline{b} \cdot a \cdot \overline{c}$ is in L, then so is $\overline{b} \cdot a \cdot a \cdot \overline{c}$. The language L is *downward stutter-closed* if for all letters a, if $\overline{b} \cdot a \cdot a \cdot \overline{c}$ is in L, then so is $\overline{b} \cdot a \cdot a \cdot \overline{c}$. The language L is *stutter-closed* if L is both upward stutter-closed and downward stutter-closed.

Index