Chapter 1

Response

In the preceding chapters we discussed methods for proving safety properties. While the class of safety properties is very important, and normally occupies a large portion of a specification, it must be complemented by properties in the other classes.

It is interesting to note that the expression of safety properties and their verification use relatively little temporal logic. The main emphasis there is directed towards finding assertions that are invariant over the computation of a program. Most of the verification efforts are concentrated on showing that the assertions are inductive. This requires proving a set of verification conditions, which are expressed by nontemporal state formulas. Temporal logic is used mainly for stating the final result of invariance of the assertion. It is true that, when considering precedence properties, we extensively use the past part of temporal logic. But as we commented there, an equivalent, though sometimes less elegant, state formulation of these properties can be managed through auxiliary or history variables.

It is only when we enter the realm of more general properties, that temporal logic becomes an essential and irreplaceable tool. Thus if, for some reason, one is willing to restrict himself to the study of safety properties of reactive programs, he does not need the full power of temporal logic.

A related observation is that, only when we go beyond safety properties does fairness become meaningful. Recall the definition of a *run* as a state sequence that satisfies the requirements of initiation and consecution but not necessarily any of the fairness requirements. It can be shown that a safety formula holds over all runs of a program if and only if it holds over all computations, i.e., fair runs. Thus, safety properties cannot distinguish between fair and unfair runs.

This is no longer the case with progress (nonsafety) properties. Note, for example, that the infinite sequence $s_0, s_0, \ldots$ is a legal run of any program P, provided $s_0 \models \Theta$. This run is generated by continuously taking the idling transition τ_I. There are very few progress properties that hold over this run.

Consequently, while safety properties do not depend on the fairness requirements for their validity, progress properties do. In Chapters 1–3 we concentrate on the important class of *response* properties, which is one of the progress classes. This class contains properties that can be expressed by a formula of the form

$$\Box \Diamond p,$$

for a past formula p. In these chapters, we introduce a family of rules for response properties that rely on the different fairness requirements. Chapters 1–2 present rules that rely on *justice*, while Chapter 3 presents rules that rely on (justice and) *compassion*.

Chapter 4 completes the picture by presenting rules for the highest progress class, that of reactivity.

Chapter 1 deals with response properties that rely on the just transitions of the system for their validity. Chapter 3 generalizes the treatment to properties that rely on both justice and compassion for their validity.

In Section 1.1 we consider a single-step rule that relies on the activation of a single just transition.

Section 1.2 shows how to combine several applications of the single-step rule into a rule that relies on a fixed number of activations of just transitions.

Section 1.3 generalizes the rule to the case that the number of just activations necessary to achieve the goal is not fixed and may depend, for example, on an input parameter.

In Section 1.4, we extend all the above methods to prove properties expressed by response formulas that contain past subformulas.

Section 1.5 deals with the class of guarantee formulas, treating them as a special case of response properties.

In a similar way, Section 1.6 considers the class of obligation properties. Again, their verification is based on their consideration as a special case of the response class.

1.1 Response Rule

Even though there are several different classes of progress properties, their verification is almost always based on the establishment of a single construct — the *response formula*

$$p \Rightarrow \Diamond q,$$

for past formulas p and q. This formula states that any position in the computation which satisfies p must be followed by a later position which satisfies q. Since

the canonical response formula $\Box \Diamond q$ is equivalent to $\mathrm{T} \Rightarrow \Diamond q$, it follows that every response property can be expressed by a formula of the form $p \Rightarrow \Diamond q$.

A general response property allows p and q to be general past formulas. In Sections 1.1–1.3 we consider the simpler case that p and q are assertions. In Section 1.4 we will generalize the treatment to the case that p and q are past formulas.

A Single-Step Rule

A single-step rule, relying on justice, is provided by rule RESP-J presented in Fig. 1.1.

For assertions p, q, φ, and transition $\tau_h \in \mathcal{J}$,

J1. $p \rightarrow q \vee \varphi$

J2. $\{\varphi\}\ \mathcal{T}\ \{q \vee \varphi\}$

J3. $\{\varphi\}\ \tau_h\ \{q\}$

J4. $\varphi \rightarrow En(\tau_h)$

$p \Rightarrow \Diamond q$

Fig. 1.1. Rule RESP-J (single-step response under justice).

The rule calls for the identification of an intermediate assertion φ and a just transition $\tau_h \in \mathcal{J}$, to which we refer as the *helpful transition*.

Premise J1 of the rule states that, in any position satisfying p, either the goal formula q already holds, or the intermediate formula φ, bridging the passage from p to q, holds. The q-disjunct of this premise covers the case that the distance between the p-position and the q-position is 0. The φ-disjunct and the other premises cover the case that the distance between these two positions is positive.

Premise J2 requires that every transition leads from a φ-position to a position that satisfies $q \vee \varphi$. That is, either a position satisfying the goal formula q is attained or, if not, then at least the intermediate φ is maintained.

Premise J3 requires that the helpful transition τ_h always leads from a φ-position to a q-position.

Premise J4 requires that the helpful transition τ_h is enabled at every φ-position.

Justification To justify the rule, consider a computation σ which satisfies the four premises of the rule. Assume that p holds at position k, $k \geq 0$. We wish to show that q holds at some position i, $i \geq k$. Assume, to the contrary, that it does not. That means that for all i, $i \geq k$, q does not hold at i. By J1, φ holds at position k. By J2, every successor of a φ-state is a $(q \vee \varphi)$-state.

Since we assumed that q never occurs beyond k, it follows that φ holds continuously beyond k. By J4, the just transition τ_h must be continuously enabled. However, τ_h is never taken beyond k. This is because if τ_h were taken, it would have been taken from a φ-position, and by J3 the next position would have satisfied q. Thus we have that τ_h is continuously enabled, but never taken beyond k. It follows that the sequence σ is not just with respect to τ_h, and is, therefore, not a computation.

This shows that, for all computations, there must exist an i, $i \geq k$ such that q holds at i. ∎

In applications of the rule, it is sufficient to establish premise J2 for all $\tau \neq \tau_h$, since J2 for $\tau = \tau_h$ is implied by J3. It is also unnecessary to check premise J2 for $\tau = \tau_I$, the idling transition, since $\{\varphi\}\ \tau_I\ \{\varphi\}$ is trivially state valid.

Example (program ANY-Y)

Program ANY-Y of Fig. 1.2 illustrates a simple program consisting of two processes communicating by the shared variable x, initially set to 0. Process P_1 keeps incrementing variable y as long as $x = 0$. Process P_2 has only one statement, which sets x to 1. Obviously, once x is set to 1, process P_2 terminates, and some time later so does P_1, as soon as it observes that $x \neq 0$.

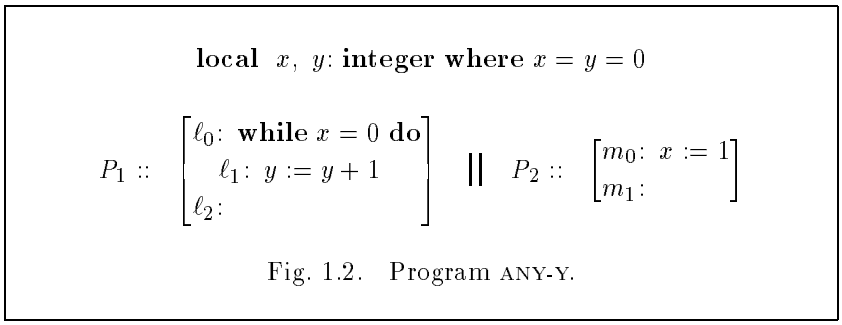


Fig. 1.2. Program ANY-Y.

We illustrate the use of rule RESP-J for proving the response property

$$at\_m_0 \ \Rightarrow\ \diamondsuit(x = 1)$$

for program ANY-Y.

As the helpful transition τ_h we take m_0. As the intermediate assertion φ we take p: at_m_0. Premise J1 assumes the form

$$\underbrace{at\_m_0}_{p} \rightarrow \underbrace{\cdots}_{q} \vee \underbrace{at\_m_0}_{\varphi},$$

which is obviously valid. Premise J2 is established by showing that all transitions, excluding m_0, preserve φ: at_m_0, which is clearly the case.

Premise J3 requires showing that m_0 leads from any φ-state to a q-state, expressed by

$$\underbrace{\cdots \wedge x' = 1}_{\rho_{m_0}} \wedge \underbrace{\cdots}_{\varphi} \rightarrow \underbrace{x' = 1}_{q'},$$

which is obviously valid. Finally, J4 requires

$$\underbrace{at\_m_0}_{\varphi} \rightarrow \underbrace{at\_m_0}_{En(m_0)},$$

which is also valid. This establishes that the property specified by the response formula $at\_m_0 \Rightarrow \diamondsuit(x = 1)$ is valid over program ANY-Y. ∎

Combining Response Properties

Rule RESP-J by itself is not a very strong rule, and is sufficient only for proving *one-step* response properties, i.e., properties that can be achieved by a single activation of a helpful transition. For example, while program ANY-Y always terminates, its termination cannot be proven by a single aplication of rule RESP-J.

In general, most response properties of the form $p \Rightarrow \diamondsuit q$ require several helpful steps in order to get from a p-position to a q-position.

To establish such properties we may use several rules that enable us to combine response properties, each established by a single application of rule RESP-J. These rules are based on general properties of response formulas that allow us to form these combinations. We list some of these properties as proof rules. All of these rules can be established as derived rules, using the standard deductive system for temporal logic[1] .

Monotonicity

An important property of response formulas is the monotonicity of both the antecedent and the consequent. This can be summarized in the form of the (monotonicity) rule MON-R, presented in Fig. 1.3.

[1] For example, the one presented in Chapter 3 of Volume I.

$$\frac{p \Rightarrow q \qquad q \Rightarrow \Diamond r \qquad r \Rightarrow t}{p \Rightarrow \Diamond t}$$

Fig. 1.3. Rule MON-R (monotonicity of response)

Rule MON-R enables us to strengthen the antecedent and weaken the consequent. Thus, if we managed to prove the response formula

$$at\_\ell_0 \Rightarrow \Diamond(x = 1),$$

we can infer from it, using rule MON-R, the formula

$$at\_\ell_0 \Rightarrow \Diamond(x > 0).$$

Reflexivity

Property RFLX-R of Fig. 1.4 states that the $\Diamond$ operator is reflexive.

$$p \Rightarrow \Diamond p$$

Fig. 1.4. Property RFLX-R (reflexivity of response)

We may use this property to prove simple response formulas such as

$$x = 0 \Rightarrow \Diamond(x = 0).$$

Transitivity

The transitivity property of response formulas is expressed by the (transitivity) rule TRNS-R, presented in Fig. 1.5.

$$\frac{p \Rightarrow \Diamond q \qquad q \Rightarrow \Diamond r}{p \Rightarrow \Diamond r}$$

Fig. 1.5. Rule TRNS-R (transitivity of response)

Thus, if we managed to prove for program ANY-Y the two response formulas

$$at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ \Diamond(at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1)$$
$$at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ \Diamond(at\_\ell_2 \ \wedge \ at\_m_1),$$

we may use rule TRNS-R to conclude

$$at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ \Diamond(at\_\ell_2 \ \wedge \ at\_m_1).$$

The soundness of rule TRNS-R is obvious. Consider a computation σ such that the first two premises are valid over σ. Let i be a position satisfying p. By the first premise, there exists a position j, $j \geq i$, satisfying q. By the second premise, there exists a position k, $k \geq j$, satisfying r. Thus, we are ensured of a position k, $k \geq i$, satisfying r, which establishes $p \Rightarrow \Diamond r$.

Proof by Cases

Another useful property of response formulas is that it is amenable to proof by cases. This possibility is presented by rule CASES-R of Fig. 1.6.

$$\frac{p \ \Rightarrow \ \Diamond r \qquad q \ \Rightarrow \ \Diamond r}{(p \ \vee \ q) \ \Rightarrow \ \Diamond r}$$

Fig. 1.6. Rule CASES-R (case analysis for response)

Assume, for example, that we have proved for program ANY-Y the two following reponse formulas.

$$at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ \Diamond(at\_\ell_2 \ \wedge \ at\_m_1)$$
$$at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ \Diamond(at\_\ell_2 \ \wedge \ at\_m_1).$$

Then, we may use rule CASES-R to conclude

$$(at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1) \ \vee \ (at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1) \ \Rightarrow \ \Diamond(at\_\ell_2 \ \wedge \ at\_m_1),$$

from which, by rule MON-R, we can infer

$$at\_\ell_{0,1} \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \Rightarrow \ \Diamond(at\_\ell_2 \ \wedge \ at\_m_1).$$

Example (program ANY-Y)

We will illustrate the use of these rules by proving termination of program ANY-Y. This property can be expressed by the response formula

$$\Theta \;\Rightarrow\; \diamondsuit(at\_\ell_2 \;\wedge\; at\_m_1),$$

where

$$\Theta\colon\quad \pi = \{\ell_0, m_0\} \;\wedge\; x = 0 \;\wedge\; y = 0$$

is the initial condition of program ANY-Y.

The proof consists of the following steps:

1. $at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = 0 \;\Rightarrow\; \diamondsuit(at\_\ell_{0,1} \;\wedge\; at\_m_1 \;\wedge\; x = 1)$
   by rule RESP-J, taking τ_h: m_0 and φ: $at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0$

2. $at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; \diamondsuit(at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule RESP-J, taking τ_h: ℓ_0 and φ: $at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1$

3. $at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; \diamondsuit(at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1)$
   by rule RESP-J, taking τ_h: ℓ_1 and φ: $at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1$

4. $at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; \diamondsuit(at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule TRNS-R, applied to 3 and 2

5. $(at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1) \;\vee\; (at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1) \;\Rightarrow\; \diamondsuit(at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule CASES-R, applied to 2 and 4

6. $at\_\ell_{0,1} \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\rightarrow\; (at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1) \;\vee\; (at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1)$
   an assertional validity

7. $at\_\ell_{0,1} \;\wedge\; at\_m_1 \;\wedge\; x = 1 \;\Rightarrow\; \diamondsuit(at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule MON-R, using 5 and 6

8. $at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = 0 \;\Rightarrow\; \diamondsuit(at\_\ell_2 \;\wedge\; at\_m_1)$
   by rule TRNS-R, applied to 1 and 7

9. $\Theta \;\rightarrow\; at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = 0$ an assertional validity

10. $\Theta \;\Rightarrow\; \diamondsuit(at\_\ell_2 \;\wedge\; at\_m_1)$ by rule MON-R, applied to 8 and 9. ∎

1.2 Chain Rule

The proof of the last example follows a very specific pattern that occurs often in proofs of response properties. According to this pattern, to establish $p \Rightarrow \diamondsuit q$, we identify a sequence of intermediate situations described by assertions $\varphi_m, \varphi_{m-1}, \ldots, \varphi_0$ such that p implies one of $\varphi_m, \ldots, \varphi_0$, and q is identical with

φ_0 (or is implied by φ_0). We then show that for every $i > 0$, being at φ_i implies that eventually we must reach φ_j for some $j < i$.

We can interpret the index i of the intermediate formula φ_i as a measure of the distance of the current state from a state that satisfies the goal q. Thus, the lower the index, the closer we are to achieving the goal q. For a position j, let φ_i be the intermediate formula with the smallest i s.t. φ_i holds at j. We refer to the index i as the *rank* of position j.

This proof pattern is summarized in rule CHAIN-J (Fig. 1.7).

For assertions p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$ and transitions $\tau_1, \ldots, \tau_m \in \mathcal{J}$

$$\text{J1.} \quad p \rightarrow \bigvee_{j=0}^{m} \varphi_j$$

$$\left.\begin{array}{ll} \text{J2.} & \{\varphi_i\}\ \mathcal{T}\ \left\{\bigvee_{j\le i} \varphi_j\right\} \\ \text{J3.} & \{\varphi_i\}\ \tau_i\ \left\{\bigvee_{j< i} \varphi_j\right\} \\ \text{J4.} & \varphi_i \rightarrow En(\tau_i) \end{array}\right\} \text{ for } i = 1, \ldots, m$$

$$p \Rightarrow \Diamond q$$

Fig. 1.7. Rule CHAIN-J (chain rule under justice).

According to premise J1, p implies that one of the intermediate formulas φ_i (possibly φ_0 implying q) holds. Premise J2 requires that taking any transition from a φ_i-position results in a next position which satisfies φ_j, for some $j \le i$. Premise J3 requires that taking the helpful transition τ_i from a φ_i-position results in a next position which satisfies φ_j for $j < i$. We can view premise J2 as stating that the rank never increases, while premise J3 states that the helpful transition guarantees that the rank decreases. Premise J4 claims that the helpful transition τ_i is enabled at every φ_i-position.

Justification Assume that all four premises are P-state valid. Consider a P-computation σ and a position t that satisfies p. We wish to prove that some

later position satisfies q. Assume to the contrary that all positions later than t (including t itself) do not satisfy q. By J1, position t satisfies φ_j for some $j \geq 0$. Index j cannot be 0 because $\varphi_0 = q$ and we assumed that no position beyond t satisfies q. Thus, position t satisfies φ_j, for some $j > 0$. By J2, position $t+1$ satisfies some φ_k, $k \leq j$. Again, $k > 0$ due to $\varphi_0 = q$. Continuing in this manner, it follows that every position beyond t satisfies some φ_j for $j > 0$, to which we refer as the rank of the position.

By J2, the rank of the position can either decrease or remain the same. It follows that there must exist some position $k \geq t$, beyond which the rank never decreases.

Assume that i is the rank of the state at position k. Since q is never satisfied and the rank never decreases beyond position k, it follows (by J2) that φ_i holds continually beyond k. By J3, τ_i cannot be taken beyond k, because that would have led to a rank decrease. By J4, τ_i is continually enabled beyond k yet, by the argument above, it is never taken. This violates the requirement of justice for τ_i.

It follows that if all the premises of the rule are P-state valid then the conclusion $p \Rightarrow \diamondsuit q$ is P-valid. ∎

Note that since premise J3 implies premise J2 for $\tau = \tau_i$, it is sufficient to check premise J2 for a given $i = 1, \ldots, m$, only for $\tau \neq \tau_i$. Also, it is unnecessary to check premise J2 for $\tau = \tau_I$, since $\{\varphi_i\}\, \tau_I\, \{\varphi_i\}$ trivially holds.

Example (Reproving termination of program ANY-Y)

Let us show how termination of program ANY-Y can be proved (again) by a single application of rule CHAIN-J.

The property we wish to prove is

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = 0 \ \wedge \ y = 0}_{p = \Theta} \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_2 \ \wedge \ at\_m_1}_{q} \ .$$

Inspired by our previous proof of this property, we choose four assertions and corresponding helpful transitions as follows:

$$\begin{array}{ll} \varphi_3\colon \quad at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0 & \tau_3\colon \ m_0 \\ \varphi_2\colon \quad at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 & \tau_2\colon \ \ell_1 \\ \varphi_1\colon \quad at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 & \tau_1\colon \ \ell_0 \\ \varphi_0 = q\colon \quad at\_\ell_2 \ \wedge \ at\_m_1 \, . & \end{array}$$

Let us consider each of the premises of rule CHAIN-J.

- Premise J1

  This premise calls for proving

$$p \;\rightarrow\; \bigvee_{j=0}^{3} \varphi_j.$$

We will prove $p \rightarrow \varphi_3$, which amounts to

$$\underbrace{at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; x = 0 \;\wedge\; y = 0}_{p} \;\rightarrow\; \underbrace{at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0}_{\varphi_3}$$

which obviously holds.

- Premises J2–J4 for $i = 3, 2, 1$

  We list below the premises that are proven for each i, $i = 3, 2, 1$.

  - Assertion φ_3: $at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0$

$$\left\{\underbrace{at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0}_{\varphi_3}\right\} \;\tau\; \left\{\underbrace{at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0}_{\varphi_3}\right\}$$

for each $\tau \neq m_0$

$$\left\{\underbrace{at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0}_{\varphi_3}\right\} \;m_0\; \left\{\begin{array}{l} \underbrace{at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_2} \;\vee \\ \underbrace{at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_1} \end{array}\right\}$$

$$\underbrace{at\_\ell_{0,1} \;\wedge\; at\_m_0 \;\wedge\; x = 0}_{\varphi_3} \;\rightarrow\; \underbrace{at\_m_0}_{En(m_0)}\;.$$

  - Assertion φ_2: $at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1$

$$\left\{\underbrace{at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_2}\right\} \;\tau\; \left\{\underbrace{at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_2}\right\}$$

for every $\tau \neq \ell_1$

$$\left\{\underbrace{at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_2}\right\} \;\ell_1\; \left\{\underbrace{at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_1}\right\}$$

$$\left[\underbrace{at\_\ell_1 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_2}\right] \;\rightarrow\; \underbrace{at\_\ell_1}_{En(\ell_1)}\;.$$

  - Assertion φ_1: $at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1$

$$\left\{\underbrace{at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_1}\right\} \;\tau\; \left\{\underbrace{at\_\ell_0 \;\wedge\; at\_m_1 \;\wedge\; x = 1}_{\varphi_1}\right\}$$

for every $\tau \neq \ell_0$

$$\left\{\underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x=1}_{\varphi_1}\right\} \ \ell_0 \ \left\{\underbrace{at\_\ell_2 \ \wedge \ at\_m_1}_{\varphi_0}\right\}$$

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x=1}_{\varphi_1} \ \rightarrow \ \underbrace{at\_\ell_0}_{En(\ell_0)} \ .$$

All of these implications and verification conditions are obviously state valid, which establishes the conclusion

$$\Theta \ \Rightarrow \ \diamondsuit(at\_\ell_2 \ \wedge \ at\_m_1). \ \blacksquare$$

Relation to rule NWAIT

There is a strong resemblance between the premises of rule CHAIN-J and those of rule NWAIT (Fig. 3.6 on page 267 of the SAFETY book). This is not surprising, since in both cases we wish to establish the evolution from p to q (q_0 in the case of NWAIT) by successively passing through $\varphi_m, \varphi_{m-1}, \ldots, \varphi_1, \varphi_0$.

The main difference is that, in rule NWAIT, we are quite satisfied if, from a certain point on, we stay forever within φ_j for some $j > 0$. This is unacceptable in a response rule, where we are anxious to establish eventual arrival at φ_0. This difference is expressed in premise J3 which requires that activation of the helpful transition τ_i takes us out of a φ_i-state, and premise J4 which requires that τ_i is enabled on all φ_i-states. These premises have no counterparts in rule NWAIT. This excludes computations that consist of states whose rank, from a certain point on, never decreases below some $j > 0$.

Another difference is that, in rule NWAIT, we allow a transition to lead from φ_i, $i > 0$, back to φ_i. This is expressed by the disjunction, $\bigvee_{j \leq i} \varphi_j$, appearing in the postcondition of premise N3. Rule CHAIN-J allows this in premise J2 for all but the helpful transition, which is required in J3 to lead to a position with a strictly lower rank.

In spite of these differences, many of the approaches used in the study of precedence properties are also applicable to the analysis of response properties. One of these useful approaches is that of *verification diagrams* introduced in Section 3.3 of the SAFETY book.

1.3 Chain Diagrams

As observed in the previous example (program ANY-Y), in many cases it suffices to specify the intermediate assertions $\varphi_0, \varphi_1, \ldots, \varphi_m$ and to identify the helpful transitions $\tau_1, \ldots, \tau_m$. The proofs of the actual verification conditions is a detail

that can be left to skeptical readers, and eventually to an automated system. Only some of the verification conditions raise interesting questions, and those are usually elaborated in a presentation of a proof. However, the main structure of the proof is adequately represented by the list of assertions and their corresponding helpful transitions.

A concise visual summary of this information, and some additional details, is provided by verification diagrams. Verification diagrams were already introduced in Section 3.3 of the SAFETY book to represent proofs using rule NWAIT. However, we give here an independent description of the diagrams that support proofs of response properties by rule CHAIN-J.

Verification Diagrams

A verification diagram is a directed labeled graph constructed as follows:

- *Nodes* in the graph are labeled by assertions. We will often refer to a node by the assertion labeling it.
- *Edges* in the graph represent transitions between assertions. The diagrams presenting proofs by rule CHAIN-J allow edges of two types, represented graphically by *single* (*-lined*) and *double* (*-lined*) arrows. Each edge of either type departs from one assertion, connects to another, and is labeled by the name of a transition. We refer to an edge labeled by τ as a *τ-edge*.
- One of the nodes may be designated as a *terminal node* ("goal" node). In the graphical representation, this node is distinguished by having a boldface boundary. No edges depart from a terminal node. Terminal nodes correspond to "goal" assertions such as φ_0 in rule CHAIN-J.

Chain Diagrams

A verification diagram is said to be a *chain diagram* if its nodes are labeled by assertions $\varphi_0, \ldots, \varphi_m$, with φ_0 being the terminal node, and if it satisfies the following requirements:

- If a single (-line) edge connects node φ_i to node φ_j, then $i \geq j$.
- If a double (-line) edge connects node φ_i to node φ_j, then $i > j$.
- Every node φ_i, $i > 0$, has a double edge departing from it. This identifies the transition labeling such an edge as *helpful* for assertion φ_i. All helpful transitions must be just.

The first two requirements ensure that the diagram is weakly acyclic in the sense defined in Section 3.3 of the SAFETY book for WAIT diagrams. That is, the terminal node is labeled by φ_0 and whenever node φ_i is connected by an edge (single or double) to node φ_j, then $j \leq i$. The stronger second requirement ensures that the subgraph based on the double edges is acyclic, forbidding self-connections by

double edges. The third requirement demands that every nonterminal assertion (i.e., φ_i for $i > 0$) has at least one helpful transition associated with it.

Verification Conditions for CHAIN Diagrams

The assertions labeling nodes in a diagram are intended to represent the intermediate assertions appearing in a CHAIN-J proof. A τ-labeled edge connecting node φ_i to node φ_j implies that it is possible for a φ_i-state to have a τ-successor satisfying φ_j. A double edge departing from node φ and labeled by transition τ identifies τ as helpful for assertion φ. Consequently, we associate verification conditions with nodes and the edges departing from them. These conditions, expressed by implications, represent premises J2–J4 of rule CHAIN-J.

For a node φ_i and transition τ, connecting φ_i to φ_j, we say that φ_j is a τ-successor of φ_i. Let φ be a nonterminal node and $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the τ-successors of φ.

V1. If all the edges connecting φ to its τ-successors are single (-lined), then we associate with φ and τ the verification condition

$$\{\varphi\}\ \tau\ \{\varphi\ \vee\ \varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}.$$

Transition τ, labeling only single edges, is identified as unhelpful for φ. This condition, similar to premise J2, allows τ to lead from a φ-state back to a φ-state, recording no progress.

The case of a transition τ that does not label any edges departing from φ is interpreted as though τ labels $k = 0$ single-lined edges departing from φ. That is, with such a transition we associate the verification condition

$$\{\varphi\}\ \tau\ \{\varphi\}.$$

V2. If some edge departing from φ is double (hence $k > 0$), we associate with φ and τ the verification condition

$$\{\varphi\}\ \tau\ \{\varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}.$$

This condition corresponds to premise J3, requiring a transition τ, identified as helpful, to lead away from φ.

V3. If τ labels some double edge departing from φ, we require

$$\varphi\ \rightarrow\ \mathit{En}(\tau).$$

This condition corresponds to premise J4, requiring that a transition helpful for φ is enabled on all φ-states. We refer to this requirement as the *enabling requirement*.

Valid CHAIN Diagrams

A CHAIN diagram is said to be *valid over a program* P (*P-valid* for short) if all the verification conditions associated with nodes of the diagram are P-state valid.

The consequences of having a P-valid CHAIN diagram are stated by the following claim:

Claim 1.1 (CHAIN diagrams)

A P-valid CHAIN diagram establishes that the response formula

$$\bigvee_{j=0}^{m} \varphi_j \;\Rightarrow\; \Diamond\, \varphi_0$$

is P-valid.

If, in addition, we can establish the P-state validity of the following implications:

$$\text{(J1)}\quad p \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \text{(J0)}\quad \varphi_0 \;\rightarrow\; q$$

then, we can conclude the P-validity of

$$p \;\Rightarrow\; \Diamond\, q.$$

Justification First, we show the first part of the claim, stating the P-validity of

$$\bigvee_{j=0}^{m} \varphi_j \;\Rightarrow\; \Diamond\, \varphi_0.$$

We use rule CHAIN-J with p: $\bigvee_{j=0}^{m} \varphi_j$, $q = \varphi_0$ and, for each $i = 1, \ldots, m$, we take τ_i (the helpful transition for φ_i) to be the transition labeling the double edge departing from φ_i.

For our choice of p and q, premise J1 of rule CHAIN-J assumes the form

$$\text{(J1)}\quad \bigvee_{j=0}^{m} \varphi_j \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j,$$

which is trivially state-valid. We proceed to show that the P-state validity of premises J2–J4 follows from the P-validity of the diagram, for each $i = 1, \ldots, m$.

Premise J2 requires showing

$$\text{(J2)}\quad \rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \varphi'_0 \;\vee\; \varphi'_1 \;\vee\; \cdots \;\vee\; \varphi'_i,$$

for each $\tau \in \mathcal{T}$. Let $\varphi_{i_1}, \ldots, \varphi_{i_k}$ be the τ-successors of φ_i in the P-valid diagram, for $\tau \in \mathcal{T} - \{\tau_i, \tau_I\}$. By the requirement of weak acyclicity $i_1 \leq i, \ldots, i_k \leq i$. Since $\tau \neq \tau_i$, all the τ-edges departing from node φ are single-line edges and the following verification condition holds:

$$\text{V1.}\quad \rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \varphi'_i \;\vee\; \varphi'_{i_1} \;\vee\; \cdots \;\vee\; \varphi'_{i_k}.$$

Since $i_j \leq i$ for each $j = 1, \ldots, k$, the disjunction on the right-hand side of V1 is taken over a subset of the assertions appearing on the right-hand side of premise J2. It follow that J2 is state-valid for assertion φ_i and transition τ. For $\tau = \tau_I$, premise J2 holds trivially since ρ_{τ_I} implies $\varphi_i' = \varphi_i$. For $\tau = \tau_i$, premise J2 is implied by J3.

Premise J3 requires

$$\text{(J3)} \quad \rho_{\tau_i} \ \wedge \ \varphi_i \ \rightarrow \ \varphi_0' \ \vee \ \varphi_1' \ \vee \ \cdots \ \vee \ \varphi_{i-1}'.$$

Let $\varphi_{i_1}, \ldots, \varphi_{i_k}$ be the τ_i-successors of φ_i in the P-valid diagram. Since all τ_i-edges departing from φ_i are double, $i_j < i$ for $j = 1, \ldots, k$ and the following verification condition holds:

$$\text{V2.} \quad \rho_\tau \ \wedge \ \varphi_i \ \rightarrow \ \varphi_{i_1}' \ \vee \ \cdots \ \vee \ \varphi_{i_k}'.$$

Repeating the subset argument, this implies the state validity of premise J3.

Premise J4 is identical to condition V3 for every φ_i and τ_i, $i = 1, \ldots, m$.

Next, we consider the more general case of p and q which are not identical to $\bigvee_{j=0}^{m} \varphi_j$ and φ_0, respectively, but satisfy the implications J1 and J0. Applying rule MON-R to p, $\bigvee_{j=0}^{m} \varphi_j$, φ_0, and q (standing for p, q, r, and t in MON-R), we obtain the conclusion $p \ \Rightarrow \ \diamondsuit q$. ∎

Note that chain diagrams and their notion of validity are a conservative extension of the WAIT diagrams, introduced in Section 3.3 of the SAFETY book. The additional requirements that disallow a self-connecting edge all refer to double edges which are not present in WAIT diagrams. It follows that a P-valid CHAIN diagram is also P-valid for proving the nested waiting-for formula

$$\bigvee_{j=0}^{m} \varphi_j \ \Rightarrow \ \varphi_m \, \mathcal{W} \, \varphi_{m-1} \cdots \varphi_1 \, \mathcal{W} \, \varphi_0.$$

Example (program ANY-Y)

Consider again program ANY-Y (Fig. 1.2). The CHAIN diagram of Fig. 1.8 provides a graphical representation for the proof of the response property

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = 0}_{p=\Theta} \ \Rightarrow \ \diamondsuit\Big(\underbrace{at\_\ell_2 \ \wedge \ at\_m_1}_{q=\varphi_0}\Big),$$

for program ANY-Y.

The diagram identifies $\varphi_3, \ldots, \varphi_0$ as the intermediate assertions and m_0, ℓ_1, ℓ_0 as their corresponding helpful transitions. This CHAIN diagram is valid over program ANY-Y, which establishes the P-validity of

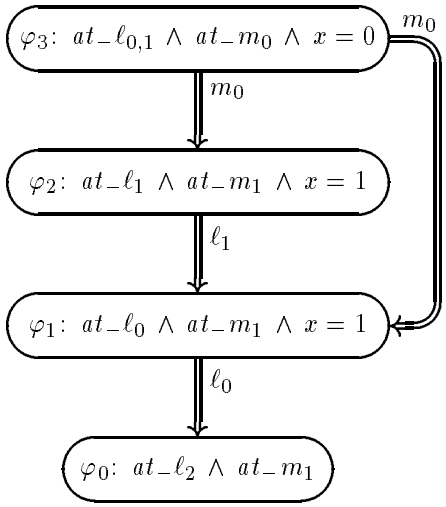


Fig. 1.8. CHAIN diagram for termination.

$$\bigvee_{j=0}^{3} \varphi_j \;\Rightarrow\; \diamondsuit(at\_\ell_2 \wedge at\_m_1)$$

over this program. Since (as shown above) $\Theta \rightarrow \varphi_3$, the second part of Claim 1.1 establishes the P-validity of the termination property

$$\Theta \;\Rightarrow\; \diamondsuit(at\_\ell_2 \wedge at\_m_1). \quad \blacksquare$$

The Advantages of Diagrams

One of the advantages of the presentation of a CHAIN-J proof sketch by a CHAIN diagram, over its presentation by a list of assertions and their corresponding helpful transitions is that the diagram provides a stronger (and more detailed) version of premises J2 and J3 than is standardly provided by rule CHAIN-J and a list of the assertions and helpful transitions.

Consider, for example, the proof presented in Fig. 1.8. Both the diagram and the textual proof identify φ_2 as $at\_\ell_1 \wedge at\_m_1 \wedge x = 1$ and ℓ_1 as its helpful transition.

However, while rule CHAIN-J suggests that we prove for premise J3 the verification condition

$$\{\varphi_2\}\ \ell_1\ \{\varphi_0 \vee \varphi_1\},$$

the diagram claims that the even stronger condition

$$\{\varphi_2\}\ \ell_1\ \{\varphi_1\}$$

is P-state valid. This results from the fact that there is no ℓ_1-edge connecting φ_2 to φ_0.

Encapsulation Conventions

There are several encapsulation conventions that lead to more structured hierarchical diagrams and improve the readability and manageability of large complex diagrams. These conventions were introduced in Section 3.3 of the SAFETY book and we reproduce them here briefly, to make the presentation self contained. The basic construct of encapsulation is that of a *compound node* that may contain several internal nodes. The encapsulation conventions attribute to a compound node aspects and relations that are common to all of their contained nodes. We refer to the contained nodes as *descendants* of the compound node. Nodes that are not compound are called *basic nodes*. We use three encapsulation conventions.

- *Departing edges*

  An edge departing from a compound node is interpreted as though it departed from each of its descendants. This is represented by the graphical equivalence of Fig. 1.9.

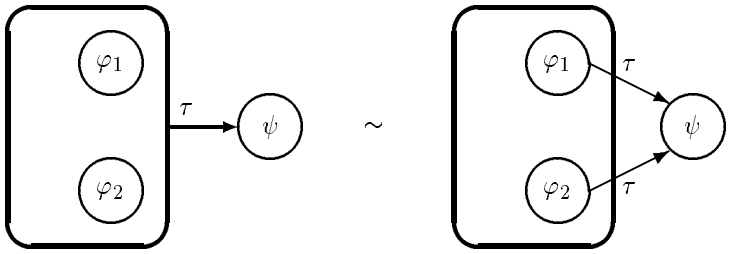


Fig. 1.9. Departing edges.

- *Arriving edges*

  In a similar way, an edge arriving at a compound node is interpreted as though it arrived at each of its descendants. This is represented in the graphical equivalence of Fig. 1.10.

- *Common factors*

  An assertion φ labeling a compound node is interpreted as though it were a conjunct added to each of the labels of its descendants. This is represented by the graphical equivalence of Fig. 1.11. We refer to φ as a *common factor* of the two nodes.

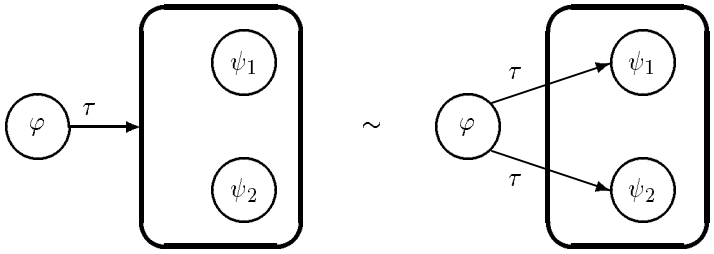


Fig. 1.10. Arriving edges.

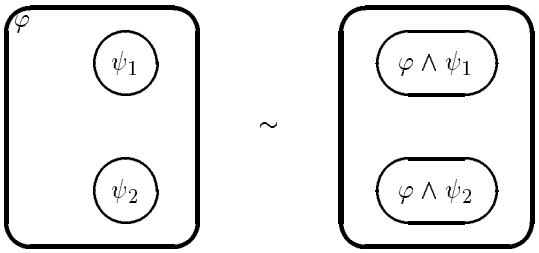


Fig. 1.11. Common factors.

Example In Fig. 1.12 we present a verification diagram which is the encapsulated version of the verification diagram of Fig. 1.8.

This encapsulation uses the arriving edge convention to denote by a single arrow the two edges connecting φ_3 to φ_2 and to φ_1. It uses the common factor convention to simplify the presentation of φ_1 and φ_2. ∎

Additional Examples

Let us consider a few more examples for the application of rule CHAIN-J and CHAIN diagrams, illustrating the encapsulation conventions.

Example (Peterson's Algorithm — version 1)

For the next example, we return to program MUX-PET1, Peterson's algorithm for mutual exclusion (Fig. 1.13). This program was previously studied in Section 1.4 of the SAFETY book, where we established for it the following invariants:

$$\psi_0\colon \quad s = 1 \ \vee \ s = 2$$

$$\psi_1\colon \quad y_1 \ \leftrightarrow \ at\_\ell_{3..5}$$

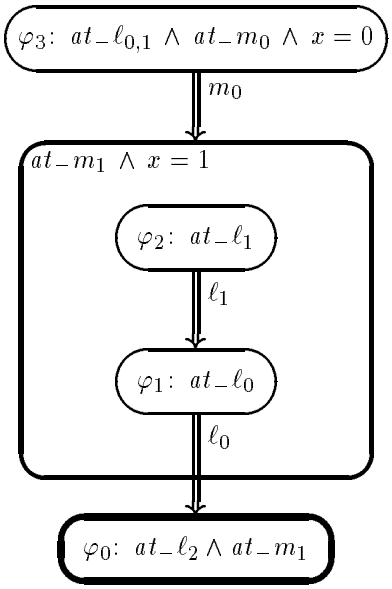


Fig. 1.12. Encapsulated version of CHAIN diagram for termination.

$$\psi_2: \quad y_2 \;\leftrightarrow\; at\_m_{3..5}.$$

We wish to prove for this program the response property of accessibility, given by

$$\underbrace{at\_\ell_2}_{p} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_4}_{q} \ .$$

To use rule CHAIN-J or its diagram representation, we have to identify intermediate assertions that characterize the intermediate situations between the starting assertion p: at_ℓ_2 and the goal assertion q: at_ℓ_4. It is obvious that the first helpful step in the progress from ℓ_2 to ℓ_4 is process P_1 moving from ℓ_2 to ℓ_3. Consequently, we can safely take φ_m to be at_ℓ_2 and the helpful transition τ_m to be ℓ_2. We cannot yet determine the value of m because it depends on the number of helpful steps necessary to get from ℓ_3 to ℓ_4. We can now concentrate on showing how to get from ℓ_3 to ℓ_4.

Similar to the heuristics employed in the application of rule NWAIT (Section 3.3 of the SAFETY book), it is often useful to work backwards from the goal assertion at_ℓ_4. Consequently, we take φ_0 to be at_ℓ_4.

For the previous intermediate assertion φ_1, we look for situations that are only one helpful step away from φ_0. Clearly, the only transition that can accomplish φ_0 in one step is ℓ_3. If we choose the helpful transition τ_1 to be ℓ_3, then

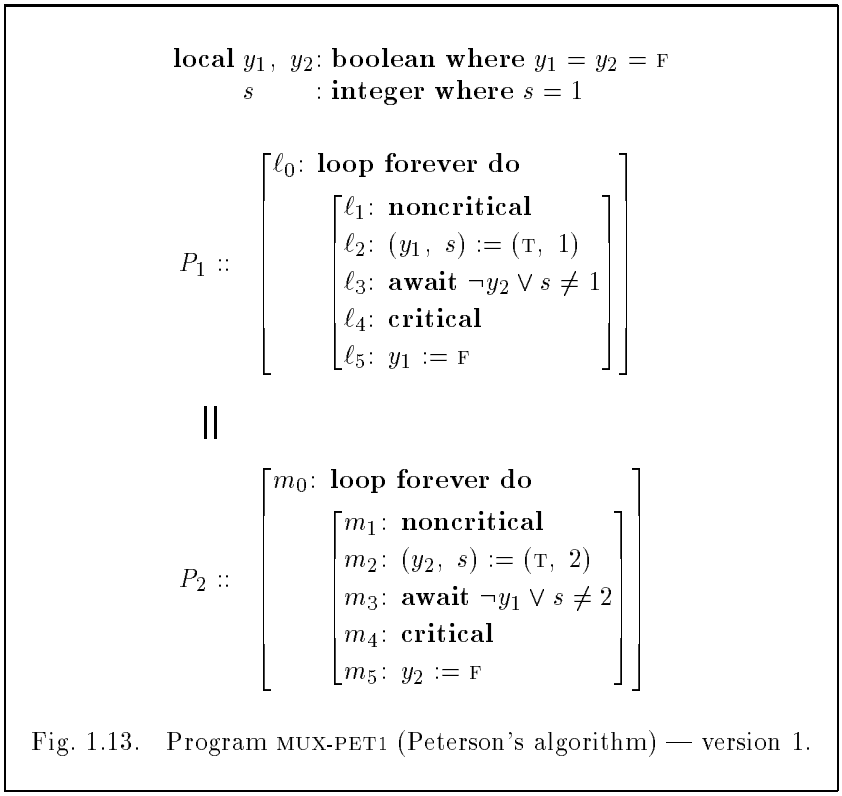


Fig. 1.13. Program MUX-PET1 (Peterson's algorithm) — version 1.

the appropriate φ_1 is the assertion characterizing all the states on which ℓ_3 is enabled. Therefore, as φ_1 we take the enabling condition of ℓ_3

$$\varphi_1: \quad at\_\ell_3 \ \wedge \ (\neg y_2 \ \vee \ s \neq 1).$$

Assertion φ_1 does not yet cover all the accessible states satsifying at_ℓ_3. Consequently, we cannot take m to be 2, and must search for additional assertions, characterizing states that satisfy at_ℓ_3 and that are one helpful step away from φ_1. Therefore, we look for transitions of P_2 that may change the disjunction $\neg y_2 \vee s = 2$ from F to T. The only candidate transition is m_5, which sets y_2 to F. Consequently, we take

$$\varphi_2: \quad at\_\ell_3 \ \wedge \ at\_m_5 \ \wedge \ y_2 \ \wedge \ s = 1.$$

The conjunct $y_2 \wedge s = 1$ can be safely added to φ_2 since all the states satisfying $at\_\ell_3 \wedge (\neg y_2 \vee s \neq 1)$ are already covered by φ_1.

Looking for $(at\_\ell_3 \wedge y_2 \wedge s = 1)$-states that are one helpful step away from φ_2, we easily identify

$$\varphi_3: \quad at\_\ell_3 \ \wedge \ at\_m_4 \ \wedge \ y_2 \ \wedge \ s = 1.$$

In a similar way, we can identify the preceding assertion as

$$\varphi_4\colon \quad at\_\ell_3 \ \wedge \ at\_m_3 \ \wedge \ y_2 \ \wedge \ s = 1.$$

Note that m_3, which is the helpful transition for φ_4, is enabled on all states satisfying φ_4.

At this point, we find out that the disjunction $\varphi_1 \vee \cdots \vee \varphi_4$ covers the range of all accessible (at_ℓ_3)-states. This is because P_2 must be in one of the locations $m_0, \ldots, m_4$. Due to ψ_2, the range $m_{0..2}$ is covered by φ_1 under the $\neg y_2$ disjunct. Locations $m_3 \ldots, m_5$ for $s \neq 1$ are covered by φ_1 under the disjunct $s \neq 1$, while the same locations for the case that $s = 1$ are covered by assertions $\varphi_4, \varphi_3, \varphi_2$, respectively.

In Fig. 1.14 we present a CHAIN diagram using the intermediate assertions constructed through the preceding analysis.

Note that we have grouped under φ_1 many possible states of P_2, and have not represented the movement of P_2 through them. This is justified by the fact that ℓ_3 is enabled on all of these states and is the transition declared as helpful for φ_1. In contrast, we separated m_3, m_4, and m_5, because the helpful transitions there changed from one of these states to the next. ▟

In **Problem 1.1**, the reader is requested to establish accessibility for another algorithm for mutual exclusion.

Example (Peterson's Algorithm — Version 2)

Consider the refined program MUX-PET2, version 2 of Peterson's algorithm, presented in Fig. 1.15.

In Section 1.4 of the SAFETY book, we established the following invariants for this program:

$$\chi_0\colon \quad s = 1 \ \vee \ s = 2$$

$$\chi_1\colon \quad y_1 \ \leftrightarrow \ at\_\ell_{3..6}$$

$$\chi_2\colon \quad y_2 \ \leftrightarrow \ at\_m_{3..6}.$$

We intend to verify the property of accessibility for program MUX-PET2, which can be expressed by the response formula

$$\underbrace{at\_\ell_2}_{p} \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_5}_{q} \ .$$

The construction of the appropriate verification diagram starts in a similar way to the diagram for program MUX-PET1 of the previous example. We take φ_m to be at_ℓ_2. From ℓ_2, process P_1 can proceed at its own pace to ℓ_3, which we take as φ_{m-1}. The next step taken by P_1 leads into ℓ_4 where a more detailed analysis is necessary.

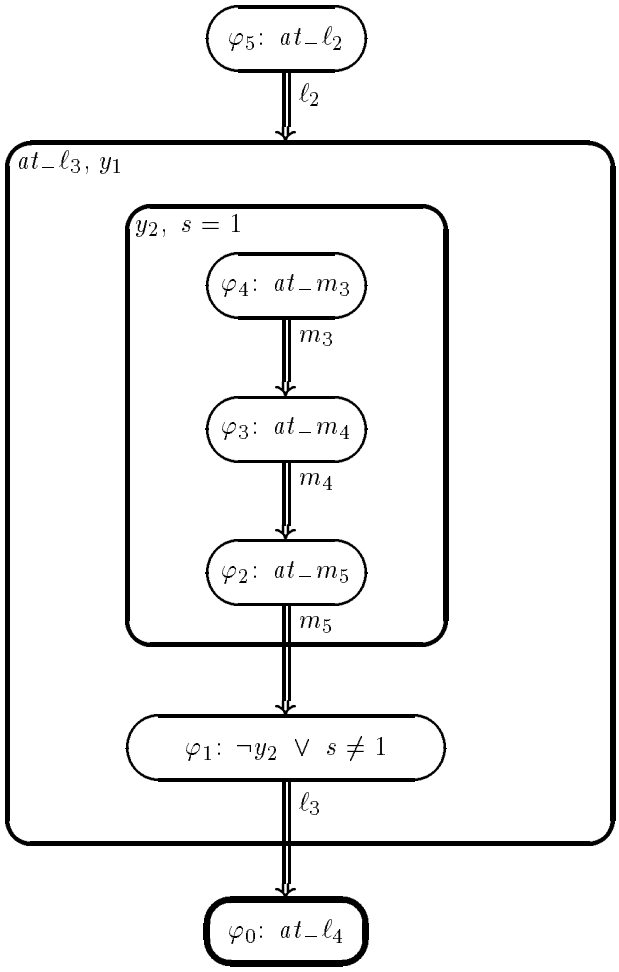


Fig. 1.14. CHAIN diagram for program MUX-PET1.

To perform this detailed analysis we take φ_0 to be the goal assertion at_ℓ_5. What should we take as φ_1? In the preceding case, we characterized φ_1 as being one helpful step away from φ_0. This characterization is not sufficient here. Another requirement is that if s' is a successor of a φ_1-state, then s' should satisfy either φ_1 or φ_0: at_ℓ_5. This shows that we cannot take φ_1 to be, as before, the assertion $at\_\ell_4 \wedge (\neg y_2 \vee s \neq 1)$. This is because the accessible state

$$s: \quad \langle \pi: \{\ell_4, m_2\},\ y_1: \mathrm{T},\ y_2: \mathrm{F},\ s: 1 \rangle$$

satisfies the candidate assertion $at\_\ell_4 \wedge (\neg y_2 \vee s \neq 1)$ but has an m_2-successor

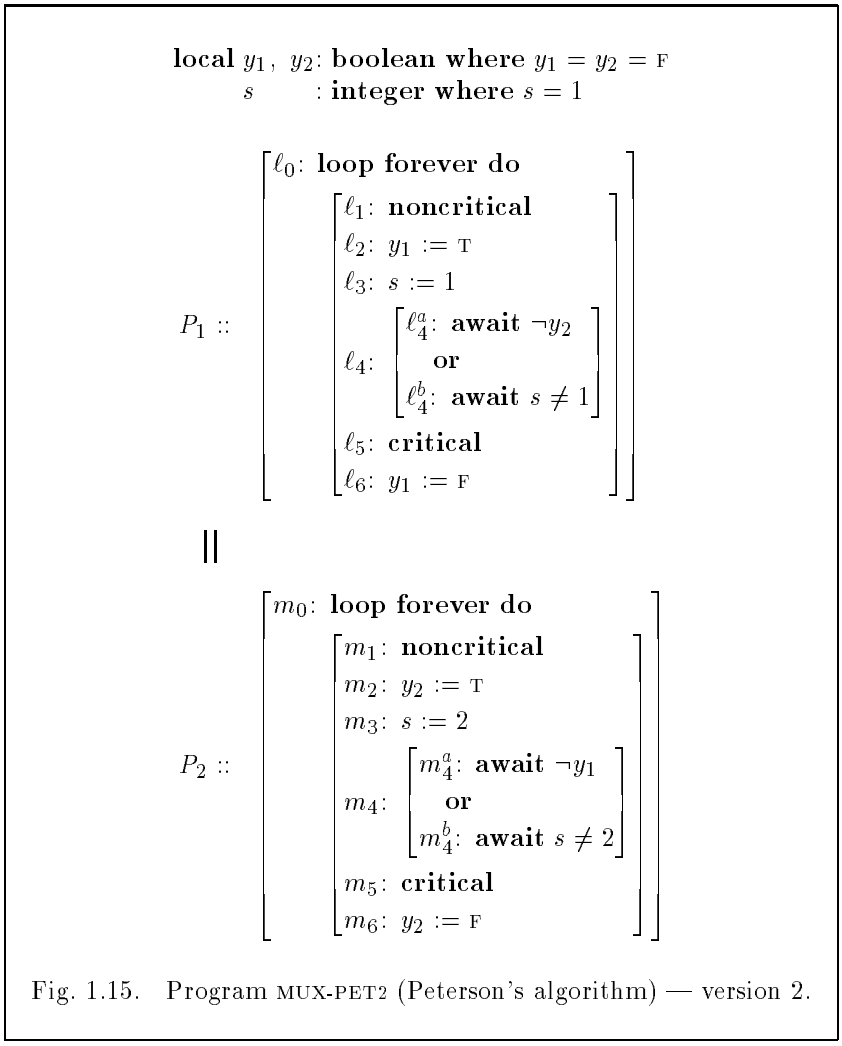


Fig. 1.15. Program MUX-PET2 (Peterson's algorithm) — version 2.

given by

$$s'\colon \quad \langle \pi\colon \{\ell_4, m_3\},\ y_1\colon \text{T},\ y_2\colon \text{T},\ s\colon 1 \rangle$$

which satisfies neither the candidate assertion nor φ_0.

We observe that the cause for this problem is the disjunct $\neg y_2$ which can be falsified (changed from T to F) by transition m_2 of P_2. There is no such problem with the disjunct $s \neq 1$ which cannot be falsified by P_2. Consequently, we take

$$\varphi_1\colon \quad at\_\ell_4 \ \wedge\ s \neq 1.$$

The only transition that can lead from a $\neg\varphi_1$-state to a φ_1-state is m_3. Therefore, we take φ_2 to be

$$\varphi_2\colon \quad at\_\ell_4 \ \wedge \ at\_m_3 \ \wedge \ s = 1.$$

We also realize that the transition leading into φ_2 is m_2 which changes y_2 from F to T and preserves the value of s. Consequently, we take φ_3 to be

$$\varphi_3\colon \quad at\_m_{0..2} \ \wedge \ \neg y_2 \ \wedge \ s = 1.$$

By now, we have covered all the states satisfying $at\_\ell_4 \wedge (\neg y_2 \vee s \neq 1)$. From now on, the analysis proceeds as it did for program MUX-PET1. The final CHAIN diagram is presented in Fig. 1.16.

This diagram partitions the range $m_{0..4}$ into three regions. The region $m_{0..2}$, represented by φ_3, guarantees the enableness of ℓ_4^a (but not the enableness of m_2 which is therefore drawn as a single edge). However it may evolve into φ_2, where no transition of P_1 is guaranteed to be enabled. Being at φ_2, m_3 is the helpful transition which eventually leads into φ_1. In φ_1, ℓ_4^b is enabled, and since P_2 cannot falsify $s \neq 1$, eventually ℓ_4^b is taken and leads to φ_0.

The edge labeled ℓ_4^a connecting node φ_1 to node φ_0 represents the possibility that ℓ_4^a may be enabled on a state satisfying $s \neq 1$. A more careful analysis shows that φ_1 in this diagram can be strenghened to the assertion

$$\widehat{\varphi}_1\colon \quad at\_m_4 \ \wedge \ s \neq 1 \ \wedge \ y_2,$$

and then this edge is unnecessary. ▟

In **Problem 1.2**, the reader is requested to verify accessibility for a family of mutual exclusion algorithms, known as the *bakery algorithms*.

Example (Dekker's algorithm)

Dekker's algorithm for solving the mutual exclusion problem is presented in program MUX-DEK of Fig. 1.17.

In comparison to Peterson's algorithm, Dekker's algorithm has a relatively simple safety proof but rather elaborate proof of accessibility.

- *Invariants*

In Section 1.4 of the SAFETY book we derived the following invariants for program MUX-DEK:

$$\chi_1\colon \quad t = 1 \ \vee \ t = 2$$

$$\chi_2\colon \quad y_1 \ \leftrightarrow \ (at\_\ell_{3..5,8..10})$$

$$\chi_3\colon \quad y_2 \ \leftrightarrow \ (at\_m_{3..5,8..10})$$

These are the invariants we needed to prove the mutual exclusion property, i.e., the invariance of

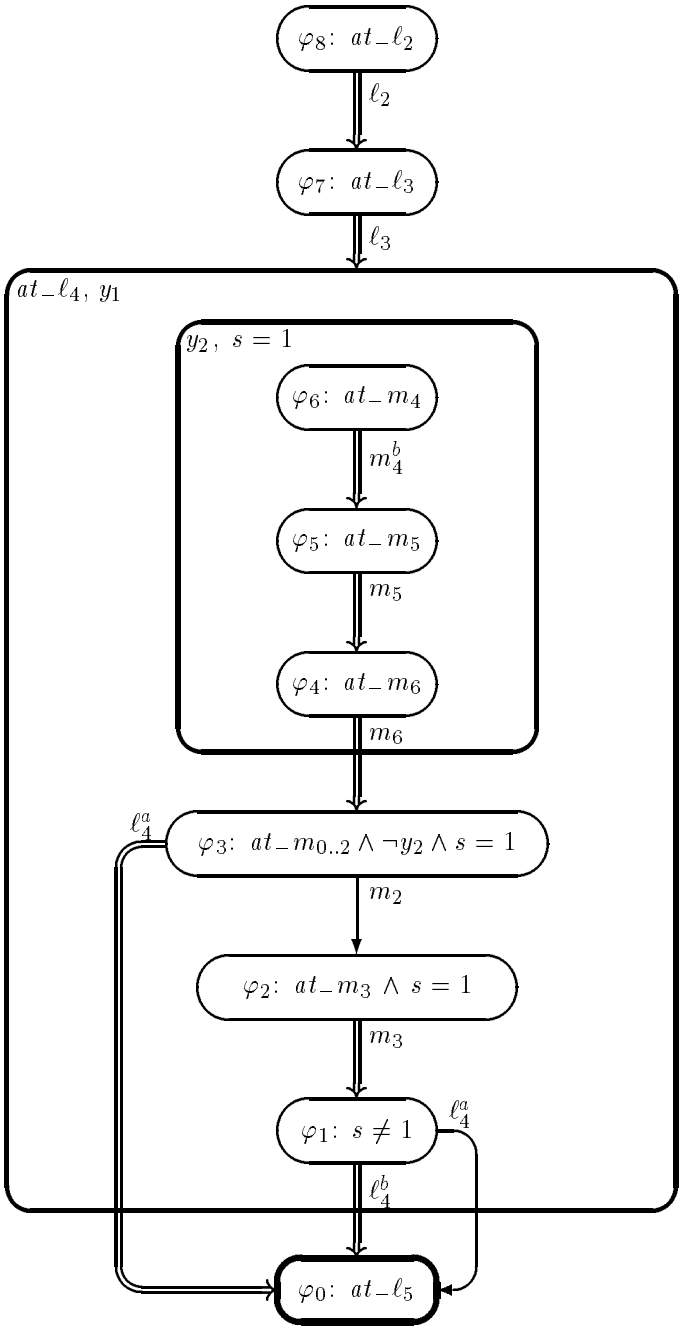


Fig. 1.16. CHAIN diagram for program MUX-PET2.

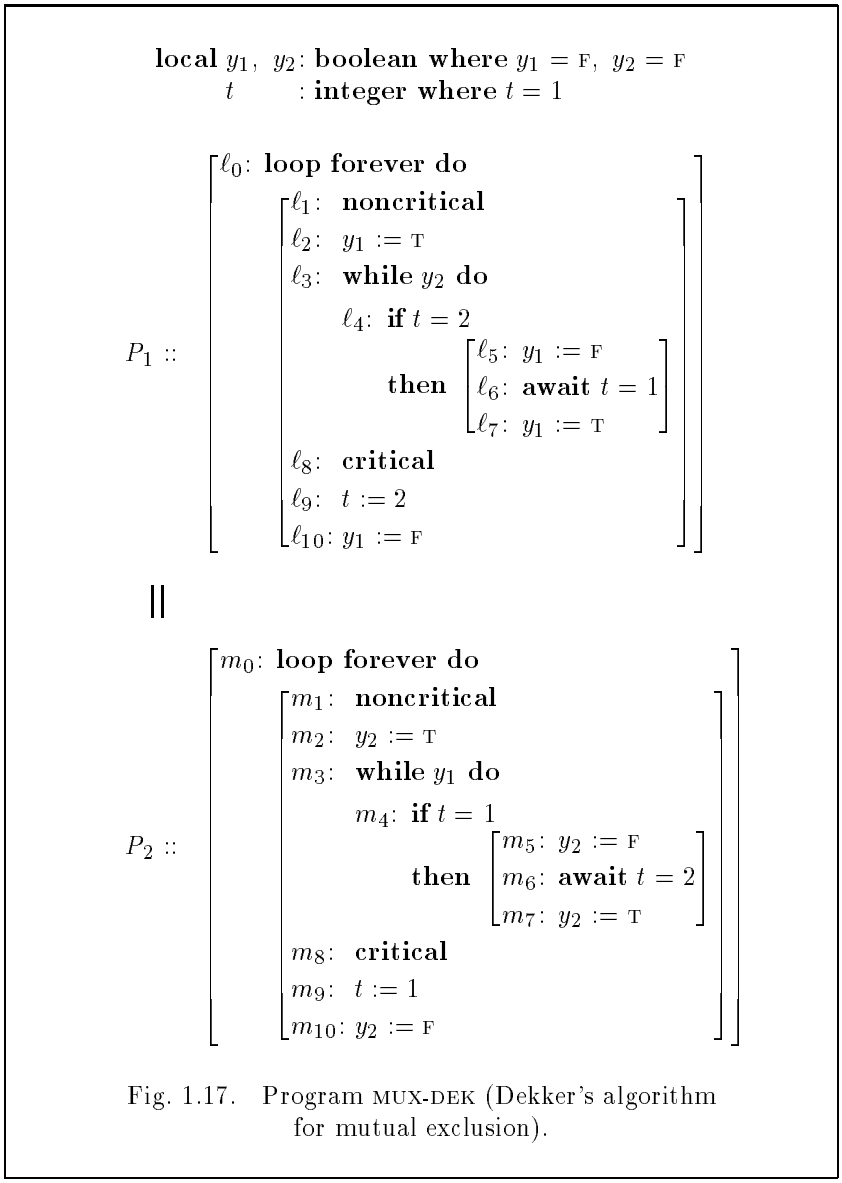


Fig. 1.17. Program MUX-DEK (Dekker's algorithm for mutual exclusion).

$$\chi_4: \quad \neg(at\_\ell_{8..10} \ \wedge \ at\_m_{8..10}).$$

As we will see, additional invariants are needed for the support of the response property. We will develop them as they are needed.

- *Response*

The main response property of this algorithm is, of course, that of accessibility. It is stated by

$$\psi: \quad at\_\ell_2 \;\Rightarrow\; \diamondsuit\, at\_\ell_8.$$

We partition the proof of the accessibility property into three lemmas, proving respectively.

Lemma A $\quad at\_\ell_2 \;\Rightarrow\; \diamondsuit\Big((at\_\ell_4 \;\wedge\; t=2) \;\vee\; (at\_\ell_{3..7} \;\wedge\; t=1) \;\vee\; at\_\ell_8\Big)$

Lemma B $\quad at\_\ell_4 \;\wedge\; t=2 \;\Rightarrow\; \diamondsuit(at\_\ell_{3..7} \;\wedge\; t=1)$

Lemma C $\quad at\_\ell_{3..7} \;\wedge\; t=1 \;\Rightarrow\; \diamondsuit\, at\_\ell_8.$

Obviously, the difficult part of the protocol is the loop at $\ell_{3..7}$. Being within this loop, P_1 is considered to have a higher priority when $t = 1$. Lemma A claims that if P_1 is just starting its journey towards the critical section, then it will either reach ℓ_4 with a lower priority, or get to $\ell_{3..7}$ with a higher priority, or reach ℓ_8. Lemma B claims that if P_1 is at ℓ_4 with a low priority it will stay within the loop and eventually gain a high priority. Lemma C shows that if P_1 is within this loop and has a higher priority, then it will eventually get to ℓ_8.

Clearly, by combining these three response properties, using the transitivity of response rule TRNS-R we obtain the required accessibility property.

■ *Proof of Lemma A*

The proof of the response property

$$at\_\ell_2 \;\Rightarrow\; \diamondsuit\Big((at\_\ell_4 \;\wedge\; t=2) \;\vee\; (at\_\ell_{3..7} \;\wedge\; t=1) \;\vee\; at\_\ell_8\Big)$$

is presented in the CHAIN diagram of Fig. 1.18.

It is easy to follow P_1 from ℓ_2 to ℓ_3. If $t = 1$ on entry to ℓ_3, then we are already at the goal $at\_\ell_{3..7} \wedge t = 1$. Otherwise, we enter ℓ_3 with $t = 2$, setting y_1 to T. Here we examine the possible locations of P_2. Assertions φ_1, φ_2, and φ_3 cover all the possibilities. The possible motions of P_2 within these three assertions consist of taking m_9, setting t to 1, which raises the priority of P_1 and attains the goal $at\_\ell_{3..7} \wedge t = 1$. The other possible movements are from φ_3 to φ_2, and then to φ_1. Being at $m_{3,4}$ with $y_1 = \mathrm{T}$ and $t = 2$, P_2 cannot move elsewhere. Transition mode ℓ_3^{T} is enabled on φ_1 and φ_3 states, while mode ℓ_3^{F} is enabled on φ_2. Both are helpful since they lead to $at\_\ell_4 \wedge t = 2$ and at_ℓ_8, respectively.

■ *Proof of Lemma B*

The proof of the response property

$$at\_\ell_4 \;\wedge\; t=2 \;\Rightarrow\; \diamondsuit(at\_\ell_{3..7} \;\wedge\; t=1)$$

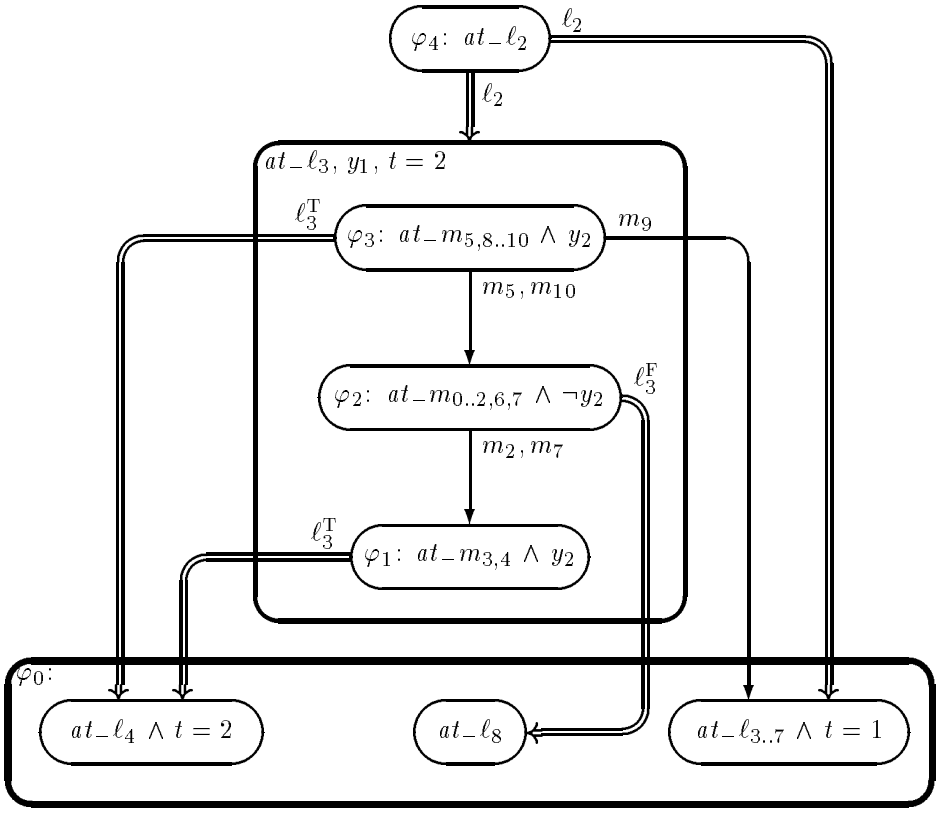


Fig. 1.18. CHAIN diagram for Lemma A.

is presented in the CHAIN diagram of Fig. 1.19.

From ℓ_4, P_1 proceeds to ℓ_5 since $t = 2$, and then to ℓ_6 while resetting y_1 to F. While being at $\ell_{4,5}$, P_2 may still set t to 1 by performing m_9, which leads to the goal $at\_\ell_{3..7} \wedge t = 1$.

However, once P_1 enters ℓ_6, it stays at ℓ_6 waiting for t to change to 1. At that point we have to inspect where P_2 may currently be. We consider as possible locations of P_2 all of m_2–m_9, tracing their possible flow under the relatively stable situation of $t = 2$, $y_1 = \mathrm{F}$. We see that all transitions are enabled and lead to m_9 which eventually sets t to 1 as required.

A tacit assumption made in this diagram is the exclusion of m_{10}, m_0, and m_1, as possible locations while P_1 is at ℓ_6 with $y_1 = \mathrm{F}$ and $t = 2$. This assumption must hold for the program, if we believe Lemma B to be valid. Indeed, consider the situation that P_1 is waiting at ℓ_6 with $y_1 = \mathrm{F}$ and $t = 2$, while P_2 is at m_1.

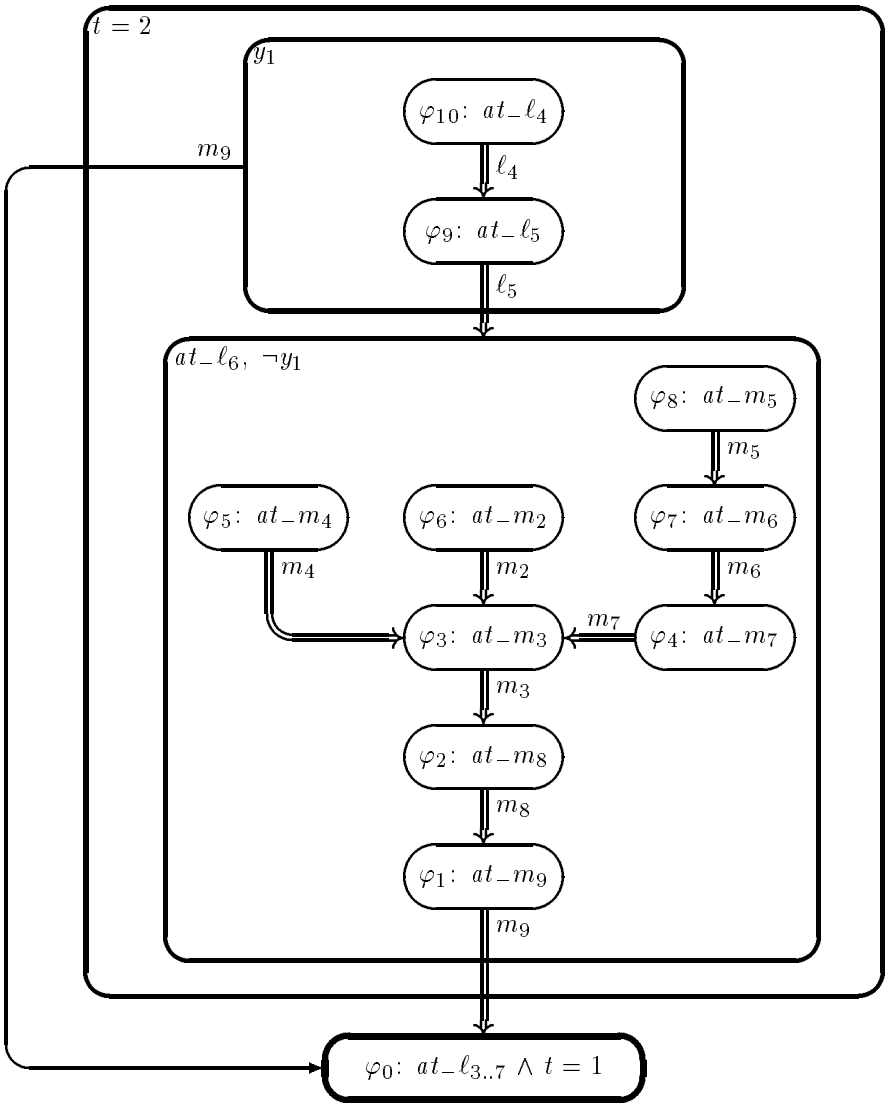


Fig. 1.19. CHAIN diagram for Lemma B.

Since P_2 is allowed to stay at the noncritical section forever, this would lead to a deadlock, denying accessibility from P_1.

We must therefore conclude that if the algorithm is correct, and guarantees accessibility to both processes, then the following assertion must be invariant

$$at\_\ell_6 \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}.$$

This invariance follows from the stronger invariant

$$\chi_5\colon \quad at\_\ell_{4..6} \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}$$

which we will prove.

By symmetry one can also require the invariance of

$$\chi_6\colon \quad at\_m_{4..6} \ \wedge \ t = 1 \ \rightarrow \ at\_\ell_{2..9}.$$

■ *Proof of invariant* χ_5 ⟪ Exercise? ⟫

Clearly,

$$\underbrace{\cdots \wedge \ at\_\ell_0 \ \wedge \cdots}_{\Theta} \ \rightarrow \ \underbrace{at\_\ell_{4..6} \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}}_{\chi_5}$$

holds.

Let us check the verification conditions for assertion χ_5, which are of the form

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ \neg y_1 \ \wedge \ \neg y_2 \ \wedge \ t = 1}_{\Theta} \ \wedge \ \underbrace{at\_\ell_{4..6} \ \wedge \ t = 2 \ \rightarrow \ at\_m_{2..9}}_{\chi_5} \ \rightarrow$$
$$\underbrace{at'\_\ell_{4..6} \ \wedge \ t' = 2 \ \rightarrow \ at'\_m_{2..9}}_{\chi'_5}.$$

There are three transitions that may potentially falsify assertion χ_5.

- Transition m_9

  Sets t to 1 which makes $t' = 2$ false and hence preserves the assertion.

- Transition ℓ_9

  Leads to at_ℓ_{10} which makes $at'\_\ell_{4..6}$ false.

- ℓ_3^{T} while $t = 2$

  This is possible only if $y_2 = \mathrm{T}$ which, by χ_3, implies $at\_m_{3..5} \vee at\_m_{8..10}$, and therefore $at\_m_{2..10}$. This almost gives us $at\_m_{2..9}$, with the exception of m_{10}. We thus need additional information that will exclude the possibility of P_2 being at m_{10} while $t = 2$.

Clearly, while entering m_{10} from m_9, P_2 sets t to 1. Can P_1 change it back to 2, while P_2 is still at m_{10}? The answer is no, because m_{10}, as we see in χ_4, is still a part of the critical section and is therefore exclusive of ℓ_9, the only statement capable of changing t to 2.

This suggests the invariant

$$\chi_7\colon \quad at\_m_{10} \ \rightarrow \ t = 1$$

and its symmetric counterpart

$$\chi_8: \quad at\_\ell_{10} \ \rightarrow \ t = 2.$$

To prove χ_7, we should inspect two transitions:

- Transition m_9

  Sets t to 1.

- ℓ_9 while at_m_{10}

  Impossible due to χ_4.

This establishes χ_7 and similarly χ_8. Having χ_7 we can use it to show that the last transition considered in the proof of χ_5, namely ℓ_3^{T} while $t = 2$, implies $at\_m_{2..9}$, which establishes χ_5.

- *Proof of Lemma C*

Lemma C states that if P_1 is within the waiting loop $\ell_{3..7}$ with higher priority, i.e., $t = 1$, then eventually it will reach ℓ_8. It is stated by

$$at\_\ell_{3..7} \ \wedge \ t = 1 \ \Rightarrow \ \diamondsuit \, at\_\ell_8.$$

The proof is presented in the CHAIN diagram of Fig. 1.20.

Note our efforts to minimize the number of assertions by grouping together situations with different control configurations, wherever possible. Thus for all the states where $y_1 = \mathrm{T}$ and P_1 is either at ℓ_3 or at ℓ_4, we do not distinguish between these two possibilities, but partition the diagram according to the location of P_2. This is because, in this general situation, it is P_2 which is the helpful process and we have to trace its progress. On the other hand, when $y_2 = \mathrm{F}$, P_1 becomes the helpful process and we start distinguishing between the cases of at_ℓ_3 and at_ℓ_4, while lumping together the locations of P_2 into two groups: $m_{0..2,7}$ and m_6. These two groups must be distinguished because it is possible (though not guaranteed) to exit the first group into a situation where $y_2 = \mathrm{T}$, but it is impossible to exit m_6 into such a situation. This is because when P_2 is at m_6 with $t = 1$, it cannot progress until t is changed to 2. ∎

In **Problem 1.3**, the reader is requested to prove accessibility for two variants of Dekker's algorithm.

Case Splitting According to the Helpful Transitions

In the preceding examples, the main reason for using rule CHAIN-J with $m > 1$ intermediate assertions has been that the program requires m helpful steps to reach the goal. In most of these applications there always was a worst case computation that actually visited each of the assertions, starting with φ_m and proceeding through $\varphi_{m-1}, \varphi_{m-2}, \ldots$ up to $\varphi_0 = q$.

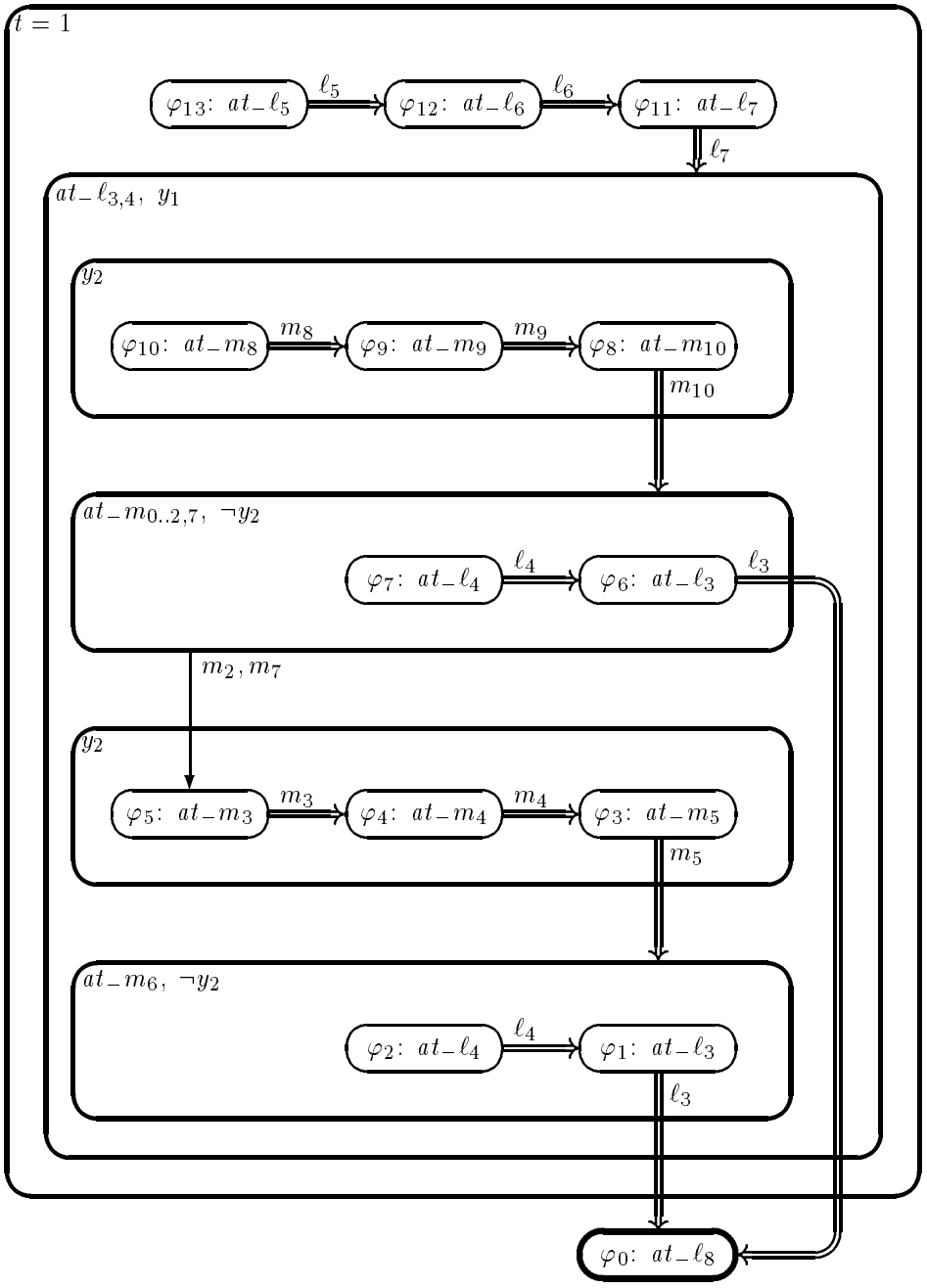


Fig. 1.20. CHAIN diagram for Lemma C.

This is not the only motivation for using several intermediate assertions. Another good reason for wishing to partition the state space lying between p and q into several assertions is that different states in that space may require different helpful transitions for getting them closer to the goal.

Example (maximum)

Consider, for example, program MAX presented in Fig. 1.21. This program places in output variable z the maximum of inputs x and y. The program consists of two parallel statements that compare the values of x and y.

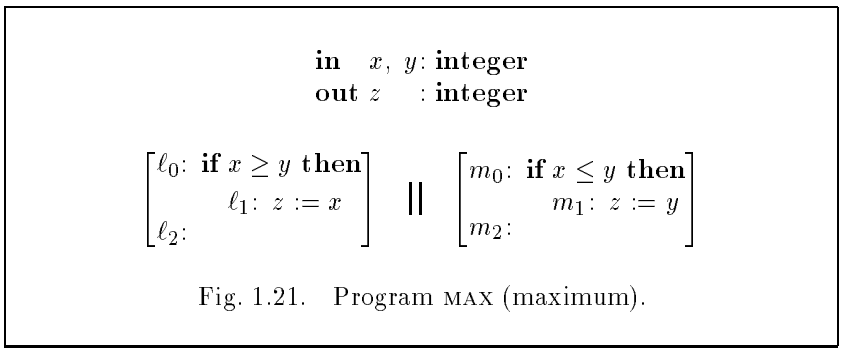


Fig. 1.21. Program MAX (maximum).

The response statement we would like to prove for this program is

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0}_{p} \ \Rightarrow \ \diamondsuit \underbrace{maximal(z,\ x,\ y)}_{q},$$

where $maximal(z, x, y)$ stands for the formula

$$maximal(z,\ x,\ y): \quad (z = x \ \vee \ z = y) \ \wedge \ z \geq x \ \wedge \ z \geq y,$$

claiming that z is the maximum of x and y.

Clearly, the goal of this response property is achieved in the helpful steps which are either ℓ_0 and ℓ_1 or m_0 and m_1. Perhaps one would expect a proof of this property by rule RESP-J that only uses one intermediate assertion φ.

However, no such proof exists. The reason for this is that we cannot identify a *single* transition that is helpful for all the states satisfying p: $at\_\ell_0 \wedge at\_m_0$. Clearly, for all states satisfying $p \wedge x \geq y$, ℓ_0 is the helpful transition, while for states satisfying $p \wedge x \leq y$, m_0 is the helpful transition. Consequently, we need at least *four* intermediate assertions in a proof of this property by rule CHAIN-J.

We choose the following assertions and helpful transitions

$$\varphi_4: \quad at\_m_0 \ \wedge \ x \leq y \qquad\qquad \tau_4: \quad m_0$$

$$\varphi_3: \quad at\_\ell_0 \ \wedge \ x \geq y \qquad\qquad \tau_3: \quad \ell_0$$

φ_2: $at\_m_1 \ \wedge \ x \leq y$ $\quad\quad$ τ_2: m_1

φ_1: $at\_\ell_1 \ \wedge \ x \geq y$ $\quad\quad$ τ_1: ℓ_1

φ_0: q.

It is straightforward to verify that all the premises of rule CHAIN-J are satisfied by this choice.

In Fig. 1.22, we present a CHAIN diagram for the proof of the considered response property.

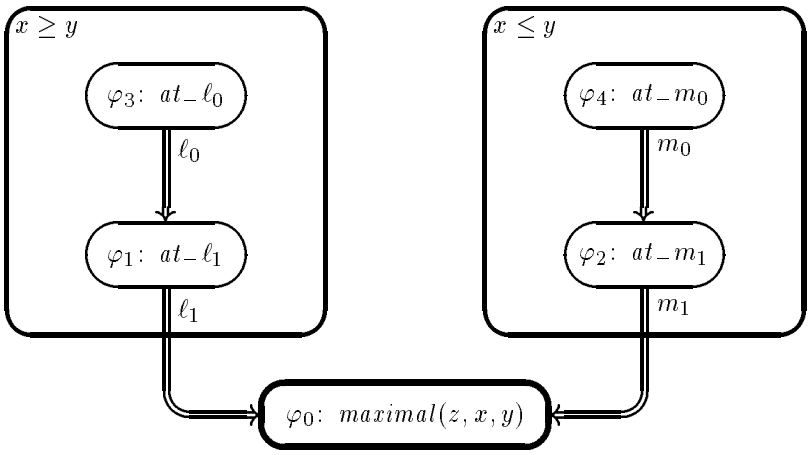


Fig. 1.22. CHAIN diagram for program MAX.

Assertions φ_3 and φ_4 partition (non-exclusively) the situation $at\_\ell_0 \wedge at\_m_0$ into states for which ℓ_0 is helpful and has not been taken yet, and states for which m_0 is helpful and has not been taken yet.

It is not difficult to verify that taking ℓ_0 from a φ_3-state, as well as taking m_0 from a φ_4-state, leads to φ_1 and φ_2, respectively. Choosing φ_4 to rank above φ_3 is quite arbitrary. In particular, we do not have a computation that goes from φ_4-states to φ_3-states. Every computation follows either the φ_3, φ_1 route or the φ_4, φ_2 route. ∎

1.4 Well-Founded Rule

In rule CHAIN-J we treated each of the participating assertions $\varphi_0, \ldots, \varphi_m$ as separate entities, and made no attempt to find a uniform representation for φ_j as

a single formula involving j. This approach is adequate for response properties which require a *bounded* number of steps for their achievement, e.g., at most five in the case of program MUX-PET1 (Fig. 1.14). The bound must be uniform and independent of the initial state.

There are many cases, however, in which no such bound can be given a priori. To deal with these cases, we must generalize the induction over a fixed finite subrange of the integers, such as $0, 1, \ldots, m$ in rule CHAIN-J, into an explicit induction over an arbitrary well-founded relation.

Well-Founded Domains

We define a *well-founded domain* $(\mathcal{A}, \succ)$ to consist of a set $\mathcal{A}$ and a *well-founded order* relation $\succ$ on $\mathcal{A}$. A binary relation $\succ$ is called an *order* if it is

- transitive: $a \succ b$ and $b \succ c$ imply $a \succ c$, and
- irreflexive: $a \succ a$ for no $a \in \mathcal{A}$.

The relation $\succ$ is called *well-founded* if there does not exist an infinitely descending sequence $a_0, a_1, \ldots$ of elements of $\mathcal{A}$ such that

$$a_0 \succ a_1 \succ \cdots .$$

A typical example of a well-founded domain is $(\mathbb{N}, >)$, where $\mathbb{N}$ are the natural numbers (including 0) and $>$ is the greater-than relation. Clearly, $>$ is well-founded over the natural numbers, because there cannot exist an infinitely descending sequence of natural numbers

$$n_0 > n_1 > n_2 > \ldots .$$

For an arbitrary order relation $\succ$ on $\mathcal{A}$, we define its *reflexive extension* $\succcurlyeq$ to hold between $a, a' \in \mathcal{A}$, written $a \succcurlyeq a'$, if either $a \succ a'$ or $a = a'$.

The Lexicographic Product

Given two well-founded domains, $(\mathcal{A}_1, \succ_1)$ and $(\mathcal{A}_2, \succ_2)$, we can form their *lexicographical product* $(\mathcal{A}, \succ)$, where

- $\mathcal{A}$ is defined as $\mathcal{A}_1 \times \mathcal{A}_2$, i.e., the set of all pairs (a_1, a_2), such that $a_1 \in \mathcal{A}_1$ and $a_2 \in \mathcal{A}_2$.
- $\succ$ is an order defined for $(a_1, a_2), (b_1, b_2) \in \mathcal{A}$ by

$$(a_1, a_2) \succ (b_1, b_2) \quad \text{iff} \quad a_1 \succ_1 b_1 \quad \text{or} \quad a_1 = b_1 \ \wedge \ a_2 \succ_2 b_2 .$$

Thus, in comparing the two pairs (a_1, a_2) and (b_1, b_2), we first compare a_1 against b_1. If $a_1 \succ_1 b_1$, then this determines the relation between the pairs to be $(a_1, a_2) \succ (b_1, b_2)$. If $a_1 = b_1$, we compare a_2 with b_2, and the result of this comparison determines the relation between the pairs.

The order $\succ$ is called *lexicographic*, which implies that, as when searching in a dictionary, we locate the position of a word by checking the first letter first and only after locating the place where the first letter matches, do we continue matching the subsequent letters.

The importance of the lexicographic product follows from the following claim:

Claim (lexicographic product)

If the domains $(\mathcal{A}_1, \succ_1)$ and $(\mathcal{A}_2, \succ_2)$ are well-founded, then so is their lexicographic product $(\mathcal{A}, \succ)$.

Clearly, by the above, the domain $(\mathbb{N}^2, \succ)$, where $\succ$ is the lexicographic order between pairs of natural numbers, is well-founded. This order is defined by

$$(n_1, n_2) \succ (m_1, m_2) \quad \text{iff} \quad n_1 > m_1 \quad \text{or} \quad n_1 = m_1 \ \wedge \ n_2 > m_2.$$

According to this definition

$$(10, 20) \succ (5, 15) \qquad (1, 0) \succ (0, 100) \qquad (1, 5) \succ (1, 3).$$

New well-founded domains can be constructed by taking lexicographic products of more than two well-founded domains. Applying this construction to the domain $(\mathbb{N}, >)$ of natural numbers, we obtain the domain $(\mathbb{N}^k, \succ)$, for $k \geq 2$, where $\succ$ is the lexicographic order between k-tuples of natural numbers. The order $\succ$ is defined by

$$(n_1, \ldots, n_k) \succ (m_1, \ldots, m_k) \quad \text{iff} \quad n_1 = m_1, \ \ldots, \ n_{i-1} = m_{i-1}, \ n_i > m_i \quad \text{for some } i, \ 1 \leq i \leq k.$$

For example, for $k = 3$

$$(7, \ 2, \ 1) \succ (7, \ 0, \ 45).$$

It is easy to show that the domain $(\mathbb{N}^k, \succ)$ is well-founded.

The Rule

Let $(\mathcal{A}, \succ)$ be a well-founded domain. As in rule CHAIN-J, we use several intermediate assertions $\varphi_1, \ldots, \varphi_m$ to describe the evolution from p to $q = \varphi_0$. Rule CHAIN-J uses the index of the assertion as a measure of the distance from the goal q. The rule presented here associates an explicit *ranking function* δ_i with each assertion φ_i, $i = 0, \ldots, k$. The function δ_i maps states into the set $\mathcal{A}$ and is intended to measure the distance of the current state to a state satisfying the goal q.

We refer to the value of δ_i in a φ_i-state as a *rank* of the state. The well-founded rule WELL for response properties is given in Fig. 1.23. Premise W1 states

that every p-position satisfies one of $\varphi_0, \ldots, \varphi_m$. Premise W2 states that every φ_i-position with positive i and rank u is eventually followed by a position which satisfies some φ_j, with a rank lower than u.

For assertions p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m$: $\Sigma \mapsto \mathcal{A}$

W1. $p \rightarrow \bigvee_{i=0}^{m} \varphi_i$

W2. $\varphi_i \wedge \delta_i = u \Rightarrow \Diamond\left(\bigvee_{j=0}^{m} (\varphi_j \wedge u \succ \delta_j)\right)$ for $i = 1, \ldots, m$

$p \Rightarrow \Diamond q$

Fig. 1.23. Rule WELL (well-founded response).

Justification It is straightforward to justify rule WELL. Consider a computation σ that satisfies premises W1, W2, and let t_1 be a position in σ which satisfies p. By W1, some φ_i is satisfied at t_1. If it is $\varphi_0 = q$, we are done. Otherwise, let φ_{i_1}, $i_1 > 0$, be the assertion holding at t_1 and let u_1 denote the rank of the state at position t_1. By W2, there exists a position t_2, $t_2 \geq t_1$, such that some φ_j holds at t_2 with a rank $u_2 \in \mathcal{A}$, such that $u_1 \succ u_2$. If $j = 0$, we are done. Otherwise, we proceed to locate a position $t_3 \geq t_2$.

In this way we construct a sequence of positions

$$t_1 \leq t_2 \leq t_3 \leq \ldots,$$

and a corresponding sequence of elements from $\mathcal{A}$ (ranks)

$$u_1 \succ u_2 \succ u_3 \succ \ldots,$$

such that either the sequence is of length k and $q = \varphi_0$ holds at the position t_k, or the sequence is infinite and some φ_j, $j > 0$, holds at each t_i with rank $\delta_j = u_i$ there. The later case is impossible since that would lead to an infinitely descending sequence of elements of $\mathcal{A}$, in contrast to the well-foundedness of $\succ$ over $\mathcal{A}$. It follows that for some $t_k \geq t_1$, $q = \varphi_0$ holds at t_k, which shows that $\Diamond q$ holds at t_1. ∎

Example (factorial)

Consider program FACT of Fig. 1.24. This program computes in z the factorial of a nonnegative integer x. We wish to prove for this program the response property

$$\underbrace{at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{q=\varphi_0} \ .$$

in x: **integer where** $x \geq 0$
local y: **integer where** $y = x$
out z: **integer where** $z = 1$

$$\left[\begin{array}{l} \ell_0\text{: } \textbf{while } y > 0 \textbf{ do} \\ \qquad \ell_1\text{: } (y,\ z) := (y-1,\ z \cdot y) \\ \ell_2\text{:} \end{array}\right]$$

Fig. 1.24. Program FACT (factorial).

Intending to use rule WELL with $m = 1$, it only remains to choose the assertion φ_1, and the ranking functions δ_0 and δ_1. This necessitates the identification of a well-founded domain $(\mathcal{A}, \succ)$, where $\mathcal{A}$ serves as the range of δ_i. Obviously, φ_1 should describe the intermediate stage in the process of getting from p to q, and δ_1 should measure the distance of this intermediate stage from the goal $q = \varphi_0$. Premise W2 ensures that steps in the computation always bring us closer to the goal.

For program FACT, a good measure of the distance from termination is the value of y. This is because when we are at ℓ_0, there are y more iterations of the *while* loop before the program terminates. We therefore choose $(\mathbb{N}, >)$ as our well-founded domain and $|y|$ as the ranking function. Thus,

$$(\mathcal{A},\ \succ) \ = \ (\mathbb{N},\ >), \qquad \delta_0\text{: } 0, \qquad \text{and} \qquad \delta_1\text{: } |y| + 1.$$

The choice of $\delta_0 = 0$ is natural because, being at a φ_0-state, we are already at the goal, and the distance to the goal can therefore be taken as 0.

The intermediate assertion φ_1 should represent the progress the computation has made, so that when $y = 0$, we can infer that $z = x!$. Clearly, the way the program operates is that it accumulates in z the product of the terms $x \cdot (x-1) \cdots$. In an intermediate stage, z contains the product $x \cdot (x-1) \cdots (y+1)$, which can also be expressed as $x!/y!$, provided $0 \leq y \leq x$.

We thus arrive at the intermediate assertion

$$\varphi\text{:} \quad at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y! \ .$$

It only remains to show that premises W1,W2 are satisfied by these choices.

- Premise W1

  This premise requires

  $$\underbrace{at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \ \rightarrow$$

  $$\underbrace{\cdots}_{\varphi_0} \ \vee \ \underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} .$$

  This implication is obviously valid.

- Premise W2

  This premise requires showing

  $$\underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ |y| + 1 = n \ \Rightarrow$$

  $$\diamondsuit \left(\begin{array}{l} \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{\varphi_0} \ \wedge \ n > \underbrace{0}_{\delta_0} \\ \quad \vee \\ \underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ n > \underbrace{|y| + 1}_{\delta_1} \end{array} \right) .$$

  Since $\varphi_1 \wedge |y| + 1 = n$ implies that $n > 0$, it is sufficient to prove this implication for every $n > 0$. As φ_1 implies $y \geq 0$, we may replace $|y|$ by y.

  Case $n = 1$:

  For this value of n, we prove

  $$\underbrace{at\_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ y + 1 = 1 \ \Rightarrow$$

  $$\diamondsuit \Big(\underbrace{at\_\ell_2 \ \wedge \ z = x!}_{\varphi_0} \ \wedge \ 1 > 0 \Big) ,$$

  which simplifies to

  $$at\_\ell_0 \ \wedge \ z = x! \ \wedge \ y = 0 \ \Rightarrow \ \diamondsuit (at\_\ell_2 \ \wedge \ z = x!) .$$

  This, of course, can be proven by a single application of rule RESP-J, observing that, under the situation described by the antecedent, only transition ℓ_0 is enabled, and taking it leads to $at\_\ell_2 \wedge z = x!$.

  Case $n > 1$:

  In this case, we will prove

$$\underbrace{at\_\ell_0 \ \wedge \ 0 \le y \le x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ y+1 = n > 1 \ \Rightarrow$$

$$\Diamond\Big(\underbrace{at\_\ell_0 \ \wedge \ 0 \le y \le x \ \wedge \ z = x!/y!}_{\varphi_1} \ \wedge \ n > y+1\Big) \ .$$

This can be proven by rule CHAIN-J using three assertions, ψ_0, ψ_1, and ψ_2.

The top assertion ψ_2 corresponds to $\varphi_1 \wedge y+1 = n$. Assertion ψ_1 describes the intermediate state, after passing the test of the *while* statement, and being at ℓ_1. The final assertion ψ_0 implies $\varphi_1 \wedge y+1 = n-1 < n$, and describes the situation after performing the assignment ℓ_1 and arriving back at ℓ_0. The verification diagram in Fig. 1.25 describes this proof.

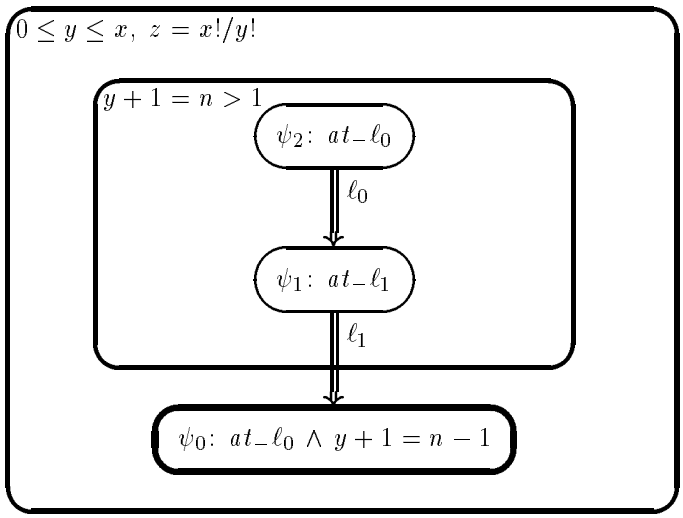


Fig. 1.25. Verification diagram for case $n > 1$.

Thus, by treating separately the cases $n = 1$ and $n > 1$, we conclude that premise W2 holds for every $n \ge 1$. This establishes that the conclusion

$$\underbrace{at\_\ell_0 \ \wedge \ x \ge 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \ \Rightarrow \ \Diamond \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{q}$$

of rule WELL is valid. ∎

A Rule with Nontemporal Premises

Rule WELL, used for proving response formulas, has as its premise W2, another response formula. This allows a recursive use of the rule, by which the temporal premise W2 is proved either by the simpler rule RESP-J, or by rule WELL again, only applied to simpler assertions. As a matter of fact, if we closely examine the proof of the previous example, we can identify there the use of those two options. For proving W2 for $n = 1$, we used rule RESP-J, since the response property for this case is accomplished in one step. On the other hand, the case of $n > 1$ accomplishes W2 in two steps, and we therefore had to use rule CHAIN-J.

However, in many cases, we do not need the recursive application of the rule, which means that premise W2 is proved directly by rule RESP-J. In these cases it is advantageous to replace the temporal premise W2 by the nontemporal premises of rule RESP-J, which are necessary for its derivation. This leads to a (combined) form of the rule in which all premises are nontemporal. Such a form is often more satisfactory because it explicitly manifests the power of the rule to derive temporal statements from nontemporal ones.

This leads to rule WELL-J (Fig. 1.26).

For assertions p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J}$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m \colon \Sigma \mapsto \mathcal{A}$

JW1. $p \rightarrow \bigvee_{j=0}^{m} \varphi_j$

JW2. $\rho_\tau \wedge \varphi_i \rightarrow \left[\begin{array}{l} \bigvee_{j=0}^{m} (\varphi'_j \wedge \delta_i \succ \delta'_j) \\ \vee \ (\varphi'_i \wedge \delta_i = \delta'_i) \end{array}\right]$ for every $\tau \in \mathcal{T}$

JW3. $\rho_{\tau_i} \wedge \varphi_i \rightarrow \bigvee_{j=0}^{m} (\varphi'_j \wedge \delta_i \succ \delta'_j)$

JW4. $\varphi_i \rightarrow En(\tau_i)$

} for $i = 1, \ldots, m$

$p \Rightarrow \diamondsuit q$

Fig. 1.26. Rule WELL-J (well-founded response under justice).

The rule requires finding auxiliary assertions φ_i and transitions τ_i, $i = 1, \ldots, m$, a well-founded domain $(\mathcal{A}, \succ)$, and ranking functions $\delta_i \colon \Sigma \mapsto \mathcal{A}$. Each assertion φ_i, $i > 1$, is associated with the transition τ_i that is helpful at positions satisfying φ_i, and with its own ranking function δ_i.

Premise JW1 requires that every p-position satisfies one of $\varphi_0, \ldots, \varphi_m$.

Premises JW2–JW4 impose three requirements for each $i = 1, \ldots, m$.

Premise JW2 requires that, taking any transition from a φ_i-position k, always leads to a successor position $k' = k + 1$, such that

- either some φ_j, $j = 0, \ldots, m$, holds at k' with a rank δ'_j lower than δ_i at k, or
- φ_i holds at k' with a rank δ'_i equal to the rank δ_i at k.

The main implication of premise JW2 is that if the situation has not improved in any noticeable way in going from k to k', i.e., the new rank still equals the old rank, at least we have not lost the identity of the helpful transition and the transition that was helpful in k is also helpful at k'.

Premise JW3 requires that transition τ_i, which is helpful for φ_i, always leads from a φ_i-position k to a next position which satisfies some φ_j and has a rank lower than that of k, i.e., $\delta_i \succ \delta'_j$.

Premise JW4 requires that the helpful transition τ_i is enabled at every φ_i-position.

Justification To justify the rule, assume a computation such that p holds at position k, and no later position $i \geq k$, satisfies $q = \varphi_0$. By this assumption and JW1, some φ_j, $j > 0$, must hold at position k. Let φ_{i_1} be the formula holding at k, and denote the rank δ_{i_1} at k by u_1. By JW4, transition τ_{i_1} is enabled at position k.

Consider the transition τ taken at position k, leading into position $k+1$. By JW2 and JW3, either position $k + 1$ has a lower rank u_2, $u_2 \prec u_1$, or it has the same rank, but then τ_{i_1} is still the helpful transition at $k+1$ and is enabled there. In the case that the rank is still u_1, we can continue the argument from $k + 1$ to $k + 2$, $k + 3$, etc. However, we cannot have all positions $i > k$ with the same rank. To see this, assume that all positions beyond k do have the same rank. By JW2 and JW4, this implies that τ_{i_1} is continuously helpful and enabled. By JW3, τ_{i_1} is not taken beyond k because taking it would have led to a state with a rank lower than u_1. Thus, our assumption that all positions beyond k have the same rank leads to the situation that τ_{i_1} is continuously enabled and not taken, violating the justice requirement for τ_{i_1}.

Thus, eventually, we must reach a position k_2, $k_2 > k$, with lower rank u_2, where $u_2 \prec u_1$. In a similar way we can establish the existence of a position $k_3 > k_2$, with rank u_3 where $u_3 \prec u_2$. Continuing in this manner, we construct

an infinitely descending sequence $u_1 \succ u_2 \succ u_3 \succ \cdots$ of elements of $\mathcal{A}$. This is impossible, due to the well-foundedness of $\succ$ on $\mathcal{A}$.

We conclude that every p-position must be followed by a q-position, establishing the consequence of the rule. ∎

Note that since premise JW3 implies premise JW2 for $\tau = \tau_i$, it is sufficient to check premise JW2 only for $\tau \neq \tau_i$.

Rule CHAIN-J can be viewed as a special case of rule WELL-J which uses $\delta_i = i$, for $i = 0, \ldots, m$, as ranking functions. It is not difficult to see that the premises J2, J3, and J4 of rule CHAIN-J correspond precisely to premises JW2, JW3, and JW4 of rule WELL-J. The well-founded domain used in this special case is the finite segment $[0..m]$ of the natural numbers ordered by $>$.

Example (factorial)

We use rule WELL-J to prove that program FACT of Fig. 1.24 satisfies the response property of total correctness

$$\underbrace{at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1}_{p} \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_2 \ \wedge \ z = x!}_{q=\varphi_0} .$$

Obviously, except for the terminating state, execution of program FACT alternates between states satisfying at_ℓ_0 in which ℓ_0 is the helpful transition, and states satisfying at_ℓ_1 in which ℓ_1 is helpful.

Consequently, we take $m = 2$ and use the following intermediate assertions.

$$\begin{aligned}
&\varphi_2\colon \quad at_\ell_0 \ \wedge \ 0 \leq y \leq x \ \wedge \ z = x!/y! \\
&\varphi_1\colon \quad at_\ell_1 \ \wedge \ 1 \leq y \leq x \ \wedge \ z = x!/y! \\
&\varphi_0\colon \quad at_\ell_2 \ \wedge \ z = x! \ .
\end{aligned}$$

Note that when control is at ℓ_1, y is required to be greater than or equal to 1.

It remains to determine the ranking functions δ_i, $i = 0, 1, 2$. Our previous analysis of the considered response property for program FACT (using rule WELL) identified $|y|$ as a good measure of progress over $(\mathbb{N}, >)$. Variable y keeps decreasing as the program gets closer to termination. Unfortunately, premise JW3 of rule WELL-J requires that δ decreases on *each* activation of a helpful transition. As we see, not every helpful transition causes $|y|$ to decrease. In particular, ℓ_0 does not change $|y|$. Consequently, we have to supplement $|y|$ by an additional component that will decrease when $|y|$ stays the same. This leads to the following choice:

$$\begin{aligned}
&\delta_2\colon \quad \big(|y|, \ 2\big) \\
&\delta_1\colon \quad \big(|y|, \ 1\big) \\
&\delta_0\colon \quad \big(0, \ 0\big) .
\end{aligned}$$

The corresponding well-founded domain is $(\mathbb{N}\times\{1,2\},\ \succ)$, where $\succ$ is the lexicographical order between pairs of integers.

In the previous proof of this property, we used the measure $|y|+1$ to ensure that the rank decreases also on the transition from ℓ_0 to ℓ_2. Since the use of pairs guarantees such a decrease by a decreasing second component, we can omit the $+1$ increment and take the first component to be simply $|y|$.

We may view the ranking function δ_i: $(|y|, i)$ as consisting of a *major* and a *minor* measures of progress. Function $|y|$ measures large steps of progress, such as one full iteration of the loop at ℓ_0. The minor component i measures smaller steps of progress. Observe that transition ℓ_1 actually causes the minor measure i to increase from 1 to 2, but at the same time it decreases the major measure $|y|$.

Let us consider the premises of rule WELL-J. Since both φ_i's imply $y \geq 0$, we may replace $|y|$ by y.

- Premise JW1

We prove JW1 by showing

$$\underbrace{at\_\ell_0 \ \wedge\ x \geq 0 \ \wedge\ y = x \ \wedge\ z = 1}_{p} \ \rightarrow$$

$$\cdots \ \vee\ \underbrace{at\_\ell_0 \ \wedge\ 0 \leq y \leq x \ \wedge\ z = x!/y!}_{\varphi_2},$$

which is obviously valid.

- Premises JW2, JW3 for $i = 2$

For $i = 2$ we will show

$$\rho_\tau \ \wedge\ \underbrace{at\_\ell_0 \ \wedge\ 0 \leq y \leq x \ \wedge\ z = x!/y!}_{\varphi_2} \ \rightarrow$$

$$\left(\begin{array}{l} \underbrace{at'\_\ell_2 \ \wedge\ z = x!}_{\varphi'_0} \ \wedge\ \underbrace{(y,2)}_{\delta_2} \succ \underbrace{(y',0)}_{\delta'_0} \\ \qquad \vee \\ \underbrace{at'\_\ell_1 \ \wedge\ 1 \leq y' \leq x \ \wedge\ z' = x!/y'!}_{\varphi'_1} \ \wedge\ \underbrace{(y,2)}_{\delta_2} \succ \underbrace{(y',1)}_{\delta'_1} \end{array}\right),$$

for each transition $\tau \in \{\ell_0, \ell_1\}$, not necessarily the helpful one. Obviously, this will satisfy both JW2 and JW3. Since at_ℓ_0 implies that transition ℓ_1 is disabled, the left-hand side of the implication for $\tau = \ell_1$ is false and the implication is trivially true.

For $\tau = \ell_0$, we prove the implication by separately considering the cases $y = 0$ and $y \neq 0$.

Case $y = 0$:

In this case ρ_{ℓ_0} implies at'_ℓ_2, $y' = y = 0$, and $z' = z$. Since $z = x!/y!$ and $y = 0$ imply $z = x!$, it follows that the left-hand side implies $\varphi_0' \ \wedge \ (0,2) \succ (0,0)$.

Case $y \neq 0$:

In this case φ_2 implies that $y > 0$, which together with ρ_{ℓ_0}, implies at'_ℓ_1, $y' = y$, and $z' = z$. Assertion φ_2 implies $y \leq x \wedge z = x!/y!$ which together with $y > 0$ establishes φ_1'. The rank decrease $(y,2) \succ (y,1)$ is obvious.

- Premises JW2, JW3 for $i = 1$

For $i = 1$ we will show

$$\rho_\tau \ \wedge \ \underbrace{at\_\ell_1 \ \wedge \ 1 \leq y \leq x \ \wedge \ z = x!/y!}_{\varphi_1} \ \rightarrow$$

$$\cdots \ \vee \ \Big(\underbrace{at'\_\ell_0 \ \wedge \ 0 \leq y' \leq x \ \wedge \ z' = x!/y'!}_{\varphi_2'} \ \wedge \ \underbrace{(y,1)}_{\delta_1} \succ \underbrace{(y',2)}_{\delta_2'} \Big) \ ,$$

for each transition $\tau \in \{\ell_0, \ell_1\}$, not necessarily the helpful one. Obviously, this will satisfy both JW2 and JW3. Since at_ℓ_1 implies that transition ℓ_0 is disabled, the left-hand side of the implication for $\tau = \ell_0$ is false and the implication is trivially true.

For $\tau = \ell_1$, we observe that ρ_{ℓ_1} implies at'_ℓ_0, $y' = y - 1$, and $z' = z \cdot y$. Substituting these expressions in the right-hand side of the implication reduces the conjunction to

$$0 \leq y - 1 \leq x \quad \wedge \quad z \cdot y = x!/(y-1)! \quad \wedge \quad (y,1) \succ (y-1,2),$$

all of which are either obviously valid or are implied by φ_1.

- Premises JW4

The helpful transitions for φ_1 and φ_2 are ℓ_1 and ℓ_0, with enabling conditions at_ℓ_1 and at_ℓ_0, respectively. Obviously, they satisfy

$$\underbrace{at\_\ell_1 \ \wedge \ \cdots}_{\varphi_1} \ \rightarrow \ \underbrace{at\_\ell_1}_{En(\tau_1)}$$

$$\underbrace{at\_\ell_0 \ \wedge \ \cdots}_{\varphi_2} \ \rightarrow \ \underbrace{at\_\ell_0}_{En(\tau_2)}$$

as required by premise JW4.

This establishes the four premises of rule WELL-J, proving the response property of total correctness for program FACT

$$at\_\ell_0 \ \wedge \ x \geq 0 \ \wedge \ y = x \ \wedge \ z = 1 \quad \Rightarrow \quad \Diamond(at\_\ell_2 \ \wedge \ z = x!). \ \blacksquare$$

A Condensed Representation of Ranking Functions

Many of our proofs consider ranking functions that consist of lexicographic pairs of natural numbers, i.e.,

$$\delta \;=\; (d_1, d_2).$$

Such a function decreases over a transition even if d_2 increases, provided d_1 decreases at the same time. Lexicographic order implies that even a small decrease in d_1 outweights an arbitrarily large increase in d_2. In some cases there exists a bound M, $M > 0$, which is larger than any possible increase in d_2. In these cases we may use the ranking function

$$\widehat{\delta} \;=\; M \cdot d_1 + d_2,$$

which ranges over $\mathbb{N}$, instead of the original $\delta = (d_1, d_2)$ which ranges over $\mathbb{N} \times \mathbb{N}$. We refer to $\widehat{\delta}$ as a *condensed representation* ranking function.

In **Problem 1.4** the reader is requested to prove that in such cases,

$$\widehat{\delta} \;=\; M \cdot d_1 + d_2 \;>\; \widehat{\delta}\,' \;=\; M \cdot d_1' + d_2' \qquad \text{iff} \qquad \delta\colon\ (d_1, d_2) \;\succ\; \delta'\colon\ (d_1', d_2').$$

For example, in the above proof of program FACT, we used the ranking functions

$$\delta_i\colon \quad (|y|,\ i).$$

Since the maximal increase in the value of i is 1 which is smaller than 2, we could have used instead the condensed ranking function

$$\widehat{\delta}_i\colon \quad 2 \cdot |y| \;+\; i.$$

Example (up down)

Consider program UP-DOWN presented in Fig. 1.27. This program can be viewed as an extension of program ANY-Y of Fig. 1.2. Process P_1 increments y, counting up in ℓ_0, ℓ_1, as long as $x = 0$. Once P_1 finds that x is different from 0, it proceeds to ℓ_2, ℓ_3, where y is decremented until it becomes 0. Process P_2's single action is to set x to 1. Obviously, due to justice, x will eventually be set to 1. However, one cannot predict the number of helpful steps required for P_1 to terminate. The longer P_2 waits before performing m_0, the higher the value y will attain on the move to ℓ_2. It is this value of y which determines the number of remaining steps to termination.

In fact, for every $n > 0$, we can construct a computation requiring more than $4n$ helpful steps to achieve $y = 0$. This computation allows P_1 to increase y up to n, and only then activates m_0. At least $2n$ more steps of P_1 are needed to decrement y back to 0.

Consequently, we need rule WELL-J to prove the response property

$$\textbf{local } x,\ y\text{: } \textbf{integer where } x = y = 0$$

$$P_1 :: \begin{bmatrix} \ell_0\text{: } \textbf{while } x = 0 \textbf{ do} \\ \quad \ell_1\text{: } y := y + 1 \\ \ell_2\text{: } \textbf{while } y > 0 \textbf{ do} \\ \quad \ell_3\text{: } y := y - 1 \\ \ell_4\text{:} \end{bmatrix} \quad \| \quad P_2 :: \begin{bmatrix} m_0\text{: } x := 1 \\ m_1\text{:} \end{bmatrix}$$

Fig. 1.27. Program UP-DOWN.

$$\underbrace{at\_\ell_0 \ \wedge \ at\_m_0 \ \wedge \ x = y = 0}_{p=\Theta} \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_4 \ \wedge \ at\_m_1}_{q}$$

for program UP-DOWN.

In order to construct the intermediate assertions and the ranking functions δ_i, we observe that there are three distinct phases in the achievement of $at\_\ell_4 \wedge at\_m_1$. The first phase waits for P_2 to perform m_0. This phase terminates when m_0 is executed. In the second phase, P_1 senses that x has been set to 1 and moves to ℓ_2. In the third phase, P_1 is within $\ell_{2,3}$ and decrements y until y reaches 0 and P_1 moves to ℓ_4.

Consequently, it seems advisable to use the well-founded domain $(\mathbb{N}^3, \succ)$ of lexicographic triples (n_1, n_2, n_3), whose first element n_1 identifies the phase, and whose remaining elements, n_2 and n_3, identify progress within the phase. Recall that lexicographic ordering on triples of natural numbers is defined by

$$(n_1,\ n_2,\ n_3) \ \succ \ (m_1,\ m_2,\ m_3) \qquad \text{iff} \qquad \begin{cases} n_1 > m_1 & \text{or} \\ n_1 = m_1,\ n_2 > m_2 & \text{or} \\ n_1 = m_1,\ n_2 = m_2,\ n_3 > m_3. \end{cases}$$

Obviously, this ordering is a well-founded relation on $\mathbb{N}^3$.

Consider the remaining elements needed to measure progress within a phase. The first phase terminates after one helpful step, m_0. The second phase terminates in two helpful steps, ℓ_1 followed by ℓ_0. The last phase has y measuring coarse progress, and it takes two steps to decrement y, ℓ_2 followed by ℓ_3.

Consequently, we define the following assertions, helpful transitions and ranking functions,

$$\begin{array}{llll} \varphi_5\text{:} & at\_\ell_{0,1} \wedge at\_m_0 \wedge x = 0 \wedge y \geq 0 & \tau_5\text{:} \ \ m_0 & \delta_5\text{:} \ \ (2,\ 0,\ 0) \\ \varphi_4\text{:} & at\_\ell_1 \ \wedge at\_m_1 \wedge x = 1 \wedge y \geq 0 & \tau_4\text{:} \ \ \ell_1 & \delta_4\text{:} \ \ (1,\ 0,\ 1) \end{array}$$

$$\begin{array}{llllll}
\varphi_3\colon & at_\ell_0 & \wedge\ at_m_1 \wedge x = 1 \wedge y \geq 0 & \tau_3\colon & \ell_0 & \delta_3\colon\ (1,\ 0,\ 0) \\
\varphi_2\colon & at_\ell_2 & \wedge\ at_m_1 \wedge x = 1 \wedge y \geq 0 & \tau_2\colon & \ell_2 & \delta_2\colon\ (0,\ |y|,\ 2) \\
\varphi_1\colon & at_\ell_3 & \wedge\ at_m_1 \wedge x = 1 \wedge y > 0 & \tau_1\colon & \ell_3 & \delta_1\colon\ (0,\ |y|,\ 1) \\
\varphi_0\colon & at_\ell_4 & \wedge\ at_m_1 & & & \delta_0\colon\ (0,\ 0,\ 0).
\end{array}$$

Note that progress within the second phase is measured by the third component, which moves from 1 to 0 on execution of ℓ_1. Progress within the third phase is measured by the pair $(y, 1 + at\_\ell_2)$ which decreases on execution of both ℓ_3 and ℓ_2.

⟪ Exercise?⟫

Let us show that all premises of rule WELL-J are satisfied by these choices.

- Premise JW1

For this premise we have to show the implication

$$\underbrace{at\_\ell_0\ \wedge\ at\_m_0\ \wedge\ x = y = 0}_{p}\ \rightarrow$$

$$\cdots\ \vee\ \underbrace{at\_\ell_{0,1}\ \wedge\ at\_m_0\ \wedge\ x = 0\ \wedge\ y \geq 0}_{\varphi_5},$$

which is obvious.

- Premise JW2 for φ_5

It is sufficient to show the following for each $\tau \neq m_0$

$$\rho_\tau\ \wedge\ \underbrace{at\_\ell_{0,1}\ \wedge\ at\_m_0\ \wedge\ x = 0\ \wedge\ y \geq 0}_{\varphi_5}\ \rightarrow$$

$$\cdots\ \vee\ \Big(\underbrace{at'\_\ell_{0,1}\ \wedge\ at'\_m_0\ \wedge\ x' = 0\ \wedge\ y' \geq 0}_{\varphi'_5}\ \wedge\ \underbrace{(2,\ 0,\ 0)}_{\delta_5}\ =\ \underbrace{(2,\ 0,\ 0)}_{\delta'_5}\Big)\ .$$

The only transitions $\tau \neq m_0$ enabled on φ_5-states are ℓ_0 and ℓ_1. For each of them, ρ_τ implies $at'\_\ell_{0,1}$, $x' = x = 0$, and $y' \geq y \geq 0$. Consequently, the implication is valid.

- Premise JW3 for φ_5

We show

$$\rho_{m_0}\ \wedge\ \underbrace{at\_\ell_{0,1}\ \wedge\ at\_m_0\ \wedge\ x = 0\ \wedge\ y \geq 0}_{\varphi_5}\ \rightarrow$$

$$\left[\begin{array}{l} \underbrace{at'\_\ell_1 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_4} \ \wedge \ \underbrace{(2,\ 0,\ 0)}_{\delta_5} \succ \underbrace{(1,\ 0,\ 1)}_{\delta'_4} \\ \qquad \vee \\ \underbrace{at'\_\ell_0 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_3} \ \wedge \ \underbrace{(2,\ 0,\ 0)}_{\delta_5} \succ \underbrace{(1,\ 0,\ 0)}_{\delta'_3} \end{array}\right].$$

Clearly, ρ_{m_0} implies $x' = 1$, $y' = y$, at'_m_1 and $at'\_\ell_i = at\_\ell_i$ for $i = 0, 1$. By φ_5, either at_ℓ_0 or at_ℓ_1 holds. In the case that $at\_\ell_0 = \text{T}$, the second disjunct is implied. In the case that $at\_\ell_1 = \text{T}$, the first disjunct is implied. The decrease in rank is obvious in both cases.

- Premises JW2, JW3 for φ_4

Since ℓ_1 is the only transition enabled on φ_4-states, the following implication establishes both JW2 and JW3 for φ_4:

$$\rho_{\ell_1} \ \wedge \ \underbrace{at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y \geq 0}_{\varphi_4} \ \rightarrow$$

$$\cdots \ \vee \ \left(\underbrace{at'\_\ell_0 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_3} \ \wedge \ \underbrace{(1,\ 0,\ 1)}_{\delta_4} \succ \underbrace{(1,\ 0,\ 0)}_{\delta'_3}\right).$$

Transition relation ρ_{ℓ_1} implies at'_ℓ_0, $at'\_m_1 = at\_m_1$, $x' = x$, and $y' \geq y$. By φ_4, it follows that φ'_3 holds, and the decrease in rank is obvious.

- Premises JW2, JW3 for φ_3

Since ℓ_0 is the only transition enabled on φ_3-states, the following implication establishes both JW2 and JW3 for φ_3

$$\rho_{\ell_0} \ \wedge \ \underbrace{at\_\ell_0 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y \geq 0}_{\varphi_3} \ \rightarrow$$

$$\cdots \ \vee \ \left(\underbrace{at'\_\ell_2 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_2} \ \wedge \ \underbrace{(1,\ 0,\ 0)}_{\delta_3} \succ \underbrace{(0,\ |y'|,\ 2)}_{\delta'_2}\right)$$

Transition relation ρ_{ℓ_0} under $x = 1$ implies at'_ℓ_2 and $at'\_m_1 = at\_m_1$. It also implies $x' = x$ and $y' = y$. The rank decrease $(1, 0, 0) \succ (0, |y'|, 2)$ is obvious, since 1, the first component of the left-hand side, is larger than 0, the first component of the right-hand side.

- Premises JW2, JW3 for φ_2

Since ℓ_2 is the only transition enabled on φ_2-states, the following implication establishes both JW2 and JW3 for φ_2

$$\rho_{\ell_2} \ \wedge \ \underbrace{at\_\ell_2 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y \geq 0}_{\varphi_2} \ \rightarrow$$

$$\left(\begin{array}{l} \underbrace{at'\_\ell_4 \ \wedge \ at'\_m_1}_{\varphi'_0} \ \wedge \ \underbrace{(0, \ |y|, \ 2)}_{\delta_2} \succ \underbrace{(0, \ 0, \ 0)}_{\delta'_0} \\ \quad \vee \\ \underbrace{at'\_\ell_3 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' > 0}_{\varphi'_1} \ \wedge \ \underbrace{(0, \ |y|, \ 2)}_{\delta_2} \succ \underbrace{(0, \ |y'|, \ 1)}_{\delta'_1} \end{array} \right).$$

We distinguish between two cases.

Case $y = 0$:

In this case, ρ_{ℓ_2} implies at'_ℓ_4, $y' = y = 0$, and $at'\_m_1 = at\_m_1$. Since φ_2 implies $at\_m_1 = \mathrm{T}$, the left-hand side of this verification condition implies the right-hand side disjunct $\varphi'_0 \ \wedge \ (0, |y|, 2) \succ (0, 0, 0)$.

Case $y \neq 0$:

By φ_2, it follows that $y > 0$. In this case, ρ_{ℓ_2} implies at'_ℓ_3, $at'\_m_1 = at\_m_1$, $x' = x$, and $y' = y > 0$. Together with φ_2, these imply φ'_1. To show the rank decrease, we observe that $|y| = |y'|$ and $2 > 1$.

- Premises JW2, JW3 for φ_1

Since ℓ_3 is the only transition enabled on φ_1-states, the following implication establishes both JW2 and JW3 for φ_1

$$\rho_{\ell_3} \ \wedge \ \underbrace{at\_\ell_3 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y > 1}_{\varphi_1} \ \rightarrow$$

$$\cdots \ \vee \ \left(\underbrace{at'\_\ell_2 \ \wedge \ at'\_m_1 \ \wedge \ x' = 1 \ \wedge \ y' \geq 0}_{\varphi'_2} \ \wedge \ \underbrace{(0, \ |y|, \ 1)}_{\delta_1} \succ \underbrace{(0, \ |y'|, \ 2)}_{\delta'_2} \right).$$

Transition relation ρ_{ℓ_3} implies at'_ℓ_2, $at'\_m_1 = at\_m_1$, $x' = x$, and $y' = y - 1$. By the clause $y > 0$ in φ_1 we have $y' = y - 1 \geq 0$. The decrease in rank follows from $y > 0$ and $(0, |y|, 1) = (0, y, 1) \succ (0, y - 1, 2) = (0, |y'|, 2)$.

- Premise JW4

This premise requires showing the following implication for each $i = 1, \ldots, 5$.

$$\varphi_i \ \rightarrow \ En(\tau_i).$$

By inspecting φ_i for each $i = 1, \ldots, 5$, we see that this is indeed the case.

This concludes the proof. ∎

Persistence of the Helpful Transitions

Premise JW2 of rule WELL-J requires that, in the case that a transition does not attain a lower rank in the next state, it must maintain the rank and lead to a state that still satisfies φ_i, and therefore maintain τ_i as the helpful transition. We refer to this clause as a requirement for the persistence of helpful transitions. One may wonder how essential this requirement is, and whether it would be possible to relax this requirement. In **Problem 1.5**, we request the reader to consider a version of rule WELL-J in which premise JW2 has been relaxed to allow the helpful transition to change without rank decrease. The problem shows that the resulting rule is unsound.

1.5 Rank Diagrams

To represent by diagrams proofs of response properties that require the use of well-founded ranking, we have to add some more components to the labels of nodes.

A verification diagram is said to be a RANK *diagram* if its nodes are labeled by assertions $\varphi_0, \ldots, \varphi_m$, with φ_0 being the terminal node, and ranking functions $\delta_0, \ldots, \delta_m$, where each δ_i maps states into $\mathcal{A}$, and it satisfies the following requirement:

- Every node φ_i, $i > 0$, has a double edge departing from it. This identifies the transition labeling such an edge as *helpful* for assertion φ_i. All helpful transitions must be just.

Note that, unlike CHAIN diagrams, we allow node φ_i to be connected to φ_j for $j > i$.

Verification and Enabling Conditions for RANK Diagrams

Consider a nonterminal node labeled by assertion φ and ranking function δ, and let $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the τ-successors of φ and $\delta_1, \ldots, \delta_k$ be their respective ranking functions.

- If transition τ is unhelpful for φ, i.e., labels only single edges departing from the node, then we associate with φ and τ the following verification condition

$$\Big\{\varphi \wedge \delta = u\Big\} \;\; \tau \;\; \Big\{(\varphi \wedge u \succcurlyeq \delta) \;\vee\; (\varphi_1 \wedge u \succ \delta_1) \;\vee\; \cdots \vee\; (\varphi_k \wedge u \succ \delta_k)\Big\}.$$

- If τ is helpful for φ (labels double edges), we associate with φ and τ the following verification condition

$$\Big\{\varphi \wedge \delta = u\Big\} \;\; \tau \;\; \Big\{(\varphi_1 \wedge u \succ \delta_1) \;\vee\; \cdots \vee\; (\varphi_k \wedge u \succ \delta_k)\Big\}.$$

- For every nonterminal node φ and a transition τ labeling a double edge departing from φ, we require

$$\varphi \quad \rightarrow \quad En(\tau).$$

Note that in the case of an unhelpful transition, we allow a τ-successor with a rank equal to that of φ, provided it satisfies the same assertion φ.

Valid RANK Diagrams

A RANK diagram is said to be *valid over program P* (*P-valid* for short) if all the verification and enabling conditions associated with the diagram are P-state valid.

The consequences of having a valid RANK diagram are stated in the following claim.

Claim 1.2 (RANK diagrams)

A P-valid RANK diagram establishes that the response formula

$$\bigvee_{j=0}^{m} \varphi_j \quad \Rightarrow \quad \Diamond \varphi_0$$

is P-valid.

If, in addition, we can establish the P-state validity of the following implications:

$$p \rightarrow \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \varphi_0 \rightarrow q$$

then, we can conclude the validity of

$$p \quad \Rightarrow \quad \Diamond q.$$

Justification It is not difficult to see that a valid RANK diagram establishes the premises of rule WELL-J with p: $\bigvee_{j=0}^{m} \varphi_j$, q: φ_0, and τ_i the transition helpful for φ_i being the transition labeling the double edge departing from φ_i in the diagram. This establishes the P-validity of

$$\bigvee_{j=0}^{m} \varphi_j \;\Rightarrow\; \Diamond \varphi_0.$$

Given assertions p and q, satisfying the implications

$$p \rightarrow \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \varphi_0 \rightarrow q,$$

we can use rule MON-R of Fig. 1.3 to infer

$$p \Rightarrow \Diamond q. \quad \blacksquare$$

Example (factorial)

The diagram of Fig. 1.28 presents a valid RANK diagram that establishes total correctness for program FACT (Fig. 1.24).

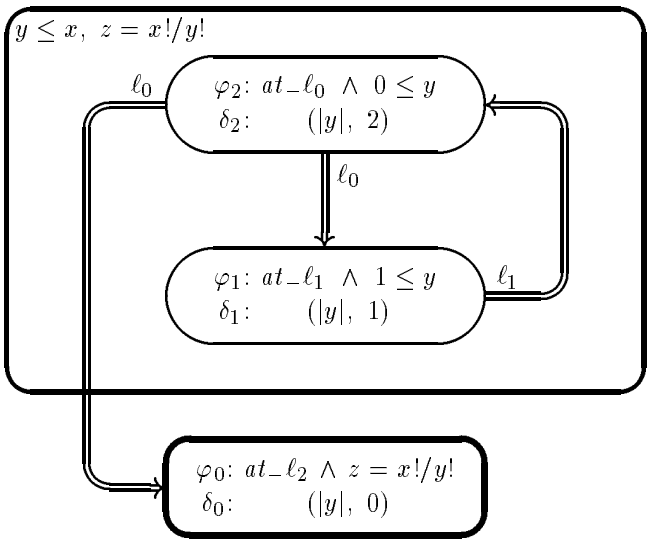


Fig. 1.28. RANK diagram for total correctness of program FACT.

This diagram contains a connection from φ_1 to φ_2 which is disallowed in chain diagrams. However, the validity of the implied verification conditions ensures that, whenever a transition is taken from a φ_2-state s_2 to a φ_1-state s_1, the rank decreases, i.e., $\delta_2(s_2) \succ \delta_1(s_1)$. $\blacksquare$

Distributing the Ranking Functions

To make RANK diagrams more readable, we introduce additional encapsulation conventions.

One of the useful conventions is that compound nodes may be labeled by a list of assertions. Such labeling indicates that the full assertion associated with a basic (noncompound) node n_i is a conjunction of the assertion labeling the node itself and all the assertions labeling compound nodes that contain n_i.

Thus, while the label of node φ_2 in the diagram of Fig. 1.28 is $at\_\ell_0 \wedge 0 \leq y$, the full assertion associated with this node is

$$at\_\ell_0 \ \wedge \ 0 \leq y \ \wedge \ y \leq x \ \wedge \ z = x!/y! \ .$$

We can view this representation as distribution of the full assertion into the part $at\_\ell_0 \wedge 0 \leq y$ labeling the node itself and the part $y \leq x \wedge z = x!/y!$ labeling the enclosing node, which is common to both φ_1 and φ_2.

In a similar way, we introduce a convention for distribution of ranking functions. The convention allows us to label a compound node by

$$\delta\colon f,$$

where f is some ranking function mapping states into a well-founded domain $\mathcal{A}$. In most of our examples, the domains are either $(\mathbb{N}, >)$ or lexicographic products of this domain.

Consider a basic node n_i labeled by assertion φ_i and local ranking function f_b. Assume that node n_i is contained in a nested sequence of compound nodes that are labeled by ranking labels $\delta\colon f_1, \ldots, \delta\colon f_m$, as we go from the outermost compound node towards n_i. This situation is depicted in Fig. 1.29. Then the full ranking function associated with the node φ_i is given by the tuple

$$\delta_i \ = \ (f_1, \ \ldots, \ f_m, \ f_b).$$

That is, we consider the outermost ranking f_1 to be the most significant component in δ_i, and the local ranking f_b to be the least significant component.

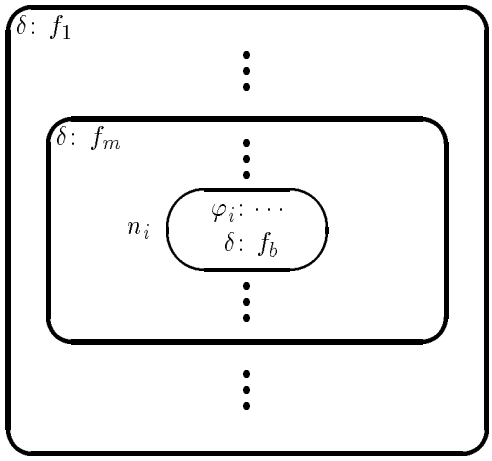


Fig. 1.29. Encapsulated sequence of nodes.

Example (factorial)

In Fig. 1.30, we present a version of the RANK diagram of Fig. 1.28, in which a common component of the ranking function appears as a ranking label of the enclosing compound state.

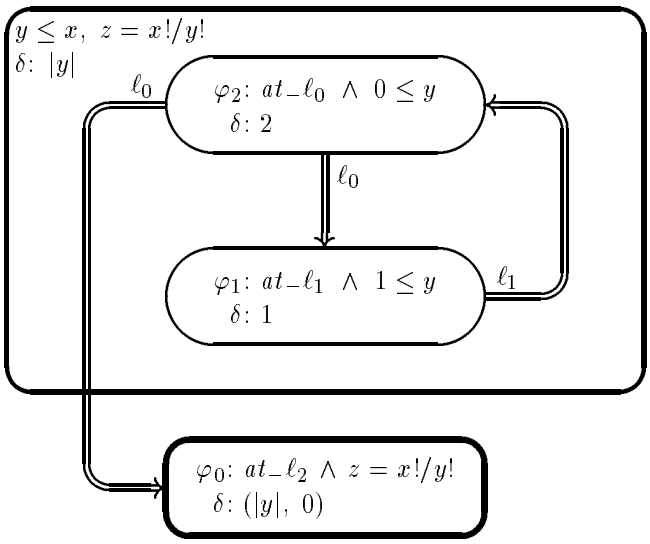


Fig. 1.30. RANK diagram with distributed ranking functions.

The full ranking functions associated with the nodes in the RANK diagram of Fig. 1.30 are identical to those appearing in Fig. 1.28. ∎

Another rank distribution convention allows one to omit the local rank labeling a node φ_i altogether. This is interpreted as if the node were labeled with the ranking function δ: i, where i is the index of the node (and the assertion labeling it). In Fig. 1.31, we present another version of the RANK diagram for program FACT, using this convention.

The full ranking functions associated with the nodes in this diagram are:

$$\delta_2\colon\ (|y|,\ 2), \qquad \delta_1\colon\ (|y|,\ 1), \qquad \text{and} \qquad \delta_0\colon\ 0.$$

This raises the question of how to compare lexicographic tuples of unequal lengths such as δ_2: $(|y|,\ 2)$ and δ_0: 0.

Since all our examples will be based on tuples of non-negative integers, we agree that the relation holding between $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_k)$ for $i < k$ is determined by lexicographically comparing $(a_1, \ldots, a_i, 0, \ldots, 0)$ to $(b_1, \ldots, b_i, b_{i+1},$

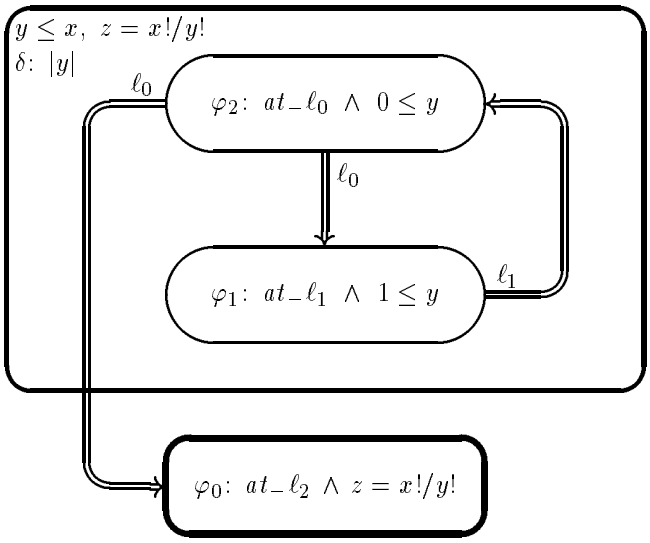


Fig. 1.31. RANK diagram with default local ranking.

$\ldots, b_k)$. That is, we pad the shorter tuple by zeros on the right until it assumes the length of the longer tuple.

According to this definition, $(|y|,\ 2) \succ 0$, since $(|y|,\ 2) \succ (0,\ 0)$.

Example (up-down)

In Fig. 1.32 we present a valid RANK diagram which implies, by monotonicity, the property of termination for program UP-DOWN (Fig. 1.27).

$$\underbrace{at\_\ell_0 \ \wedge\ at\_m_0 \ \wedge\ x = y = 0}_{p=\Theta} \ \Rightarrow\ \diamondsuit \underbrace{at\_\ell_4 \ \wedge\ at\_m_1}_{\varphi_0} \ .$$

The diagram provides a detailed description of the progress of the computation from φ_5 to φ_0. It shows that progress from φ_5 to φ_2 is due to a CHAIN-like reasoning. Then, the progress from φ_2 and φ_1 to φ_0 requires a well-founded argument with the measure $|y|$ for coarse progress, and the index $j = 1, 2$ of φ_j for measuring fine progress.

The ranking functions appearing in this diagram are somewhat different from the ones used originally. When padded to the maximum length of 3, they are given by

δ_5: $(5,\ 0,\ 0)$

δ_4: $(4,\ 0,\ 0)$

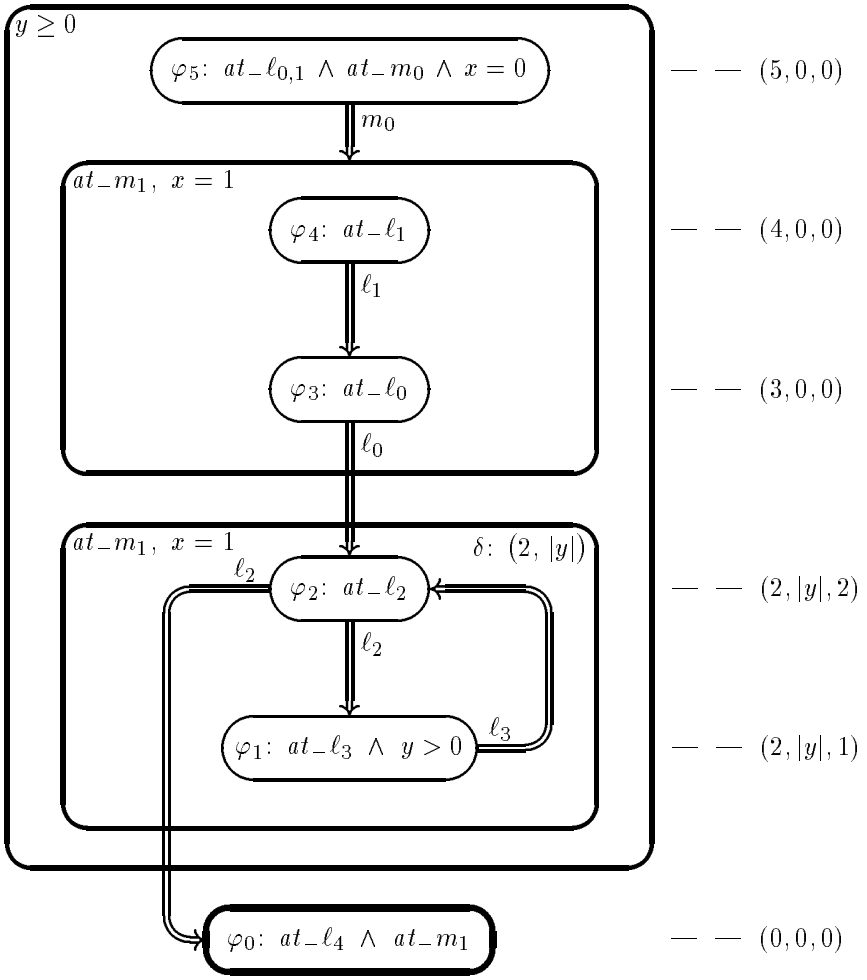


Fig. 1.32. RANK diagram for termination of UP-DOWN.

δ_3: $(3,\ 0,\ 0)$

δ_2: $(2,\ |y|,\ 2)$

δ_1: $(2,\ |y|,\ 1)$

δ_0: $(0,\ 0,\ 0)$.

Note that the diagram contains a connection from φ_1 to φ_2. This is allowed because ℓ_3 decrements y and leads to a decrease in rank, stated by

$$(2,\ |y|,\ 1) \succ (2,\ |y-1|,\ 2).$$ ∎

In **Problems 1.6–1.9**, the reader is requested to prove total correctness of several programs.

1.6 Response with Past Subformulas

In this section we generalize the methods and proof rules presented in the preceding sections to handle response formulas $p \Rightarrow \Diamond q$, where p and q are past formulas.

The generalization is straightforward. It involves the following systematic modifications and replacements.

- Wherever a rule calls for one or more intermediate assertions, the past-version of the rule requires finding past formulas.
- Each premise of the form $\varphi \rightarrow \psi$, for assertions φ and ψ, is replaced by an entailment $\widehat{\varphi} \Rightarrow \widehat{\psi}$ for corresponding past formulas $\widehat{\varphi}$ and $\widehat{\psi}$.
- A verification condition $\{p\}\,\tau\,\{q\}$, for past formulas p and q and transition τ, is interpreted as the entailment $\rho_\tau \wedge p \Rightarrow q'$, where the primed version of a past formula is calculated as in Section 4.1 of the SAFETY book.

For example, in Fig. 1.33, we present the past version of rule WELL-J.

Similar past versions can be derived for rules RESP-J and CHAIN-J.

Example Let us illustrate the use of the past version of rule WELL-J for proving the response property

$$0 \le n \le y \quad \Rightarrow \quad \Diamond\big(y = n \ \wedge\ \diamondminus(at\_\ell_0 \ \wedge\ y = n)\big)$$

for program UP-DOWN of Fig. 1.27.

This property states that any position i at which y is greater or equal to some $n \ge 0$, is followed by a position j at which $y = n$ and such that, at a preceding position $k \le j$, y equaled n while control was at ℓ_0. This property characterizes a feature of program UP-DOWN by which a computation that achieves $y \ge n$ at some state, has at least two occurrences of states in which $y = n$. One occurrence has control at ℓ_0 while the other occurrence has control at ℓ_2.

In our proof, we use the following invariants for program UP-DOWN

$$\begin{aligned}
&\chi_0\colon && y \ge 0\\
&\chi_1\colon && (at_\ell_{0,1} \ \wedge\ at_m_0 \ \wedge\ x = 0) \quad \vee \quad (at_\ell_{0..4} \ \wedge\ at_m_1 \ \wedge\ x = 1)\\
&\chi_2\colon && at_\ell_4 \ \rightarrow\ y = 0\\
&\chi_3\colon && y < n \ \vee\ \diamondminus(at_\ell_0 \ \wedge\ y = n).
\end{aligned}$$

For past formulas p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J}$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m$: $\Sigma \mapsto \mathcal{A}$

JW1. $p \;\Rightarrow\; \bigvee_{j=0}^{m} \varphi_j$

JW2. $\rho_\tau \wedge \varphi_i \;\Rightarrow\; \left[\begin{array}{l} \bigvee_{j=0}^{m} (\varphi'_j \wedge \delta_i \succ \delta'_j) \\ \vee\ (\varphi'_i \wedge \delta_i = \delta'_i) \end{array}\right]$ for every $\tau \in \mathcal{T}$ } for $i = 1, \ldots, m$

JW3. $\rho_{\tau_i} \wedge \varphi_i \;\Rightarrow\; \bigvee_{j=0}^{m} (\varphi'_j \wedge \delta_j \succ \delta'_j)$

JW4. $\varphi_i \;\Rightarrow\; En(\tau_i)$

$p \;\Rightarrow\; \diamondsuit q$

Fig. 1.33. Past version of rule WELL-J.

State invariants χ_0, χ_1, and χ_2 are derived and proven in the usual way.

Proving the Invariance of χ_3

Formula χ_3 is a past invariant, and can be proven by rule P-INV, taking $\varphi = \psi$: $y < n \vee \diamondminus(at\_\ell_0 \wedge y = n)$. Premise P1 of rule P-INV is trivial since $\varphi = \psi$. Premise P2 requires

$$\underbrace{\cdots at\_\ell_0 \ \wedge\ y = 0 \ \wedge\ \cdots}_{\Theta} \ \rightarrow\ \underbrace{y < n \ \vee\ (at\_\ell_0 \ \wedge\ y = n)}_{\varphi_0} .$$

As $n \geq 0$, we consider two cases. If $n > 0$ then $y = 0$ implies $y < n$. If $n = 0$ then $at\_\ell_0 \wedge y = 0$ implies $at\_\ell_0 \wedge y = n$.

Finally, premise P2 requires showing

$$\rho_\tau \ \wedge\ \underbrace{y < n \ \vee\ \diamondminus(at\_\ell_0 \ \wedge\ y = n)}_{\varphi} \ \Rightarrow$$

$$\underbrace{y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n) \ \vee \ \diamondminus(at\_\ell_0 \ \wedge \ y = n)}_{\varphi'} \ .$$

This can be shown by temporal instantiation of the implication

$$\rho_\tau \ \wedge \ (y < n \ \vee \ p) \ \rightarrow \ \big(y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n) \ \vee \ p\big) .$$

Obviously if $p = \mathrm{T}$ the implication is trivially valid. It therefore remains to show that the following holds:

$$\rho_\tau \ \wedge \ y < n \ \rightarrow \ y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n).$$

This implication can be potentially falsified only by a transition that can transform a state satisfying $y < n$ into a next state satisfying $\neg(y' < n)$, i.e., $y' \geq n$. The only candidate transition is ℓ_1. Therefore, we consider

$$\underbrace{\cdots \ \wedge \ at'\_\ell_0 \ \wedge \ y' = y + 1}_{\rho_\tau} \ \wedge \ y < n \ \rightarrow \ y' < n \ \vee \ (at'\_\ell_0 \ \wedge \ y' = n).$$

As $y < n$, we consider two cases. If $y < n - 1$ then $y' = y + 1 < n$. If $y = n - 1$ then $at'\_\ell_0 \wedge y' = y + 1$ implies $at'\_\ell_0 \wedge y' = n$.

This concludes the proof of past invariant χ_3.

Proving the Response Formula

To prove the response formula

$$\underbrace{0 \leq n \leq y}_{p} \ \Rightarrow \ \diamondsuit \underbrace{y = n \ \wedge \ \diamondminus(at\_\ell_0 \ \wedge \ y = n)}_{q} ,$$

we use the past version of rule WELL-J as presented in Fig. 1.33.

The choice of intermediate past formulas φ_1–φ_5, helpful transitions and ranking functions is presented in the verification diagram of Fig. 1.34.

Note that each of $\varphi_0, \ldots, \varphi_5$ is a past formula. For example, the full formula φ_4 is given by

$$\varphi_4\colon \quad at\_\ell_1 \ \wedge \ at\_m_1 \ \wedge \ x = 1 \ \wedge \ y > n \geq 0 \ \wedge \ \diamondminus(at\_\ell_0 \ \wedge \ y = n).$$

Let us consider some of the premises required by rule WELL-J. Premise JW1 requires the following implication

$$0 \leq n \leq y \ \Rightarrow$$

$$\underbrace{\diamondminus(at\_\ell_0 \ \wedge \ y = n) \ \wedge \ \left(y = n \ \vee \ y > n \wedge \begin{pmatrix} at\_\ell_{0,1} \ \wedge \ at\_m_0 \ \wedge \ x = 0 \\ \vee \\ at\_\ell_{0..3} \ \wedge \ at\_m_1 \ \wedge \ x = 1 \end{pmatrix}\right)}_{\varphi_0 \ \vee \ \cdots \ \vee \ \varphi_5} .$$

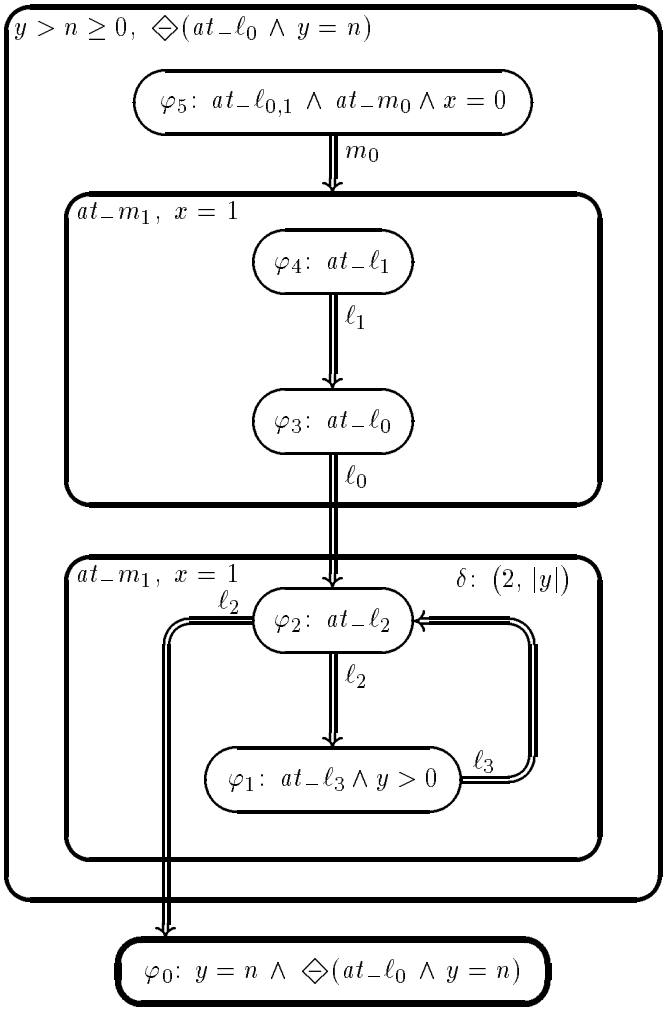


Fig. 1.34. RANK diagram for $(0 \leq n \leq y) \ \Rightarrow\ \diamondsuit\big(y = n \wedge \diamondsuit\!\!-(at\_\ell_0 \wedge y = n)\big)$.

It is not difficult to see that this entailment follows from invariants χ_0–χ_3.

Observe that each φ_i, $i = 0, \ldots, 5$ can be written in the form

$$\varphi_i \ = \ \diamondsuit\!\!-(at\_\ell_0 \ \wedge \ y = n) \ \wedge \ \widehat{\varphi}_i,$$

where $\widehat{\varphi}_i$ is a state formula.

Consequently, premise JW2 can be written as follows

$\rho_\tau \ \wedge \ \widehat{\varphi}_i \ \wedge \ \diamondsuit\!\!-(at\_\ell_0 \ \wedge \ y = n) \ \Rightarrow$

$$\big(\diamondminus(at\_\ell_0 \ \wedge \ y = n)\big)' \ \wedge \ \Big(\widehat{\varphi}'_0 \ \vee \ \bigvee_{j=1}^{5} (\widehat{\varphi}'_j \ \wedge \ \delta_i \succ \delta'_j) \ \vee \ (\widehat{\varphi}_i \ \wedge \ \delta_i = \delta'_i)\Big).$$

Since $\diamondminus(at\_\ell_0 \wedge y = n)$ entails $\big(\diamondminus(at\_\ell_0 \wedge y = n)\big)'$, which expands to the disjunction

$$\big(\diamondminus(at\_\ell_0 \ \wedge \ y = n)\big)': \quad (at'\_\ell_0 \ \wedge \ y' = n) \ \vee \ \diamondminus(at\_\ell_0 \ \wedge \ y = n),$$

it only remains to establish the following state entailment

$$\rho_\tau \ \wedge \ \widehat{\varphi}_i \quad \Rightarrow \quad \widehat{\varphi}'_0 \ \vee \ \bigvee_{j=1}^{5} (\widehat{\varphi}'_j \ \wedge \ \delta_i \succ \delta'_j) \ \vee \ (\widehat{\varphi}_i \ \wedge \ \delta_i = \delta'_i).$$

This entailment can be proven in a way similar to the proof of the verification conditions in the RANK diagram of Fig. 1.32, whose ranking functions are identical to those of Fig. 1.34.

The past conjunct $\diamondminus(at\_\ell_0 \ \wedge \ y = n)$ can be similarly factored out also for premise JW3.

This concludes the proof that property

$$0 \leq n \leq y \quad \Rightarrow \quad \diamondsuit\big(n = y \ \wedge \ \diamondminus(at\_\ell_0 \ \wedge \ y = n)\big)$$

for program UP-DOWN. ∎

1.7 Compositional Verification of Response Properties

Compositional verification is a method intended to reduce the complexity of verifying properties of large programs. The method infers properties of the whole system from properties of its components, which are proven separately for each component.

We apply compositional verification to programs that can be decomposed into several top-level processes, called *components*, which communicate by shared variables and such that every variable of the program can be modified by at most one of these components. A variable which is modified by component P_i is said to be *owned* by P_i.

Modular Computations

Let $P :: \big[\text{declarations}; \ \big[P_1 :: [\ell_0^1 \colon S_1]\| \cdots \|P_k :: [\ell_0^k \colon S_k]\big]\big]$ be a program, and P_i be a component of P. Denote by $V = \{\pi\} \cup Y$ the set of system variables of P, and let $Y_i \subseteq Y$ be the set of variables owned by P_i. Let L_i denote the set of locations of process P_i.

Assume that we have constructed the fair transition system (FTS) S_P: $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ corresponding to P. We assume that the initial condition has the form

$$\Theta: \quad \pi \ = \ \{\ell_0^1, \ldots, \ell_0^k\} \ \wedge \ \bigwedge_{y \in Y} p_y(y),$$

where, for each $y \in Y$, $p_y(y)$ is an assertion constraining the initial values of variable y. Obviously, $p_y(y)$ is derived from one of the *where* clauses in the declarations of variables in P. If there is no *where* clause constaining y, then $p_y(y) = \text{T}$. Let T_i denote the transitions of S_P associated with the statements of P_i.

Based on the FTS S_P and process P_i, we construct a new FTS $S_{P_i}^M$: $\langle V_i, \Theta_i, \mathcal{T}_i, \mathcal{J}_i, \mathcal{C}_i \rangle$ called the *modular* FTS *corresponding to* P_i. The FTS $S_{P_i}^M$ is intended to capture the possible behavior of process P_i in any context (not necessarily that of P) which respects the ownership of Y_i by P_i. That is, we are ready to consider any context whose only restriction is that it cannot modify any variable owned by P_i. The constituents of $S_{P_i}^M$ ar given by:

- V_i: V

  The system variables of $S_{P_i}^M$ are identical to the system variables of the complete FTS S_P.

- Θ_i: $(\pi \ \cap \ L_i \ = \ \{\ell_0^i\}) \quad \wedge \quad \bigwedge_{y \in Y_i} p_y(y)$

  The initial condition of $S_{P_i}^M$ requires that, initially, the only L_i-location contained in π is ℓ_0^i and all the variables owned by P_i satisfy their initial constraints as specified in the *where* clauses of the program declarations. Except for π, nothing is required by Θ_i concerning the system variables not owned by P_i.

- $\mathcal{T}_i \ = \ T_i \ \cup \ \{\tau_E\}$

  The transitions of $S_{P_i}^M$ include all transitions associated with statements of P_i (T_i) and a special *environment transition* τ_E. Transition τ_E is intended to represent the actions of an arbitrary context which respects the ownership of Y_i by P_i. For each $\tau \in T_i$, the transition relation $S_{P_i}^M$ associates with τ is ρ_τ, the transition relation S_P associates with τ. The transition relation for τ_E is given by

  $$\rho_E: \quad (\pi' \ \cap \ L_i \ = \ \pi \ \cap \ L_i) \quad \wedge \quad \mathit{press}(Y_i).$$

  This transition relation guarantees the preservation of the L_i-part of π and preservation of the values of all variables owned by P_i. The special treatment of π can be described by saying that, in addition to owning the variables in Y_i, P_i also owns the L_i-part of π (projection of π on L_i).

- $\mathcal{J}_i \;=\; \mathcal{J} \;\cap\; T_i$

  The just transitions of $S^M_{P_i}$ are the just transitions among T_i.

- $\mathcal{C}_i \;=\; \mathcal{C} \;\cap\; T_i$

  The compassionate transitions of $S^M_{P_i}$ are the compassionate transitions among T_i.

There is no need to include the idling transition τ_I in $\mathcal{T}_i$ because the effect of τ_I, a transition that changes no system variable, can be obtained as a special case of τ_E.

We refer to each computation of FTS $S^M_{P_i}$ as a *modular computation* of process P_i. As previously explained, any such computation represents a possible behavior of process P_i when put in an arbitrary context which is only required to respect the ownership rights of P_i.

Example (program KEEPING-UP)

Consider program KEEPING-UP presented in Fig. 1.35. Top-level process P_1 owns variable x and the $\ell_{0..2}$-part of π. We use $\ell_{0..2}$ as abbreviation for $\{\ell_0, \ell_1, \ell_2\}$. We can construct $S^M_{P_1}$, the modular FTS corresponding to process P_1 as follows:

local $x,\ y$: **integer where** $x = y = 0$

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{await } x < y+1 \\ \ell_2\text{: } x := x+1 \end{array}\right] \end{array}\right] \;\Big\|\; P_2 :: \left[\begin{array}{l} m_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{await } y < x+1 \\ m_2\text{: } y := y+1 \end{array}\right] \end{array}\right]$$

Fig. 1.35. Program KEEPING-UP.

- $V_1\colon \{\pi, x, y\}$
- $\Theta_1\colon \big(\pi \;\cap\; \ell_{0..2} \;=\; \{\ell_0\}\big) \;\wedge\; x = 0$
- $\mathcal{T}_1\colon \{\tau_{\ell_0}, \tau_{\ell_1}, \tau_{\ell_2}, \tau_E\}$

  with the following transition relations (after some simplifications):

  $\rho_{\ell_0}\colon\ move(\ell_0, \ell_1) \;\wedge\; pres(x, y)$

  $\rho_{\ell_1}\colon\ move(\ell_1, \ell_2) \;\wedge\; x < y+1 \;\wedge\; pres(x, y)$

  $\rho_{\ell_2}\colon\ move(\ell_2, \ell_0) \;\wedge\; x' = x+1 \;\wedge\; pres(y)$

$$\rho_E\colon\ (\pi' \cap \ell_{0..2} = \pi \cap \ell_{0..2}) \wedge pres(x)$$

In this relations, we used the following abbreviation:

$$move(\ell_i, \ell_j)\colon\ at\_\ell_i \wedge \pi' = (\pi - \{\ell_i\}) \cup \{\ell_j\}$$

- $\mathcal{J}_1\colon\ \{\tau_{\ell_0}, \tau_{\ell_1}, \tau_{\ell_2}\}$
- $\mathcal{C}_1\colon\ \emptyset$

The following is a modular computation of process P_1:

$$\begin{aligned}\widehat{\sigma}\colon\ & \langle \pi\colon\{\ell_0, m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_0} \langle \pi\colon\{\ell_1, m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_1} \\ & \langle \pi\colon\{\ell_2, m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\tau_E} \langle \pi\colon\{\ell_2, m_0\},\ x\colon 0,\ y\colon -1\rangle \xrightarrow{\ell_2} \\ & \langle \pi\colon\{\ell_0, m_0\},\ x\colon 1,\ y\colon -1\rangle \ \cdots\ .\end{aligned}$$

In a similar way, we can construct $S^M_{P_2}$, the modular FTS corresponding to process P_2. ∎

The following claim establishes a connection between computations of the entire program and modular computations of its processes.

Claim 1.3 (computations of programs and modular computations)

Every computation of a program is a modular computation of each of its top-level processes.

Thus, the set of computations of the entire program is a subset of the set of modular computations of each of its top-level processes.

Example Consider, for example, the following computation of program KEEPING-UP

$$\begin{aligned}\sigma\colon\ & \langle \pi\colon\{\ell_0, m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_0} \langle \pi\colon\{\ell_1, m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_1} \\ & \langle \pi\colon\{\ell_2, m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{m_0} \langle \pi\colon\{\ell_2, m_1\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_2} \\ & \langle \pi\colon\{\ell_0, m_1\},\ x\colon 1,\ y\colon 0\rangle \xrightarrow{m_1} \langle \pi\colon\{\ell_0, m_2\},\ x\colon 1,\ y\colon 0\rangle \xrightarrow{m_2} \\ & \langle \pi\colon\{\ell_0, m_0\},\ x\colon 1,\ y\colon 1\rangle \xrightarrow{\ell_0} \langle \pi\colon\{\ell_1, m_0\},\ x\colon 1,\ y\colon 1\rangle \ \cdots\ .\end{aligned}$$

Viewed as a modular computation of process P_1, this computation can be presented as:

$$\begin{aligned}
\sigma_1\colon\ & \langle \pi\colon\{\ell_0,m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_0} \langle \pi\colon\{\ell_1,m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_1} \\
& \langle \pi\colon\{\ell_2,m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\tau_E} \langle \pi\colon\{\ell_2,m_1\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\ell_2} \\
& \langle \pi\colon\{\ell_0,m_1\},\ x\colon 1,\ y\colon 0\rangle \xrightarrow{\tau_E} \langle \pi\colon\{\ell_0,m_2\},\ x\colon 1,\ y\colon 0\rangle \xrightarrow{\tau_E} \\
& \langle \pi\colon\{\ell_0,m_0\},\ x\colon 1,\ y\colon 1\rangle \xrightarrow{\ell_0} \langle \pi\colon\{\ell_1,m_0\},\ x\colon 1,\ y\colon 1\rangle \ \cdots\ .
\end{aligned}$$

Viewed as a modular computation of process P_2, this computation can be presented as:

$$\begin{aligned}
\sigma_2\colon\ & \langle \pi\colon\{\ell_0,m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\tau_E} \langle \pi\colon\{\ell_1,m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\tau_E} \\
& \langle \pi\colon\{\ell_2,m_0\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{m_0} \langle \pi\colon\{\ell_2,m_1\},\ x\colon 0,\ y\colon 0\rangle \xrightarrow{\tau_E} \\
& \langle \pi\colon\{\ell_0,m_1\},\ x\colon 1,\ y\colon 0\rangle \xrightarrow{m_1} \langle \pi\colon\{\ell_0,m_2\},\ x\colon 1,\ y\colon 0\rangle \xrightarrow{m_2} \\
& \langle \pi\colon\{\ell_0,m_0\},\ x\colon 1,\ y\colon 1\rangle \xrightarrow{\tau_E} \langle \pi\colon\{\ell_1,m_0\},\ x\colon 1,\ y\colon 1\rangle \ \cdots\ .
\end{aligned}$$

This illustrates that a computation of a program is a modular computation of each of its top-level processes. ∎

The weak converse of Claim 1.3 is not true. There are modular computations of process P_i which do not correspond to computations of the entire program. This is illustrated by $\widehat{\sigma}$ the previously presented modular computation of process P_1 in program KEEPING-UP. This computation contains a state with $y = -1$ as a τ_E-successor of a state with $y = 0$. No such state can occur in a computation of KEEPING-UP. This shows that the definition of modular computations of process P_i allows more general contexts than the actual context provided by the program containing P_i. The actual context of P_1 within program KEEPING-UP is process P_2 which can never change y from a value of 0 to a value of -1.

On the other hand, the strong converse of Claim 1.3 is true. Let σ be a model (infinite state sequence) such that the interpretation of π is a subset of the locations of P. The valid converse of Claim 1.3 states that if σ is simultaneously a modular computation of every top-level process of P then σ is a computation of P. In **Problem 1.10**, we request the reader to prove this fact.

Modular Validity and a Basic Compositionality Rule

For a top-level process P_i within program P, we say that formula φ is *modularly valid over* P_i, denoted

$$P_i \ \models_m \ \varphi,$$

if φ holds over all modular computations of P_i. For example, the formula $\Box(x \geq x^-)$, stating that x never decreases, is modularly valid over process P_1 of program KEEPING-UP (Fig. 1.42), while $\Box(y \geq y^-)$ is modularly valid over process P_2 of the same program[2].

[2] x^- and y^- denote the values of x and y in the preceding state.

Rule COMP-B, presented in Fig. 1.36, infers the P-validity of a formula φ from the premise that φ is modularly valid over some top-level process of P.

For a program p, P_i a top-level process of P, and φ a temporal formula

$$\frac{P_i \models\!\!\!\!\!\!\underset{m}{} \varphi}{P \models \varphi}$$

Fig. 1.36. Rule COMP-B (basic compositionality).

Soundness

Let P be a program and P_i be a top-level process of P. Assume that formula φ is modularly valid over P_i, i.e. $P_i \models\!\!\!\!\!\!\underset{m}{} \varphi$. This means that φ holds over all modular computations of P_i. By Claim 1.3, every computation of P is also a modular computation of P_i. It follows that all computations of P satisfy φ and, hence, φ is P-valid. ∎

Rule COMP-B can be used to reduce the goal of establishing $P \models \varphi$ into the subgoals of establishing several modular validities (not necessarily of the same formula φ. In **Problem 1.11** the reader is requested to establish this fact.

A Compositional Rule for Safety Properties

In theory, rule COMP-B is adequate for compositional verification of any temporal formula. In practice, however, its application often proves inconvenient and calls for additional temporal reasoning. Therefore, it is advantageous to derive more specific rules, each of which is tailored to deal with temporal formulas of particular classes.

In Fig. 1.37 we present rule COMP-S which can be used for compositional verification of safety formulas.

Premise CS1 states that χ is an invariant of the entire program P. Premise CS2 states that the entailment $\boxminus \chi \Rightarrow p$ is modularly valid over some top-level process P_i. From these two assumptions, the rule infers that p is an invariant of P.

Justification Assume that premises CS1 and CS2 hold and let σ be a computation of program P. By premise CS1, formula χ holds at all positions of σ. Since

For P_i, a top-level process of program P, and past formulas χ, p,

$$\frac{\begin{array}{ll}\text{CS1.} & P \models \Box\chi \\ \text{CS2.} & P_i \models_m \boxminus\chi \Rightarrow p\end{array}}{P \models \Box p}$$

Fig. 1.37. Rule COMP-S (compositional verification of safety properties).

every computation of P is also a modular computation of P_i, premise CS2 implies that the formula $\boxminus\chi \rightarrow p$ holds at all positions of σ.

Consider an arbitrary position $j \geq 0$ of σ. By D1, χ holds at all positions $k \leq j$ and, therefore $\boxminus\chi$ holds at j. By CS2, $\boxminus\chi \rightarrow p$ holds at j and, therefore, so does p.

We conclude that p holds at all positions of σ. ∎

Rule COMP-S is often used in an incremental style. As a first step we take $\chi = \mathrm{T}$ and prove $P_i \models_m \Box p_1$. From this the rule infers

$$P \models \Box p_1 .$$

Next, we take $\chi = p_1$ and prove $P_i \models_m \boxminus p_1 \Rightarrow p_2$. This leads to

$$P \models \Box p_2 ,$$

which may be followed by additional steps.

The advantage of this proof pattern is that in each step we concentrate on proving a modular validity over a single process P_i. If P_i is only a small part of the program, each compositional verification step has to consider only a small fraction of the transitions in the complete program.

We illustrate the use of rule COMP-S on a simple example.

Example (program KEEPING-UP)

Consider program KEEPING-UP presented in Fig. 1.35. Process P_1 in this program repeatedly increments x, provided x does not exceed $y + 1$. In a symmetric way, process P_2 repeatedly increments y, provided y does not exceed $x + 1$.

We wish to prove for this program the invariance of the assertion $|x - y| \leq 1$, i.e.,

$$\Box \underbrace{|x - y| \leq 1}_{p} ,$$

claiming that the difference between x and y never exceeds 1 in absolute value.

We prove this property by compositional verification, using rules INV-P and COMP-S. We first show the P-validities

$$P \models \Box(x \geq x^-) \qquad \text{and} \qquad P \models \Box(y \geq y^-),$$

and then the P-validities

$$P \models \Box(x \leq y + 1) \qquad \text{and} \qquad P \models \Box(y \leq x + 1).$$

The invariants $x \leq y + 1$ and $y \leq x + 1$ imply the desired P-validity

$$P \models \Box(|x - y| \leq 1).$$

For more details of this proof, we refer the reader to Section 4.3 of the SAFETY book. ▟

A Compositional Rule for Response Properties

Next, we present a rule that can support compositional verification of response properties and illustrate its use. This is rule COMP-R, presented in Fig. 1.38.

For P_i, a top-level process of program P, and past formulas χ, p, and q,

$$\frac{\begin{array}{ll}\text{CR1.} & P \models \Box\,\chi \\ \text{CR2.} & P_i \underset{m}{\models} p \Rightarrow \Diamond(q \vee \neg\chi)\end{array}}{P \models p \Rightarrow \Diamond q}$$

Fig. 1.38. Rule COMP-R (compositional verification of response properties).

Justification Assume that premises CR1 and CR2 hold and let σ be a computation of program P. By premise CR1, formula χ holds at all positions of σ. Since every computation of P is also a modular computation of P_i, premise MR2 implies that the formula $p \rightarrow \Diamond(q \vee \neg\chi)$ holds at all positions of σ. Let j be a p-position of σ. By MR2, there exists a position $k \geq j$ such that either q holds at k or χ is false at k. The second alternative is impossible, due to CR1. We conclude that every p-position is followed by a q-position and, therefore, $p \Rightarrow \Diamond q$ is valid over P. ▟

Example (program PING-PONG)

local $x,\ y,\ z$: **integer where** $x = y = z = 0$

$$P_1 :: \begin{bmatrix} \textbf{own out } x,\ z \\ \ell_0\colon\ x := 1 \\ \ell_1\colon\ \textbf{await } y > 0 \\ \ell_2\colon\ z := 1 \\ \ell_3\colon \end{bmatrix} \quad \| \quad P_2 :: \begin{bmatrix} \textbf{own out } y \\ m_0\colon\ \textbf{await } x > 0 \\ m_1\colon\ y := 1 \\ m_2\colon \end{bmatrix}$$

Fig. 1.39. Program PING-PONG.

We illustrate the use of rule COMP-R for compositional verification of response properties on an example. In Fig. 1.39, we present program PING-PONG.

The two processes of this program maintain a coordination protocol. The protocol starts by P_1 setting x to 1 at statement ℓ_0. This is sensed by P_2 at m_0, and is responded to by setting y to 1 at statement m_1. This is sensed by P_1 at ℓ_1, and is responded to by setting z to 1 at ℓ_2.

We wish to establish for this program the response property

$$\Theta \ \Rightarrow\ \diamondsuit(z = 1).$$

We start by proving, using rule P-INV, the modular invariance

$$P_1 \ \models\!\!\!\!\!\!{}_{m}\ \ \square(x \geq x^-).$$

This is easy to prove since the local formula $x \geq x^-$ is inductive over the modular FTS corresponding to process P_1.

By rule COMP-B we can infer

$$P \ \models\ \square(x \geq x^-).$$

In a similar way, we establish $P_2 \models\!\!\!\!\!\!{}_{m}\ \square(y \geq y^-)$, leading to

$$P \ \models\ \square(y \geq y^-).$$

Now, we use rule RESP-J to prove

$$P_1 \ \models\!\!\!\!\!\!{}_{m}\ \ \underbrace{\Theta}_{p} \ \Rightarrow\ \diamondsuit \underbrace{x > 0}_{q}\ .$$

As the intermediate assertion and helpful transition, we take φ: at_ℓ_0 and τ_h: ℓ_0.

Using rule COMP-R with χ: T, we conclude

$$P \ \models\ \ \Theta \ \Rightarrow\ \diamondsuit(x > 0).$$

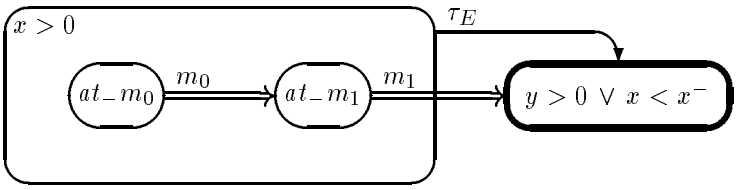
Next, we intend to establish

$$P_2 \models_m x > 0 \Rightarrow \diamondsuit(y > 0 \vee x < x^-).$$

As a first step, we use rule INV to prove

$$P_2 \models_m \square \underbrace{at\_m_{0,1} \vee y > 0}_{\varphi_2}.$$

Having established the modular invariance of φ_2 over P_2, the following verification diagram proves that $x > 0 \Rightarrow \diamondsuit(y > 0 \vee x < x^-)$ is modularly valid over P_2.



Note that the diagram allows the possibility that the environment changes x from a positive value to a nonpositive one. However, such a change leads to a position satisfying $x < x^-$.

Now, use rule COMP-R with χ: $x \geq x^-$, p: $x > 0$, and q: $y > 0$, to conclude

$$P \models x > 0 \Rightarrow \diamondsuit(y > 0).$$

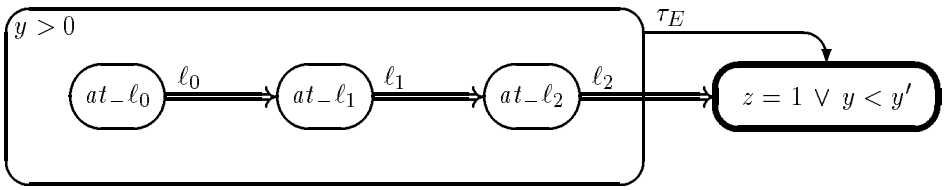
Next, we plan to establish

$$P_1 \models_m y > 0 \Rightarrow \diamondsuit(z = 1 \vee y < y^-).$$

As a first step, we use rule INV to prove

$$P_1 \models_m \square(at\_\ell_{0..2} \vee z = 1).$$

Having established this modular invariant, the following verification diagram proves that $y > 0 \Rightarrow \diamondsuit(z = 1 \vee y < y^-)$ is modularly valid over P_1.



Now, use rule COMP-R with χ: $y \geq y^-$, p: $y > 0$, and q: $z = 1$, to conclude

$$P \models y > 0 \Rightarrow \diamondsuit(z = 1).$$

Thus, we have shown that the three response properties $\Theta \Rightarrow \diamondsuit(x > 0)$, $x > 0 \Rightarrow \diamondsuit(y > 0)$, and $y > 0 \Rightarrow \diamondsuit(z = 1)$ are all valid over program PING-PONG. Using rule TRNS-R, we conclude

$$\Theta \;\Rightarrow\; \diamondsuit(z = 1). \;\blacksquare$$

1.8 Guarantee Properties

In this and the following section we consider methods for proving properties belonging to the *guarantee* and *obligation* classes. Our approach to these classes is to consider them as special cases of the response class, and to use response rules with some simplifications for their verification.

As defined in Section 0.5 of the SAFETY book, guarantee properties are properties that can be specified by a formula of the form

$$\diamondsuit\, r$$

for some past formula r.

Clearly, guarantee properties are a special case of response properties, $p \Rightarrow \diamondsuit\, r$, where the antecedent p refers to the beginning of the computation. Consequently, an obvious rule for proving guarantee properties, is rule GUAR (Fig. 1.40).

For past formula r

$$\frac{\mathit{first} \;\wedge\; \Theta \;\;\Rightarrow\;\; \diamondsuit\, r}{\diamondsuit\, r}$$

Fig. 1.40. Rule GUAR (proving guarantee properties).

The rule requires as a premise a response property by which the initial condition of the program guarantees the eventual realization of r.

Example Consider system INC2, presented in Fig. 1.41.

We wish to establish the guarantee property

$$\diamondsuit\big(x \geq 20 \;\wedge\; \diamondminus(x = 10)\big)$$

for system INC2, using rule GUAR. The premise of rule GUAR requires the response property

$$\overbrace{\mathit{first} \;\wedge\; \underbrace{x = 0}_{\Theta}}^{p} \;\;\Rightarrow\;\; \diamondsuit\Big(\underbrace{x \geq 20 \;\wedge\; \diamondminus(x = 10)}_{r}\Big)\;.$$

This response property can be proven by rule WELL-J, using the intermediate past formula

V: $\{x\colon \textbf{integer}\}$
Θ: $x = 0$
$\mathcal{T}$: $\{\tau_I, \tau\}$ where $\rho_\tau\colon x' = x + 2$
$\mathcal{J}$: $\{\tau\}$
$\mathcal{C}$: $\emptyset$

Fig. 1.41. System INC2.

$$\varphi\colon \quad even(x) \ \wedge\ 0 \le x \le 18 \ \wedge\ \big(x \ge 10 \ \rightarrow\ \diamondminus(x = 10)\big),$$

the helpful transition τ, and the ranking function $\delta\colon |20 - x|$.

Let us establish premise JW1 of rule WELL-J. It requires showing

$$\underbrace{\cdots \wedge\ x = 0}_{p} \ \Rightarrow\ \cdots \ \vee\ \underbrace{even(x) \ \wedge\ 0 \le x \le 18 \ \wedge\ \big(x \ge 10 \ \rightarrow\ \diamondminus(x = 10)\big)}_{\varphi}.$$

Clearly, $x = 0$ entails all three conjuncts comprising φ. Note that, in this case, we do not use the conjunct *first* which is part of p.

By rule GUAR, we conclude that property

$$\diamondsuit\big(x \ge 20 \ \wedge\ \diamondminus(x = 10)\big)$$

is valid over system INC2. ∎

The premise of rule GUAR contains *first* as part of the antecedent. Its purpose is to ensure that we only consider Θ at the beginning of the computation. As illustrated in the last example, in many cases we do not use this conjunct and simply prove $\Theta \Rightarrow \diamondsuit p$. There are, however, some cases in which this conjunct is necessary, as illustrated below.

Example Consider the simple program

$$\textbf{local } x\colon \textbf{integer where } x = 1$$
$$\ell_0\colon \textbf{loop forever do } [\ell_1\colon \textbf{skip};\ \ell_2\colon x := -x].$$

We consider the guarantee property

$$\psi\colon \quad \diamondsuit\big(at\_\ell_2 \ \wedge\ \boxminus(x = 1)\big).$$

This property states that every computation of the program contains a position j satisfying at_ℓ_2, and such that $x = 1$ at all positions $i \le j$.

This property is certainly valid for the program. To prove it, we have to establish the response property

$$\mathit{first} \ \wedge \ at\_\ell_0 \ \wedge \ x = 1 \ \ \Rightarrow \ \ \diamondsuit\big(at\_\ell_2 \ \wedge \ \boxminus(x = 1)\big).$$

Note, however, that the *first* conjunct is essential in this case, since the formula

$$at\_\ell_0 \ \wedge \ x = 1 \ \ \Rightarrow \ \ \diamondsuit\big(at\_\ell_2 \ \wedge \ \boxminus(x = 1)\big)$$

is not valid over the program.

Consider position i in the computation, corresponding to the third visit to ℓ_2. At this position $x = 1$, but there exist earlier positions in which $x = -1$. Therefore, there exists no position later than i, at which $at\_\ell_2 \wedge \boxminus(x = 1)$ holds. It follows that the implication $at\_\ell_0 \wedge x = 1 \rightarrow \diamondsuit\big(at\_\ell_2 \wedge \boxminus(x = 1)\big)$ does not hold at i.

To prove that the full premise

$$\underbrace{\mathit{first} \ \wedge \ at\_\ell_0 \ \wedge \ x = 1}_{p} \ \ \Rightarrow \ \ \diamondsuit\Big(\underbrace{at\_\ell_2 \ \wedge \ \boxminus(x = 1)}_{r}\Big) \ ,$$

is valid, we may use rule CHAIN-J (Fig. 1.7) with the following intermediate past formulas and helpful transitions:

$$\begin{array}{lll} \varphi_2\colon & at\_\ell_0 \ \wedge \ \boxminus(x = 1) & \tau_2\colon \ \ell_0 \\ \varphi_1\colon & at\_\ell_1 \ \wedge \ \boxminus(x = 1) & \tau_1\colon \ \ell_1 \\ \varphi_0 = r\colon & at\_\ell_2 \ \wedge \ \boxminus(x = 1) & \tau_0\colon \ \ell_2. \end{array}$$

Premise J1 requires showing

$$\underbrace{\mathit{first} \ \wedge \ at\_\ell_0 \ \wedge \ x = 1}_{p} \ \ \Rightarrow \ \ \cdots \ \vee \ \underbrace{at\_\ell_0 \ \wedge \ \boxminus(x = 1)}_{\varphi_2} \ .$$

Clearly, the antecedent implies at_ℓ_0. To see that it also implies $\boxminus(x = 1)$, we observe that under *first*, any past formula p is congruent to $(p)_0$, the initial version of p. Since the initial version of $\boxminus(x = 1)$ is

$$\big(\boxminus(x = 1)\big)_0 \ = \ (x = 1),$$

the right-hand side simplifies to $at\_\ell_0 \wedge x = 1$ which is entailed by the left-hand side.

Another premise that has to be checked is J3 for transition ℓ_0,

$$\rho_{\ell_0} \ \ \wedge \ \ \underbrace{at\_\ell_0 \ \wedge \ \boxminus(x = 1)}_{\varphi_2} \ \ \Rightarrow \ \ \cdots \ \vee \ \underbrace{at'\_\ell_1 \ \wedge \ x' = 1 \ \wedge \ \boxminus(x = 1)}_{\varphi_1'} \ .$$

Since ρ_{ℓ_0} implies at'_ℓ_1 and $x' = x$, and $\boxminus(x = 1)$ implies $x = 1$, the entailment is valid.

The rest of the premises are proven in a similar way. This establishes the validity of $\diamondsuit\big(at\_\ell_2 \wedge \boxminus(x = 1)\big)$. ▟

Completeness of Rule GUAR

Rule GUAR is obviously sound, which means that the P-validity of the premise $\mathit{first} \wedge \Theta \Rightarrow \diamondsuit r$ implies the P-validity of the conclusion $\diamondsuit r$.

The rule is also *complete*, which means that the P-validity of the conclusion implies the P-validity of the premise. Consider σ, an arbitrary computation of program P. By the assumption that $\diamondsuit r$ is P-valid, there exists a position k at which r holds. For σ to satisfy the premise we have to show that every position $i \geq 0$, satisfying $\mathit{first} \wedge \Theta$, is followed by a position j, $j \geq i$, satisfying r. Since 0 is the only position satisfying $\mathit{first} \wedge \Theta$, we can take j to be k.

Completeness of rule GUAR is important because it tells us that the rule is adequate for proving all P-valid guarantee formulas.

1.9 Obligation Properties

Before studying the class of obligation properties, we introduce a special class of response formulas.

Escape Formulas

Some response properties are naturally expressed by formulas of the form

$$p \quad \Rightarrow \quad \square q \ \vee \ \diamondsuit r,$$

for past formulas p, q, and r.

This formula claims that, following a p-state, either q will hold forever or r eventually occurs. We may view such a formula as stating that, following p, q should hold continually unless we escape to a state that eventually leads to r. Consequently, we refer to formulas of this form as *escape formulas*.

To see that this formula specifies a response property, observe that it is equivalent to

$$\neg q \ \wedge \ (\neg r)\, \Sigma\, (p \wedge \neg r) \quad \Rightarrow \quad \diamondsuit r.$$

In this form, the formula states that every $\neg q$-position preceded by a p-position such that no r has occured since, must be followed by an r-position.

While, in principle, it is possible to use the general response rules to establish escape formulas, it is more convenient to use a special rule, presented in Fig. 1.42.

For past formulas p, q, r, and φ

$$\begin{array}{ll} \text{E1.} & p \;\Rightarrow\; q \,\mathcal{W}\, \varphi \\ \text{E2.} & \varphi \;\Rightarrow\; \diamondsuit\, r \\ \hline & p \;\Rightarrow\; \square\, q \;\vee\; \diamondsuit\, r \end{array}$$

Fig. 1.42. Rule ESC (escape).

Rule ESC uses an auxiliary past formula φ. Premise E1 requires that, following a p-position, either q will hold forever or q will hold until an occurrence of φ. Premise E2 requires that every φ-position is followed by an r-position. Typically, we prove E1 by rule P-WAIT, a past version of rule WAIT (Fig. 3.3 of the SAFETY book), and E2 by appropriate response rules.

Example Consider program MAY-HALT of Fig. 1.43. This trivial program has a nondeterministic choice at ℓ_1 between getting deadlocked at ℓ_2 or, taking the ℓ_1^a branch, proceeding to ℓ_3. Consequently, the program has some computations that reach ℓ_2 and stay there forever, and some computations that never halt.

$$\ell_0\colon \textbf{loop forever do} \left[\begin{array}{l} \ell_1\colon \left[\begin{array}{l} \ell_1^a\colon \textbf{skip} \\ \quad \textbf{or} \\ \ell_1^b\colon \textbf{skip};\ \ell_2\colon \textbf{halt} \end{array}\right] \\ \ell_3\colon \textbf{skip} \\ \ell_4\colon \textbf{skip} \end{array}\right]$$

Fig. 1.43. Program MAY-HALT (possible deadlock).

We use rule ESC to prove the property

$$\psi_1\colon \quad \underbrace{at\_\ell_{0,1}}_{p} \;\Rightarrow\; \square\, \underbrace{at\_\ell_{0..2}}_{q} \;\vee\; \diamondsuit\, \underbrace{at\_\ell_4}_{r}$$

for this program.

Take

$$\varphi\colon \quad at\_\ell_3.$$

Consider the two premises of rule ESC.

- Premise E1

  This premise requires

  $$\underbrace{at\_\ell_{0,1}}_{p} \;\Rightarrow\; \underbrace{at\_\ell_{0..2}}_{q} \;\mathcal{W}\; \underbrace{at\_\ell_3}_{\varphi}\ .$$

  It is straightforward to derive this property by rule WAIT.

- Premise E2

  $$\underbrace{at\_\ell_3}_{\varphi} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_4}_{r}\ .$$

  A single application of rule RESP-J establishes this property.

This establishes the considered escape property. ∎

From Escape to Obligation

The (simple) obligation class includes all the properties that can be specified by a formula of the form

$$\square q \;\vee\; \diamondsuit r,$$

for past formulas q and r.

We observe that such a formula can be rewritten as

$$\underbrace{\mathit{first} \wedge \Theta}_{p} \;\Rightarrow\; \square q \;\vee\; \diamondsuit r,$$

which represents it as a special case of an escape formula.

This observation inspires rule OBL (Fig. 1.44) for proving obligation proper-

For past formulas q, r, and φ,

$$\begin{array}{ll} \text{O1.} & \mathit{first} \wedge \Theta \;\Rightarrow\; q \,\mathcal{W}\, \varphi \\ \text{O2.} & \varphi \;\Rightarrow\; \Diamond r \\ \hline & \Box q \vee \Diamond r \end{array}$$

Fig. 1.44. Rule OBL (proving obligation properties).

ties.

Premise O1 requires that, from the beginning of the computation, q holds continuously and can be interrupted only at a position satisfying φ. Premise O2 states that φ guarantees an eventual r. Consequently, q can be interrupted only when r is guaranteed. It follows that $q \,\mathcal{W}\, (\Diamond r)$ is valid. By properties of the waiting-for operator, we may deduce $\Box q \vee \Diamond r$.

Example (incrementor-decrementor)

Consider Program INC-DEC presented in Fig. 1.45, which nondeterministically increments or decrements an integer variable y.

$$\begin{array}{l} \textbf{local } x\text{: } \textbf{boolean where } x = \text{T} \\ \qquad\quad y\text{: } \textbf{integer where } y = 10 \\ \\ \ell_0\text{: } \textbf{loop forever do} \\ \\ \qquad \ell_1\text{: } \left[\begin{array}{l} \ell_1^a\text{: } \langle \textbf{when } x \textbf{ do } y := y + 1 \rangle \\ \qquad\quad \textbf{or} \\ \ell_1^b\text{: } \langle \textbf{when } x \textbf{ do } x := \text{F} \rangle \\ \qquad\quad \textbf{or} \\ \ell_1^c\text{: } \langle \textbf{when } \neg x \textbf{ do } y := y - 1 \rangle \end{array}\right] \end{array}$$

Fig. 1.45. Program INC-DEC (nondeterministic incrementor-decrementor).

We wish to prove for this program the obligation property

$$\Box \underbrace{y \geq 10}_{q} \;\vee\; \Diamond \underbrace{y = 0}_{r} \;.$$

Intending to apply rule OBL, it only remains to identify the intermediate formula φ. The main characterization of φ is that it describes the event whose occurrence guarantees the eventual realization of r.

Examining the program, we see that the first moment we realize that $y = 0$ is going to happen is when x becomes false. Consequently we take

$$\varphi: \quad \neg x \ \wedge \ y > 0.$$

The premises that have to be verified are as follows:

- Premise O1

To establish this premise it suffices to prove

$$\underbrace{\cdots \wedge \Theta}_{\widehat{p}} \quad \Rightarrow \quad \underbrace{y \geq 10}_{\widehat{q}=q} \ \mathcal{W} \ \underbrace{\neg x \ \wedge \ y > 0}_{\widehat{r}=\varphi} .$$

To prove this we use rule WAIT (Fig. 3.3 of the SAFETY book) with the intermediate assertion

$$\widehat{\varphi}: \quad x \ \wedge \ y \geq 10.$$

The three premises of rule WAIT require

$$\text{W1.} \qquad \underbrace{at\_\ell_0 \ \wedge \ x \ \wedge \ y = 10}_{\widehat{p}} \quad \rightarrow \quad \underbrace{x \ \wedge \ y \geq 10}_{\widehat{\varphi}} \ \vee \ \cdots ,$$

which is obviously state valid.

$$\text{W2.} \qquad \underbrace{x \ \wedge \ y \geq 10}_{\widehat{\varphi}} \ \rightarrow \ \underbrace{y \geq 10}_{\widehat{q}} ,$$

which is trivially state valid.

$$\text{W3.} \qquad \rho_\tau \ \wedge \ \underbrace{x \ \wedge \ y \geq 10}_{\widehat{\varphi}} \ \rightarrow \ \underbrace{x' \ \wedge \ y' \geq 10}_{\widehat{\varphi}'} \ \vee \ \underbrace{\neg x' \ \wedge \ y' > 0}_{\widehat{r}'} ,$$

for all transitions τ in the program. It is not difficult to see that all three requirements are valid.

- Premise O2

To establish this premise we have to prove

$$\underbrace{\neg x \ \wedge \ y > 0}_{\varphi} \quad \Rightarrow \quad \diamondsuit \underbrace{y = 0}_{r} .$$

This is proven by the RANK verification diagram presented in Fig. 1.46.

This concludes the proof of

$$\square(y \geq 10) \ \vee \ \diamondsuit(y = 0). \qquad \blacksquare$$

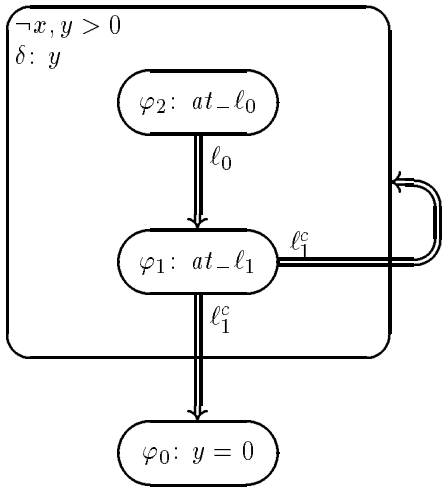


Fig. 1.46. Verification diagram for $\neg x \wedge y > 0 \Rightarrow \diamondsuit(y = 0)$.

A general obligation property is a conjunction of the form

$$\bigwedge_{i=1}^{n} (\square\, q_i \ \vee\ \diamondsuit\, r_i).$$

Consequently, to prove the validity of such a formula, it is sufficient (and necessary) to prove the validity of each conjunct, which is a simple obligation property.

Completeness of rule OBL

Rule OBL is *complete* for proving (simple) obligation properties. This means that, whenever $\square\, q \vee \diamondsuit\, r$ is P-valid, we can find a past formula φ, such that premises O1 and O2 are also P-valid. The choice we can always make is taking

$$\varphi: \qquad r \ \vee\ (\neg q \ \wedge\ \boxminus \neg r).$$

We will show that, if the property $\square\, q \vee \diamondsuit\, r$ is P-valid, then the two premises of rule OBL are also P-valid for this choice of φ.

- Premise O1

For premise O1 it suffices to show that

$$q \ \mathcal{W}\ \underbrace{r \ \vee\ (\neg q \ \wedge\ \boxminus \neg r)}_{\varphi}$$

is P-valid.

This formula states that either q holds forever, or it is interrupted by an r, or

it is interrupted by a $\neg q$-position which is not preceded by any r-position. This formula is valid in general so, in particular, it holds over all computations.

- Premise O2

This premise requires

$$\underbrace{r \ \vee \ (\neg q \ \wedge \ \boxminus \neg r)}_{\varphi} \ \Rightarrow \ \diamondsuit r,$$

for which it is sufficient to show

$$\neg q \ \wedge \ \boxminus \neg r \ \Rightarrow \ \diamondsuit r.$$

Assume to the contrary, that there exists a computation σ and a position i such that $\neg q \wedge \boxminus \neg r$ holds at i, but r does not hold at any position $j \geq i$. Since $\neg q \wedge \boxminus r$ holds at i, r does not occur at any position $j < i$. Therefore σ does not satisfy $\diamondsuit r$. On the other hand, since $\neg q$ at i, σ also does not satisfy $\square q$. This contradicts our assumption that $\square q \vee \diamondsuit r$ is P-valid.

Consequently, premise O2 is also P-valid.

As we continuously remind the reader, the auxiliary formula φ constructed during a proof of completeness is not necessarily the one we recommend for actual use. In practice, we can almost always find better assertions.

Problems

Problem 1.1 (three values) page 22

Prove accessibility for process P_1 of program MUX-VAL-3 of Fig. 1.47. This program uses the shared integer variables y_1 and y_2. Obviously, these variables can only assume one of the values $\{-1, 0, 1\}$.

Accessibility for P_1 is stated by the response formula

$$at\_\ell_2 \ \Rightarrow \ \diamondsuit \, at\_\ell_4.$$

Problem 1.2 (bakery algorithms) page 25

(a) Prove accessibility for process P_1 of program MUX-BAK-A of Fig. 1.48. Note that the two processes are not exactly symmetric due to the difference between statements ℓ_3^b and m_3^b.

The algorithm is called the *bakery* algorithm, since it is based on the idea that customers, as they enter, pick numbers which form an ascending sequence. Then, a customer with a lower number has higher priority in accessing its critical section. Statements ℓ_2 and m_2 ensure that the number assigned to y_i, $i = 1, 2$, is greater than the current value of y_j, $j \neq i$.

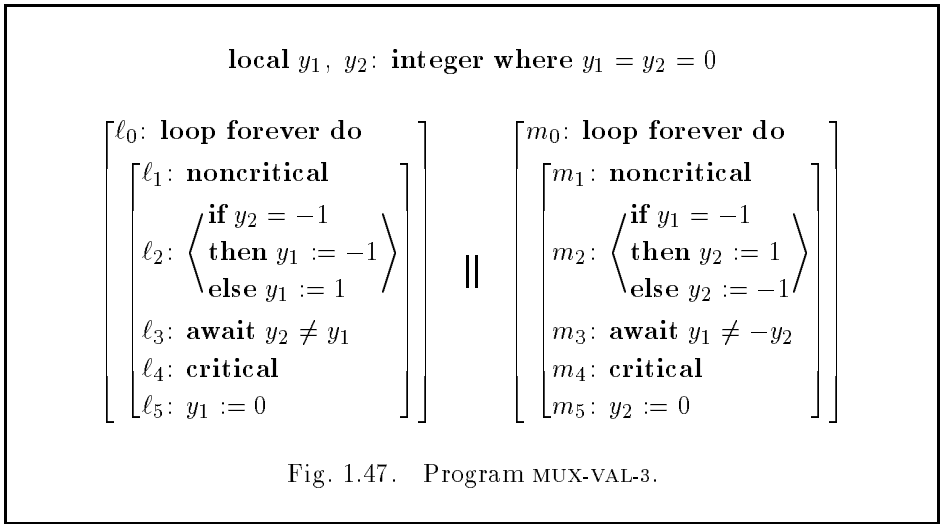


Fig. 1.47. Program MUX-VAL-3.

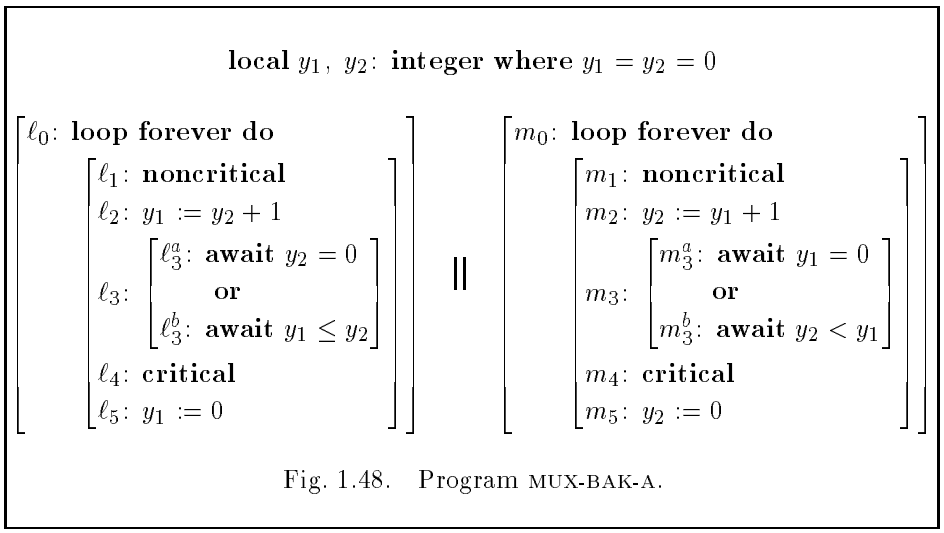


Fig. 1.48. Program MUX-BAK-A.

(b) Program MUX-BAK-A does not obey the LCR restriction. In particular, statements ℓ_2 and m_2 each contain two critical references: to y_1 and to y_2. To correct this situation, we propose program MUX-BAK-C of Fig. 1.49. This LCR-program contains two additional *await* statements that ensure that processes do not wait too long at locations ℓ_3 or m_3. Show that program MUX-BAK-C guarantees accessibility for process P_1.

Problem 1.3 (variants of Dekker's algorithm)) page 32

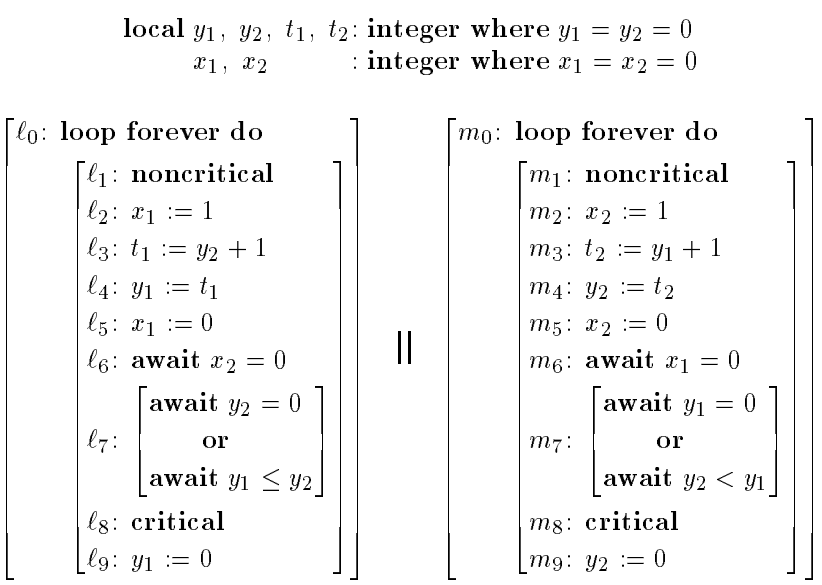


Fig. 1.49. Program MUX-BAK-C.

(a) Prove accessibility for process P_1 of program MUX-DEK-A of Fig. 1.50. That is, show that the response formula

$$at\_\ell_2 \ \Rightarrow\ \diamondsuit\, at\_\ell_7$$

is valid over MUX-DEK-A.

(b) Prove accessibility for process P_1 of program MUX-DEK-B of Fig. 1.51. That is, show that the response formula

$$at\_\ell_2 \ \Rightarrow\ \diamondsuit\, at\_\ell_7$$

is valid over MUX-DEK-B.

Problem 1.4 (condensed form of ranking functions) page 47

In the text, it was suggested that a ranking function $\delta = (d_1, d_2)$, where d_1 and $d_2 < M$ are natural numbers, can always be replaced by the ranking function $\widehat{\delta} = M \cdot d_1 + d_2$. Show that if both $d_2 < M$ and $d_2' < M$, then

$$\widehat{\delta} \ =\ M \cdot d_1 + d_2 > \widehat{\delta'} \ =\ M \cdot d_1' + d_2' \qquad \text{iff} \qquad \delta\colon (d_1, d_2) \ \succ\ \delta'\colon (d_1', d_2').$$

Problem 1.5 (rule with relaxed premise JW2) page 52

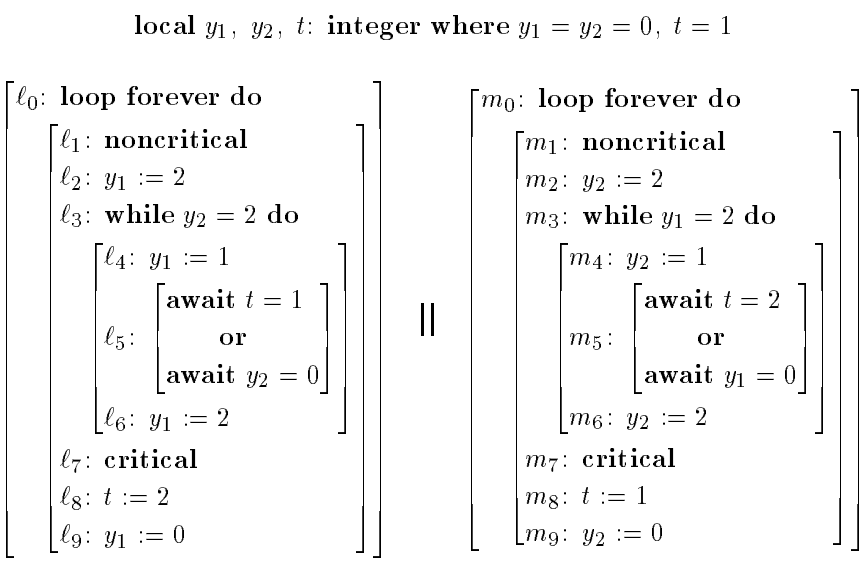


Fig. 1.50. Program MUX-DEK-A (a variant of Dekker's algorithm).

Consider a version of rule WELL-J in which premise JW2 has been replaced by the weaker premise

$$\widehat{\text{JW2}}. \qquad \rho_\tau \ \wedge \ \varphi_i \ \rightarrow \ q' \ \vee \ \bigvee_{j=1}^{k} (\varphi'_j \ \wedge \ \delta_i \succcurlyeq \delta'_j),$$

where $\delta_i \succcurlyeq \delta'_j$ stands for $(\delta_i \succ \delta'_j) \vee (\delta_i = \delta'_j)$. This premise requires that either q is achieved by τ, or the rank does not increase and *some* assertion φ_j (not necessarily φ_i) holds after the transition.

Show that the resulting rule is unsound. That is, show a property that satisfies premises JW1, $\widehat{\text{JW2}}$, JW3, and JW4, over a given program, and yet is invalid. This will show that persistence of helpful transitions is essential.

Problem 1.6 (integer division) page 59

Program IDIV of Fig. 1.52 accepts two positive integers in variables x and y and places in variable z their integer quotient $x \ div \ y$, and in variable w the remainder of their division $x \ mod \ y$. Prove total correctness of program IDIV, which can be specified by the response formula

$$\Theta \ \Rightarrow \ \diamondsuit(at\_\ell_5 \ \wedge \ x = z \cdot y + w \ \wedge \ 0 \leq w < y).$$

local $y_1,\ y_2, t$: **integer where** $y_1 = 0,\ y_2 = 0,\ t = 1$

$$
\left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } y_1 := 2 \\
\ell_3\text{: } \textbf{while } y_2 = 2 \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_4\text{: } \textbf{if } t = 2 \textbf{ then} \\
\quad \left[\begin{array}{l}
\ell_5\text{: } y_1 := 1 \\
\ell_6\text{: } \textbf{await } t = 1 \\
\ell_7\text{: } y_1 := 2
\end{array}\right]
\end{array}\right] \\
\ell_8\text{: } \textbf{critical} \\
\ell_9\text{: } t := 2 \\
\ell_{10}\text{: } \textbf{await } y_2 \neq 1 \\
\ell_{11}\text{: } y_1 := 0
\end{array}\right]
\end{array}\right]
\quad \| \quad
\left[\begin{array}{l}
m_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
m_1\text{: } \textbf{noncritical} \\
m_2\text{: } y_2 := 2 \\
m_3\text{: } \textbf{while } y_1 = 2 \textbf{ do} \\
\quad \left[\begin{array}{l}
m_4\text{: } \textbf{if } t = 1 \textbf{ then} \\
\quad \left[\begin{array}{l}
m_5\text{: } y_2 := 1 \\
m_6\text{: } \textbf{await } t = 2 \\
m_7\text{: } y_2 := 2
\end{array}\right]
\end{array}\right] \\
m_8\text{: } \textbf{critical} \\
m_9\text{: } t := 1 \\
m_{10}\text{: } \textbf{await } y_1 \neq 1 \\
m_{11}\text{: } y_2 := 0
\end{array}\right]
\end{array}\right]
$$

Fig. 1.51. Program MUX-DEK-B.

This formula states that every computation of program IDIV terminates (i.e., reaches the terminal location ℓ_5) with values of z, w satisfying $x = z \cdot y + w$ and $0 \leq w < y$.

in $x,\ y$: **integer where** $x > 0,\ y > 0$
local t : **integer**
out $z,\ w$: **integer where** $z = w = 0$

ℓ_0: $t := x$

ℓ_1: **while** $t > 0$ **do**

$\quad \ell_2$: **if** $w + 1 = y$
$\quad\quad$ **then** ℓ_3: $(z,\ w,\ t) := (z + 1,\ 0,\ t - 1)$
$\quad\quad$ **else** ℓ_4: $(z,\ w,\ t) := (z,\ w + 1,\ t - 1)$

ℓ_5:

Fig. 1.52. Program IDIV (integer division).

Problem 1.7 (greatest common divisor) page 59

Program GCDM of Fig. 1.53 accepts two positive integers in variables x_1 and x_2. It computes in z the greatest common divisor (gcd) of x_1 and x_2, and in variables w_1 and w_2 two integers which express z as a linear combination of the inputs x_1 and x_2. Prove total correctness of program GCDM, which can be stated by the response formula

$$\Theta \;\Rightarrow\; \diamondsuit\big(at\_\ell_6 \;\wedge\; z = gcd(x_1, x_2) \;\wedge\; z = w_1 \cdot x_1 + w_2 \cdot x_2\big).$$

in $x_1,\ x_2$: **integer where** $x_1 > 0,\ x_2 > 0$
local $y_1,\ y_2,\ t_1,\ t_2,\ t_3,\ t_4,\ u$: **integer**
out $z,\ w$: **integer**

ℓ_0: $(y_1,\ y_2,\ t_1,\ t_2,\ t_3,\ t_4) := (x_1,\ x_2,\ 1,\ 0,\ 0,\ 1)$

ℓ_1: $(y_1,\ y_2,\ u) := (y_2\ mod\ y_1,\ y_1,\ y_2\ div\ y_1)$

ℓ_2: **while** $y_1 \neq 0$ **do**

$$\left[\begin{array}{l} \ell_3\text{: } (t_1,\ t_2,\ t_3,\ t_4) := (t_2 - u \cdot t_1,\ t_1,\ t_4 - u \cdot t_3,\ t_3) \\ \ell_4\text{: } (y_1,\ y_2,\ u) := (y_2\ mod\ y_1,\ y_1,\ y_2\ div\ y_1) \end{array}\right]$$

ℓ_5: $(z,\ w_1,\ w_2) := (y_2,\ t_2,\ t_3)$

ℓ_6:

Fig. 1.53. Program GCDM(greatest common divisor with multipliers).

The program uses the operation div of integer division and the operation mod which computes the remainder of an integer division. In your proof you may use the following properties of the gcd function which hold for every nonzero integers m and n (possibly negative):

$$gcd(m, n) \;=\; gcd(m - n, n) \quad \text{for every } m \neq n$$
$$gcd(m, m) \;=\; |m|.$$

Problem 1.8 (computing the gcd and lcm) page 59

Program GCDLCM of Fig. 1.54 accepts two positive integers in variables x_1 and x_2. It computes in variable z their greatest common divisor and in variable w their least common multiple. Prove total correctness of program GCDLCM, which can be stated by the response formula

$$\Theta \;\Rightarrow\; \diamondsuit\big(at\_\ell_7 \;\wedge\; z = gcd(x_1, x_2) \;\wedge\; w = lcm(x_1, x_2)\big).$$

$$
\begin{array}{lll}
\textbf{in} & x_1,\ x_2 & : \textbf{integer where } x_1 > 0,\ x_2 > 0 \\
\textbf{local} & y_1,\ y_2,\ y_3,\ y_4 & : \textbf{integer} \\
\textbf{out} & z,\ w & : \textbf{integer}
\end{array}
$$

$$
\begin{array}{l}
\ell_0\colon\ (y_1,\ y_2,\ y_3,\ y_4) := (x_1,\ x_2,\ x_2,\ 0) \\
\ell_1\colon\ \textbf{while } y_1 \neq y_2 \textbf{ do} \\
\qquad \left[\begin{array}{l}
\ell_2\colon\ \textbf{if } y_1 > y_2 \textbf{ then} \\
\qquad\quad \ell_3\colon\ (y_1,\ y_4) := (y_1 - y_2,\ y_3 + y_4) \\
\ell_4\colon\ \textbf{if } y_1 < y_2 \textbf{ then} \\
\qquad\quad \ell_5\colon\ (y_2,\ y_3) := (y_2 - y_1,\ y_3 + y_4)
\end{array}\right] \\
\ell_6\colon\ (z,\ w) := (y_1,\ y_3 + y_4) \\
\ell_7\colon
\end{array}
$$

Fig. 1.54. Program GCDLCM (computing the *gcd* and *lcm*).

In your proof you may use the properties of the *gcd* function listed in Problem 1.7, and the following property of the *lcm* function:

$$lcm(m,n) \ = \ m \cdot n / gcd(m,n).$$

Problem 1.9 (set partitioning) page 59

Consider program EXCH presented in Fig. 1.55. The program accepts as input two sets of natural numbers S and T, whose initial values are S_0 and T_0, respectively.

Process P_1 repeatedly identifies and removes the maximal element in S and sends it to P_2 which places it in T. Symmetrically, P_2 identifies and removes the minimal element in T and sends it to P_1 which places it in S. The processes use the operations $\max(S)$ and $\min(T)$ which find, respectively, the maximal element in the set S and the minimal element in the set T. Show total correctness of program EXCH, which can be specified by the response formula

$$\Theta \ \Rightarrow \ \diamondsuit(at\_\ell_9 \ \wedge \ at\_m_7 \ \wedge \ |S| = |S_0| \ \wedge \ |T| = |T_0| \ \wedge \ S \leq T).$$

This formula states that the program terminates and on termination, sets S and T have preserved their initial sizes and that every element in S is smaller than or equal to every element in T.

Problem 1.10 (converse of Claim 1.3) page 67

Let $P :: [P_1 :: S_1 \| \cdots \| P_k :: S_k]$ be a program whose top-level processes communicate by shared variables and such that every program variable is owned by one

$$
\begin{array}{ll}
\textbf{in} & S_0,\ T_0\text{: } \textbf{set of natural where } S_0 \neq \emptyset,\ T_0 \neq \emptyset \\
\textbf{out} & S,\ T \ \text{: } \textbf{set of natural where } S = S_0,\ T = T_0 \\
\textbf{local} & \alpha,\ \beta \ \text{: } \textbf{channel of integer}
\end{array}
$$

$$
P_1 :: \left[\begin{array}{l}
\textbf{local } x,\ mx\text{: } \textbf{integer} \\
\ell_0\text{: } x := -1 \\
\ell_1\text{: } mx := \max(S) \\
\ell_2\text{: } \textbf{while } x < mx \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_3\text{: } \alpha \Leftarrow mx \\
\ell_4\text{: } S := S - \{mx\} \\
\ell_5\text{: } \beta \Rightarrow x \\
\ell_6\text{: } S := S \cup \{x\} \\
\ell_7\text{: } mx := \max(S)
\end{array}\right] \\
\ell_8\text{: } \alpha \Leftarrow -1 \\
\ell_9\text{:}
\end{array}\right]
\quad \| \quad
P_2 :: \left[\begin{array}{l}
\textbf{local } y,\ mn\text{: } \textbf{integer} \\
m_0\text{: } \alpha \Rightarrow y \\
m_1\text{: } \textbf{while } y \geq 0 \textbf{ do} \\
\quad \left[\begin{array}{l}
m_2\text{: } T := T \cup \{y\} \\
m_3\text{: } mn := \min(T) \\
m_4\text{: } \beta \Leftarrow mn \\
m_5\text{: } T := T - \{mn\} \\
m_6\text{: } \alpha \Rightarrow y
\end{array}\right] \\
m_7\text{:}
\end{array}\right]
$$

Fig. 1.55. Program EXCH (partitioning two sets).

of the top-level processes. Let σ be a model such that the interpretation of π is a subset of the locations of P and σ is simultaneously a modular computation of every P_i, $i = 1, \ldots, k$. Show that σ is a computation of P.

Problem 1.11 (completeness of rule COMP-B) page 68

Let $P :: [P_1 :: S_1 \| \cdots \| P_k :: S_k]$ be a program whose top-level processes communicate by shared variables and such that every program variable is owned by one of the top-level processes. Let φ be a P-valid formula. Show that the P-validity of φ can be compositionally inferred from modular validities, using rule COMP-B and temporal reasoning. This establishes the completeness of rule COMP-B for compositional verification.

A solution to this problem can be organized as follows.

- For each top-level process P_i, $i = 1, \ldots, k$, construct a formula ψ_i capturing the temporal modular semantics of P_i. That is, a model σ satisfies ψ_i iff σ is a modular computation of P_i. Argue semantically that

  $$P_i \ \models_m \ \psi_i,$$

  for each $i = 1, \ldots, k$.

- Use rule COMP-B and temporal reasoning to infer

$$P \models \psi_1 \wedge \cdots \wedge \psi_k.$$

- Argue semantically that a model σ satisfies $\psi_1 \wedge \cdots \wedge \psi_k$ iff σ is a computation of the entire program P. Therefore, if φ is P-valid, then the following general validity holds:

$$\models \psi_1 \wedge \cdots \wedge \psi_k \rightarrow \varphi.$$

- Apply temporal reasoning to $P \models \psi_1 \wedge \cdots \wedge \psi_k$ and $\models \psi_1 \wedge \cdots \wedge \psi_k \rightarrow \varphi$ to infer

$$P \models \varphi.$$

Chapter 3
Response Under Fairness

Chapters 1 and 2 considered proof rules for response properties that depend on *justice* for their validity. While justice is an important fairness requirement, it is only one of the two fairness requirements listed in a general fair transition system. In this chapter we consider proof rules for response properties that depend on the two *fairness* requirements, i.e., *justice* and *compassion*, for their validity.

In a way similar to Chapter 1, we first introduce a single-step rule for response under compassion, and then introduce the fair versions of CHAIN and WELL.

Let us review the basic phenomena that are modeled by the requirements of justice and compassion.

Justice

Justice represents a natural property of multi-processor systems. It represents the obvious intuition that parallel processes, implemented on separate processors, should progress independently as long as they do not need to synchronize with other processes. This intuitively obvious fact for multi-processor systems must be represented explicitly by justice in our model, due to the artifact of modeling concurrency by interleaving.

Justice excludes from the set of computations runs in which an enabled process does not progress beyond some point. Consequently, justice represents in our model the natural phenomena of independent progress in parallel components of a multi-processor system.

Since justice is associated with the general phenomenon of parallelism, it is not restricted to particular statements. In the computational model we consider here, all transitions, with two exceptions, are required to be just, i.e., are members of the justice set $\mathcal{J}$. The exceptions are the idling transition τ_I and the transition associated with the schematic *noncritical* statement.

Compassion

Compassion excludes from the set of computations runs in which one of the compassionate transitions is enabled infinitely many times but taken only finitely many times.

In comparison with justice, compassion has a much narrower scope and is associated with only a small subset of statements and their corresponding transitions. A case in point is the semaphore *request* statement.

We should bear in mind that a correct implementation of a semaphore which ensures the required compassion property must be based on either a nontrivial software protocol or a special hardware arbiter. For example, a protocol for semaphore services may place all requests in an internal queue, and allocate the semaphore on a first-come-first-serve basis.

In our high-level programming language, all these low-level details are abstracted away and the whole transaction between customer and server is represented by the single statement **request** y. The compassion requirement associated with this statement provides an abstract representation of the internal queing mechanism, replacing the *first-come-first-serve* guarantee by the weaker *infinitely-often-requested-eventually-served* guarantee.

We can view our extensive study of mutual exclusion algorithms that do not use semaphores as an investigation of ways to implement compassion using only justice. In fact, each of these algorithms can be used to implement a semaphore, which is the simplest manifestation of compassion.

Another class of statements associated with compassion requirements are the communication statements. As in the case of semaphores, the compassion assumption provides a useful abstraction that hides a more involved low-level protocol that ensures fair arbitration between processes that compete for an access to a common channel.

Compassion can also be used to model *probabilistic* behavior. Consider a program consisting of two processes that communicate over an unreliable channel. A probabilistic characterization of such a channel may specify, for example, that on the average it loses one message out of ten. This assigns a probability of 0.9 that a message sent will arrive on the other side. Assume that we have a protocol that overcomes the unreliability of the channel by repeatedly sending a message until it is acknowledged. To verify the correctness of such a protocol by the tools presented in this book, we replace the quantitative characterization

A message submitted will get across with probability 0.9

by the compassion requirement

A message submitted infinitely many times will eventually get across.

Some examples presented in this chapter illustrate the use of compassion for modeling probabilistic behavior.

In most of the chapter, we consider the simpler case of response formulas $p \Rightarrow \Diamond q$, where p and q are assertions. In the last subsection of Section 3.7, we discuss the generalization to the case that p and q are arbitrary past formulas.

In Section 3.1 we present a single-step rule that relies on the activation of a single compassionate transition.

Section 3.2 shows how to combine several applications of the just and compassionate single-step rules into a chain rule that relies on a fixed number of helpful transitions. We also introduce verification diagrams which provide a graphical representation of the chain rule.

Section 3.3 illustrates applications of the chain rule under general fairness to example programs that involve message passing.

Section 3.4 explores the use of compassion to model channels which are only eventually reliable, i.e., can only guarantee that a message that is repeatedly transmitted eventually gets through.

Section 3.5 presents a well-founded rule for response under general fairness.

Section 3.6 considers different proposed solutions to the dining philosophers problem. In the SAFETY book, we established the safety properties of these solutions. Here, we complete their formal verification by proving that most of those solutions satisfy the progress property of accessibility.

In Section 3.7 we complete the formal verification of several resource-allocation programs considered in the SAFETY book, by establishing their response properties.

Section 3.8 establishes the (relative) completeness of the well-founded rule introduced in Section 3.5 for proving all response properties of a given program. This section considers the special (but most prevalent) case of a formula $p \Rightarrow \Diamond q$ where p and q are state formulas.

Section 3.9 extends the completeness proof to the general case of a response formula $p \Rightarrow \Diamond q$ where p and q are past formulas.

Section 3.10 presents algorithms for automatic verification of response formulas $p \Rightarrow \Diamond q$ over finite-state systems, for the case that p and q are state formulas.

Finally, Section 3.11 completes the discussion of algorithmic verification of response properties by considering the general case where p and q are past formulas.

3.1 A Single-Step Rule

The rules based on justice, considered in Chapters 1 and 2, are not powerful enough to prove response properties based on compassion. To see this, consider the standard semaphore program for mutual exclusion, presented in Fig. 3.1. This program was first introduced in Fig. 1.6 of the SAFETY book.

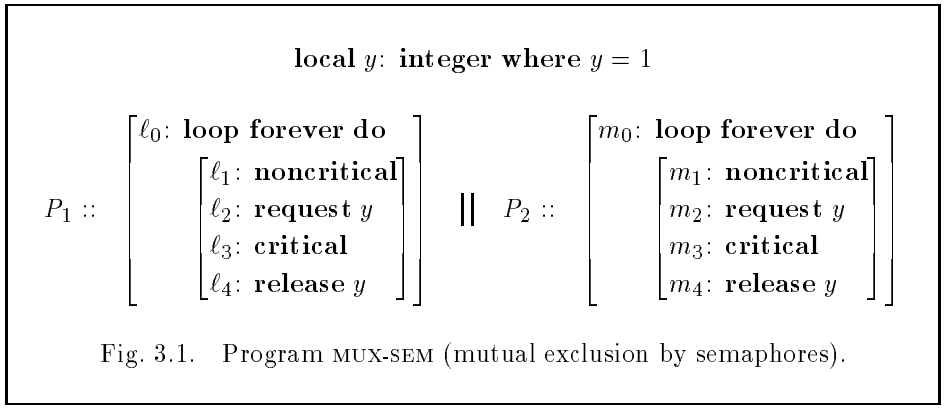


Fig. 3.1. Program MUX-SEM (mutual exclusion by semaphores).

The main response property associated with this program is individual accessibility. Stated for process P_1, it is given by

$$\underbrace{at\_\ell_2}_{p} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_3}_{q} \;.$$

The situation is very easy to analyze. Clearly, we can identify ℓ_2 as a helpful transition in this case, which obviously needs only one helpful step to achieve the goal at_ℓ_3.

Let us consult rule RESP-J (Fig. 1.1), which is the strongest rule for single-step response properties we have. Premises J1–J3 of this rule are satisfied for

$$\varphi = p\colon \quad at\_\ell_2 \qquad\qquad \tau\colon \quad \ell_2 .$$

That is, the helpful transition ℓ_2 leads from at_ℓ_2 to at_ℓ_3, while all other transitions preserve at_ℓ_2. The premise we fail to prove is J4, which requires

$$at\_\ell_2 \;\rightarrow\; En(\ell_2),$$

or equivalently

$$at\_\ell_2 \;\rightarrow\; at\_\ell_2 \;\wedge\; y > 0.$$

This, of course, is not a valid invariant for program MUX-SEM.

Let us recall the reasons for including premise J4 in rule RESP-J. While premises J2 and J3 ensure that φ is preserved until q is achieved, and that the next τ-transition taken from a φ-state achieves q, premise J4 forces, through justice,

that transition τ eventually will be taken. Justice guarantees the activation of τ only if τ is continuously enabled beyond some point. Since φ holds continuously until q is achieved, premise J4 ties the enableness of τ with φ, ensuring that τ will be continuously enabled until q is achieved. If q is not achieved, τ will be continuously enabled and never taken, violating justice with respect to τ.

Compassion is less demanding about the degree of enableness of τ that is required to ensure activation. For a compassionate transition τ, it is only necessary that τ be infinitely often enabled to guarantee its activation. Consequently, we may relax the tie between φ and the enableness of τ and, instead of requiring that τ is enabled *whenever* φ holds, only require that if φ holds then *eventually* τ should be enabled.

The Single-Step Rule

Relaxation of the last premise leads to rule RESP-C for proving single-step response under compassion (Fig. 3.2).

For assertions p, q, φ, and transition $\tau_h \in \mathcal{C}$

$$\begin{array}{ll} \text{C1.} & p \;\rightarrow\; q \;\vee\; \varphi \\ \text{C2.} & \{\varphi\}\; \mathcal{T}\; \{q \;\vee\; \varphi\} \\ \text{C3.} & \{\varphi\}\; \tau_h\; \{q\} \\ \text{C4.} & \varphi \;\Rightarrow\; \diamondsuit\big(\neg\varphi \;\vee\; \mathit{En}(\tau_h)\big) \\ \hline & p \;\Rightarrow\; \diamondsuit\, q \end{array}$$

Fig. 3.2. Rule RESP-C (single-step response under compassion).

Premise C1 requires that any p-position, that does not already satisfy q satisfies the intermediate assertion φ. Premise C2 requires that the application of any transition to a φ-state yields a state satisfying q or φ. Premise C3 requires that the application of the helpful transition τ_h to a φ-position yields a q-position. Premise C4, which distinguishes RESP-C from RESP-J, requires that if φ holds then eventually either φ will stop holding, or the helpful transition τ_h will become enabled. Note that, due to C2 and C3, the first transition that falsifies φ leads to a q-position. Consequently, C4 can also be written as

$$\varphi \;\Rightarrow\; \diamondsuit\big(q \;\vee\; \mathit{En}(\tau_h)\big).$$

Justification To justify the rule, consider a computation σ satisfying the four premises of the rule. Assume that p holds at some position $k \geq 0$. We wish to show that q holds at some later position $j \geq k$.

Assume, to the contrary, that q never holds beyond k. By C1, φ also holds at k. By C2 and the assumption that q does not hold beyond k, φ holds continuously beyond k.

Consider position $i_1 = k$, at which φ holds. By C4, there exists a position j_1, $j_1 \geq i_1$, at which either φ is false or τ_h is enabled. Since φ is true at all positions beyond k, τ_h is enabled at j_1. Take $i_2 = j_1 + 1$. By C4, since φ also holds at i_2, there exists a position j_2, $j_2 \geq i_2$, at which, again, τ_h is enabled. By repeating the argument we construct an infinite sequence of positions $j_1 < j_2 < \cdots$, at all of which τ_h is enabled.

Thus, τ_h is enabled at infinitely many positions in σ. Transition τ_h is not taken beyond k since, by C3, any application of τ_h to a φ-position results in a q-position, and by our assumption, there are no q-states beyond k. This violates the requirement of compassion with respect to τ_h. Thus, q must hold at some position beyond k. ∎

In applications of the rule, it is sufficient to check premise C2 for $\tau \neq \tau_h$, since C2 for $\tau = \tau_h$ is implied by C3.

It is easy to see that, for a transition τ_h that is both just and compassionate, rule RESP-C is stronger than rule RESP-J. This means that the premises of rule RESP-J imply the premises of rule RESP-C. This is because the first three premises of the two rules are identical, and due to the reflexivity and monotonicity of the $\diamondsuit$ operator,

$$\text{J4.} \quad \varphi \ \rightarrow \ En(\tau_h)$$

implies

$$\text{C4.} \quad \varphi \ \Rightarrow \ \diamondsuit\big(\neg\varphi \ \vee \ En(\tau_h)\big).$$

On the other hand, rule RESP-C has a more limited applicability than rule RESP-J. In rule RESP-J we may choose any just transition to be the helpful one. Rule RESP-C is restricted to use only compassionate transitions, which form a much smaller set than that of the just transitions.

In most applications of rule RESP-C premise C4 is established by proving the stronger statement

$$\widehat{\text{C4}}. \qquad \varphi \ \Rightarrow \ \diamondsuit \, En(\tau_h),$$

which obviously implies C4.

Rule RESP-C appears to be recursive. To prove one response property, it uses as one of its premises another response property. How is this premise to be

proven? We will give several answers to this puzzling question, showing that such recursive rules do not necessarily lead to circular reasoning.

Example Let us use rule RESP-C to prove the accessibility property

$$\underbrace{at\_\ell_2}_{p} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_3}_{q}$$

for program MUX-SEM of Fig. 3.1.

Take

$$\varphi = p\colon \quad at\_\ell_2 \qquad\qquad \tau_h\colon \quad \ell_2 .$$

This choice identifies ℓ_2 as the helpful compassionate transition.

- Premise C1

This premise requires

$$\underbrace{at\_\ell_2}_{p} \;\rightarrow\; \cdots \;\vee\; \underbrace{at\_\ell_2}_{\varphi} ,$$

which is clearly valid.

- Premise C2

This premise is proven by showing

$$\rho_\tau \;\wedge\; \underbrace{at\_\ell_2}_{\varphi} \;\rightarrow\; \cdots \;\vee\; \underbrace{at'\_\ell_2}_{\varphi'}$$

for any $\tau \neq \ell_2$. Since while P_1 is at ℓ_2 no other transition of P_1 is enabled and no transition of P_2 can affect at_ℓ_2, the premise follows.

- Premise C3

This premise requires

$$\underbrace{\cdots \wedge\; at'\_\ell_3}_{\rho_{\ell_2}} \;\wedge\; \underbrace{at\_\ell_2}_{\varphi} \;\rightarrow\; \underbrace{at'\_\ell_3}_{q'}$$

which is obviously valid.

- Premise C4

This premise is established by proving

$$\underbrace{at\_\ell_2}_{\varphi} \;\Rightarrow\; \diamondsuit\Big(\cdots \;\vee\; \underbrace{at\_\ell_2 \;\wedge\; y > 0}_{En(\ell_2)}\Big) .$$

The proof of this response property using rule CHAIN-J is presented in the verification diagram of Fig. 3.3. In the proof we are aided by the invariant

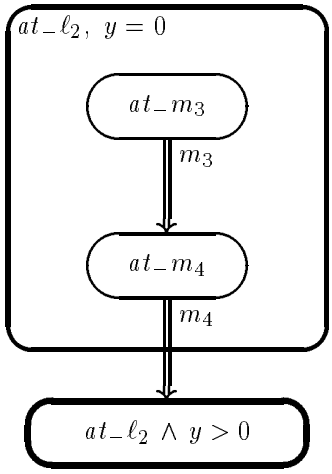


Fig. 3.3. Verification diagram for premise C4.

$$\chi_1\colon \quad at\_\ell_{3,4} \ + \ at\_m_{3,4} \ + \ y \ = \ 1.$$

Due to invariant χ_1, if at_ℓ_2 holds but $y \leq 0$, then $y = 0$ and P_2 must be at m_3 or m_4.

This concludes the proof of accessibility for program MUX-SEM. ∎

The example above provides a first answer to the problem of recursiveness of rule RESP-C. In some cases, the temporal premise C4 can be proven by using the J-rules.

3.2 Chain Rule

Naturally, after presenting a single-step rule for response under compassion, we should consider the compassionate version of rule CHAIN. Such a version can be obtained in the standard way, by considering a sequence of assertions $\varphi_0, \ldots, \varphi_m$ such that $\varphi_i \Rightarrow \diamondsuit\left(\bigvee_{j<i} \varphi_j\right)$ can be proven using rule RESP-C for each $i = 1, \ldots, m$. The conclusion of such a rule is $\left(\bigvee_{i=0}^{m} \varphi_i\right) \Rightarrow \diamondsuit\, \varphi_0$.

However, this version is not very useful. When tracing a chain of helpful steps, it is rarely the case that all of them are associated with compassionate transitions. The typical case is that most of the helpful steps are due to just transitions while

only one or two are due to compassionate transitions. Consequently, rule CHAIN-F presented in Fig. 3.4 allows both just and compassionate helpful transitions.

For assertions p and $q = \varphi_0, \ldots, \varphi_m$ and
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J} \cup \mathcal{C}$

F1. $p \rightarrow \bigvee_{j=0}^{m} \varphi_j$

F2. $\{\varphi_i\}\ \mathcal{T}\ \left\{\bigvee_{j \le i} \varphi_j\right\}$

F3. $\{\varphi_i\}\ \tau_i\ \left\{\bigvee_{j < i} \varphi_j\right\}$

F4. If $\tau_i \in \mathcal{C}$, then
C4. $\varphi_i \Rightarrow \diamondsuit\big(\neg\varphi_i \vee En(\tau_i)\big)$
Otherwise
J4. $\varphi_i \rightarrow En(\tau_i)$

} for $i = 1, \ldots, m$

$p \Rightarrow \diamondsuit q$

Fig. 3.4. Rule CHAIN-F (chain under fairness).

Premises F1–F3 of rule CHAIN-F are identical to the corresponding premises J1–J3 of rule CHAIN-J (Fig. 1.7).

Premise F4 of rule CHAIN-F requires either eventual or immediate enableness, according to whether τ_i is compassionate or only just. Note that if τ_i is compassionate, we may still attempt to prove for it the simpler (and stronger) version of F4, i.e., $\varphi_i \rightarrow En(\tau_i)$, because this implication obviously implies $\varphi_i \Rightarrow \diamondsuit\big(\neg\varphi_i \vee En(\tau_i)\big)$. Only if we fail to prove the simpler version, should we resort to a direct proof of the more complex version.

Justification The justification of rule CHAIN-F is similar to that of rule CHAIN-J. Assume a computation σ in which p holds at position k and no position beyond k satisfies q. From premises F1–F3 it follows that there exists a position $r \ge k$ and an index $i > 0$ such that φ_i holds continually beyond r and no φ_j, with $j < i$, holds at any such position.

Consider two cases. If τ_i, the helpful transition for φ_i, is compassionate then by F4, τ_i is enabled infinitely many times. It follows that σ does not fulfill the requirement of compassion with respect to τ_i, contradicting the assumption that σ is a computation. If τ_i is just but not compassionate, F4 implies that it is continually enabled which shows that σ violates the requirement of justice with respect to τ_i.

We conclude that the described situation is impossible, and every p-position must be followed by a q-position. ∎

In applications of the rule, it is sufficient to establish premise F2 for all $\tau \neq \tau_i$, since F2 for $\tau = \tau_i$ is implied by premise F3.

Example (producer-consumer)

We illustrate rule CHAIN-F on program PROD-CONS, presented in Fig. 3.5. This program was first introduced in Fig. 2.23 of the SAFETY book.

The response property we wish to establish for this program is accessibility, which for process *Prod* is given by

$$\underbrace{at\_\ell_2}_{p} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_4}_{q} \;.$$

In the proof, we are aided by the following invariants

$$\chi_0\colon \quad r \geq 0 \;\wedge\; ne \geq 0 \;\wedge\; nf \geq 0$$

$$\chi_1\colon \quad r + at\_\ell_{4,5} + at\_m_{3,4} \;=\; 1$$

$$\chi_2\colon \quad ne + nf + at\_\ell_{3..6} + at\_m_{2..5} \;=\; N.$$

To prove the accessibility property we use rule CHAIN-F. Inspecting the program, it is easy to identify the intermediate stages in the progress from ℓ_2 to ℓ_4. We choose

$$\varphi_2\colon\ at\_\ell_2 \qquad \tau_2\colon\ \ell_2$$

$$\varphi_1\colon\ at\_\ell_3 \qquad \tau_1\colon\ \ell_3$$

$$\varphi_0\colon\ at\_\ell_4.$$

It is straightforward to verify that each τ_i, $i = 1, 2$, leads from φ_i to φ_{i-1}, and that being at φ_i, each transition τ, $\tau \neq \tau_i$ preserves φ_i.

The only nontrivial premises to verify are the enableness premises F4 for $i = 1, 2$. In this case both τ_1 and τ_2 are compassionate transitions, so we should prove eventual enableness.

- Premise F4 for $i = 2$

For this premise it suffices to prove

$$\begin{array}{l} \textbf{local } r,\ ne,\ nf : \textbf{integer where } r = 1,\ ne = N,\ nf = 0 \\ \qquad b \qquad\quad : \textbf{list of integer where } b = \Lambda \end{array}$$

$$Prod :: \left[\begin{array}{l} \textbf{local } x\text{: integer} \\ \ell_0\text{: loop forever do} \\ \qquad \left[\begin{array}{l} \ell_1\text{: produce } x \\ \ell_2\text{: request } ne \\ \ell_3\text{: request } r \\ \ell_4\text{: } b := b \bullet x \\ \ell_5\text{: release } r \\ \ell_6\text{: release } nf \end{array}\right] \end{array}\right]$$

$$\Big\|$$

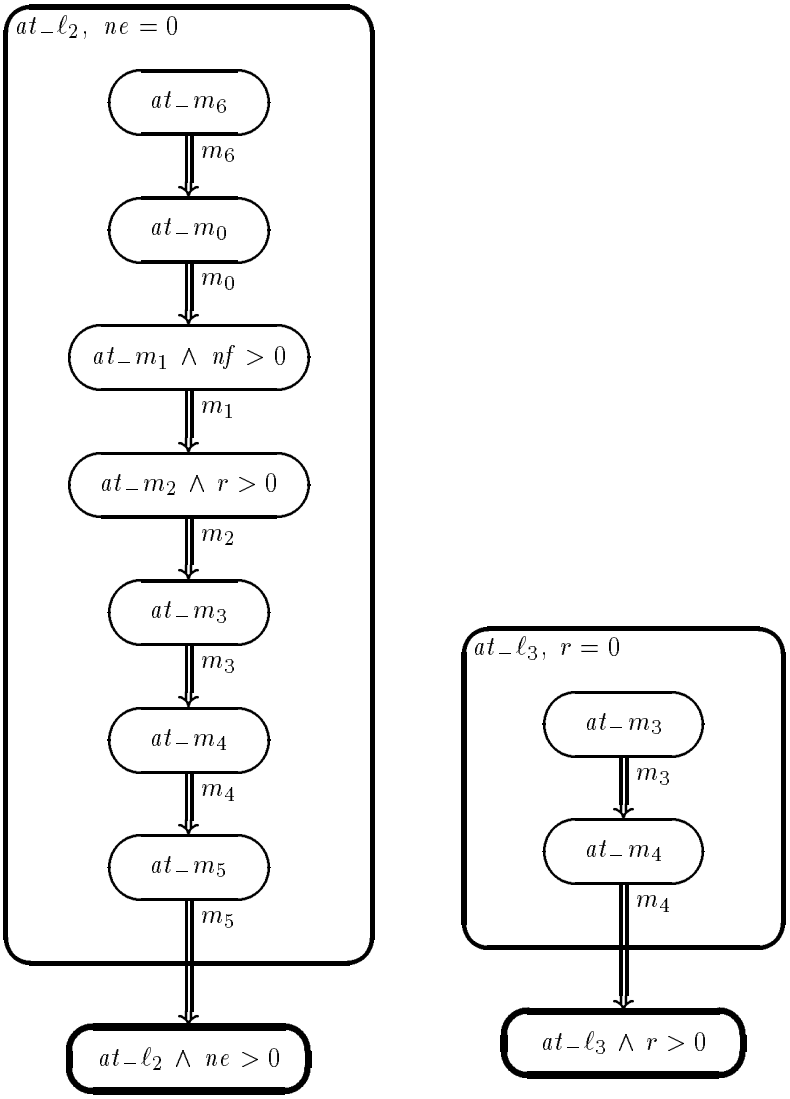
$$Cons :: \left[\begin{array}{l} \textbf{local } y\text{: integer} \\ m_0\text{: loop forever do} \\ \qquad \left[\begin{array}{l} m_1\text{: request } nf \\ m_2\text{: request } r \\ m_3\text{: } (y,\ b) := \big(hd(b),\ tl(b)\big) \\ m_4\text{: release } r \\ m_5\text{: release } ne \\ m_6\text{: consume } y \end{array}\right] \end{array}\right]$$

Fig. 3.5. Program PROD-CONS (producer-consumer).

$$\underbrace{at\_\ell_2}_{\varphi_2} \ \Rightarrow \ \diamondsuit\Big(\cdots \ \vee \ \underbrace{at\_\ell_2 \ \wedge \ ne > 0}_{En(\ell_2)}\Big) \ .$$

To prove this response property we use rule CHAIN-J. The proof is presented in the CHAIN diagram of Fig. 3.6(a). Note that we use the invariant $ne \geq 0$ to infer that if $ne \leq 0$ then in fact $ne = 0$. Invariant χ_2 is used to deduce that, when $ne = 0$ and $at\_\ell_2 \wedge at\_m_1$ holds, then $nf = N > 0$. Invariant χ_1 is used to deduce that, when $at\_\ell_2 \wedge at\_m_2$ holds, then $r = 1 > 0$.

An interesting point is that, even though m_1 and m_2 are compassionate transitions, we included them in a proof by rule CHAIN-J. This is possible because, under the assumption that P_1 is frozen at ℓ_2 and $ne = 0$, these transitions are immediately enabled when control is at their respective locations.



(a) Proof of premise F4 for $i = 2$. (b) Proof of premise F4 for $i = 1$.

Fig. 3.6. CHAIN diagrams.

- Premise F4 for $i = 1$

This premise is proven by showing

$$\underbrace{at\_\ell_3}_{\varphi_1} \;\Rightarrow\; \diamondsuit\Big(\cdots \;\vee\; \underbrace{at\_\ell_3 \;\wedge\; r>0}_{En(\ell_3)}\Big)\,.$$

The proof is based again on an application of rule CHAIN-J, and is presented in the CHAIN diagram of Fig. 3.6(b). We use the invariant $r \geq 0$ to infer that if $r \leq 0$ then $r = 0$. Then, we use χ_1 to deduce that if $r = 0$ and P_1 is at ℓ_3, then P_2 can only be at m_3 or at m_4.

This concludes the proof of the accessibility property

$$at\_\ell_2 \;\Rightarrow\; \diamondsuit\, at\_\ell_4.$$

In **Problem 3.2**, we request the reader to prove that the accessibility property is valid even over a weaker version of the program, in which the transitions corresponding to the semaphore *request* statements, are taken to be just, rather than compassionate. ∎

CHAIN-F *Diagrams*

Diagrams can also be used to provide a concise representation of a proof by rule CHAIN-F. The main extension over CHAIN diagrams is the introduction of a third type of an edge — a *solid* edge, where a solid edge will correspond to a helpful transition which is compassionate. Consequently, edges connecting nodes (assertions) in a diagram can be single (-line), double, or solid.

CHAIN-F Diagrams

A verification diagram is said to be an CHAIN-F *diagram* if its nodes are labeled by assertions $\varphi_0, \ldots, \varphi_m$ with φ_0 being the terminal node, and if it satisfies the following requirement:

- If a single edge connects node φ_i to node φ_j, then $i \geq j$.
- If a double or a solid edge connects φ_i to φ_j, then $i > j$.
- Every node φ_i, $i > 0$, has at least one departing edge which is either double or solid. The transition labeling this edge is identified as helpful for assertion φ_i.

Verification Conditions for CHAIN-F Diagrams

Let φ be a nonterminal node and $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the τ-successors of φ.

- If τ labels a single edge, we associate with φ and τ the verification condition
$$\{\varphi\}\ \tau\ \{\varphi \;\vee\; \varphi_1 \;\vee\; \cdots \;\vee\; \varphi_k\}.$$
- If τ labels a double or solid edge, we associate with φ and τ the verification condition

$$\{\varphi\}\ \tau\ \{\varphi_1 \vee \cdots \vee \varphi_k\}.$$

- If τ labels a double edge departing from φ, we associate with φ and τ the requirement

$$\varphi \rightarrow En(\tau).$$

- If τ labels a solid edge departing from φ, we associate with φ and τ the response formula

$$\varphi \Rightarrow \diamondsuit\big(\neg\varphi \vee En(\tau)\big).$$

We extend the notion of verification conditions to include also the enabling requirement associated with double edges and the response formulas associated with solid edges.

Valid CHAIN-F Diagrams

A CHAIN-F diagram is said to be *P-valid* if the first three types of verification conditions associated with its nodes and transitions are P-state valid, and the response formula conditions, associated with solid edges, are P-valid.

The consequences of having a valid CHAIN-F diagram are stated in the following claim.

Claim 3.1 (fair chain)

A P-valid CHAIN-F diagram establishes that the response formula

$$\bigvee_{j=0}^{m} \varphi_j \Rightarrow \diamondsuit\, \varphi_0$$

is P-valid.

If, in addition, we can establish the following implication:

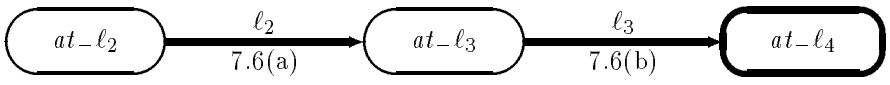
$$p \rightarrow \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \varphi_0 \rightarrow q$$

then we can conclude the P-validity of

$$p \Rightarrow \diamondsuit\, q.$$

Example (producer consumer)

The proof of the accessibility property of program PROD-CONS (Fig. 3.5) can be represented by the CHAIN-F diagram



Since each transition labeling a solid edge needs a subproof that establishes the validity of the compassionate version of premise F5, we often attach a reference to such edges, which points to the diagram for the subproof. Thus, in the above diagram, we referred to CHAIN diagram 3.6(a) as establishing the eventual enableness of ℓ_2, and to CHAIN diagram 3.6(b) as establishing the eventual enableness of ℓ_3. ∎

Example (multiple resource acquisition)

In Fig. 2.12 of the SAFETY book we introduced program RES-3 as an example of a program consisting of several processes that need several resources in order to perform their critical activity. We reproduce this program in Fig. 3.7.

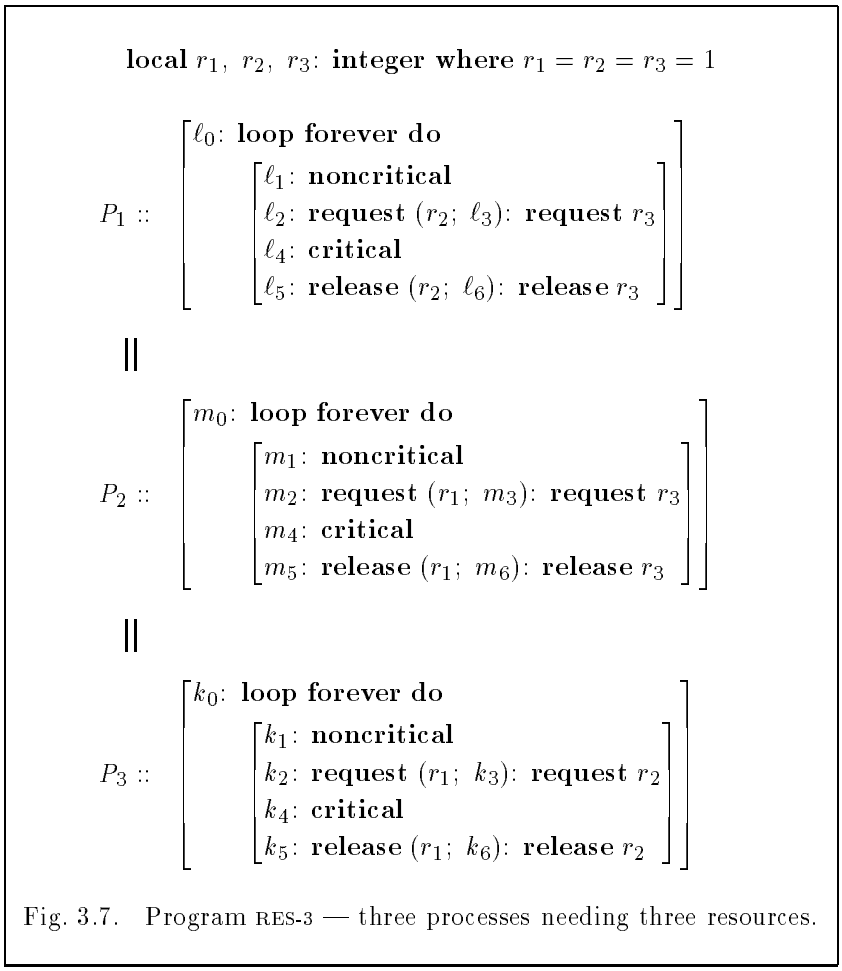


Fig. 3.7. Program RES-3 — three processes needing three resources.

In Section 2.3 of the SAFETY book, we established the safety property of mutual exclusion for program RES-3. The proof was based on the following invariants

$$\begin{aligned}
\chi_0\colon &\quad r_1 \geq 0 \ \wedge \ r_2 \geq 0 \ \wedge \ r_3 \geq 0 \\
\chi_1\colon &\quad r_1 + at_m_{3..5} + at_k_{3..5} = 1 \\
\chi_2\colon &\quad r_2 + at_\ell_{3..5} + at_k_{4..6} = 1 \\
\chi_3\colon &\quad r_3 + at_\ell_{4..6} + at_m_{4..6} = 1.
\end{aligned}$$

It is easy to see that these invariants ensure pairwise exclusion between the processes. For example, to show exclusion between P_1 and P_3, we examine the invariant χ_2, corresponding to the resource r_2 common to P_1 and P_3. From χ_0 and χ_2 we obtain

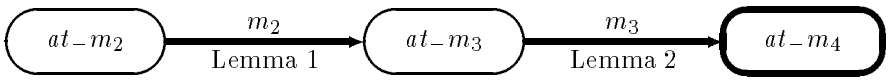
$$at\_\ell_4 + at\_k_4 \ \leq \ at\_\ell_{3..5} + at\_k_{4..6} \ \leq \ 1,$$

which shows that P_1 and P_3 cannot be at their critical sections at the same time.

Here we prove the response property of accessibility. Stated for P_2, it is given by

$$at\_m_2 \ \Rightarrow \ \diamondsuit \, at\_m_4$$

A proof of this property by rule CHAIN-F is presented in the following diagram



To complete the proof we need to show the enableness premise for m_2 and m_3, which is done in Lemmas 1 and 2.

Lemma 1 $\quad at\_m_2 \ \Rightarrow \ \diamondsuit(at\_m_2 \ \wedge \ r_1 > 0)$

Examining invariant χ_1 we see that, with P_2 being at m_2, the only way for r_1 to be nonpositive is that $r_1 = 0$ and P_3 is at $k_{3..5}$. We therefore have to follow P_3 until it exits $k_{3..5}$, at which point k_5 sets r_1 back to 1. This development is shown in the CHAIN-F diagram of Fig. 3.8.

Of the three helpful transitions identified in this diagram, immediate enableness of k_4 and k_5 follows from assertions at_k_4 and at_k_5 respectively. The situation is different with k_3 which also needs $r_2 > 0$ to proceed. Since k_3 is a compassionate transition, it is sufficient to prove eventual enableness, which is proven in Lemma 1.1.

Lemma 1.1 $\quad at\_m_2 \ \wedge \ at\_k_3 \ \wedge \ r_1 = 0 \ \Rightarrow \ \diamondsuit(at\_k_3 \ \wedge \ r_2 > 0)$

Examining invariant χ_2, we see that with P_3 at k_3, the only way for r_2 to be nonpositive is that $r_2 = 0$ and P_1 is at $\ell_{3..5}$. The diagram of Fig. 3.9 follows P_1

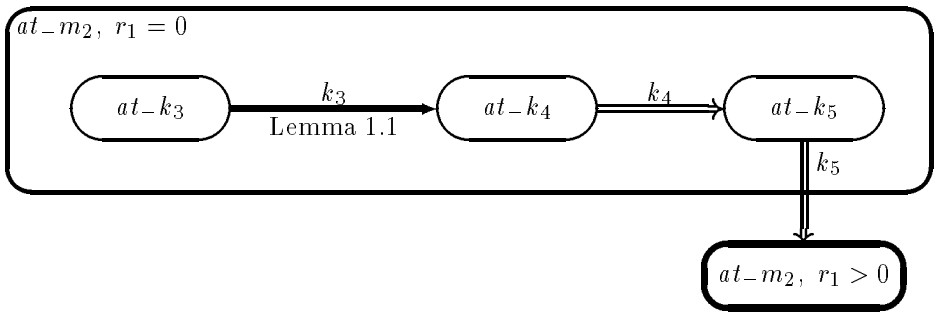


Fig. 3.8. CHAIN-F diagram for Lemma 1.

until it executes ℓ_5, at which point r_2 becomes positive. Notice that since P_2 is waiting at m_2 with $r_1 = 0$, and P_3 is waiting at k_3 with $r_2 = 0$, they are both disabled, and the only process that can save the day is P_1. The CHAIN diagram of Fig. 3.9 shows that indeed it does.

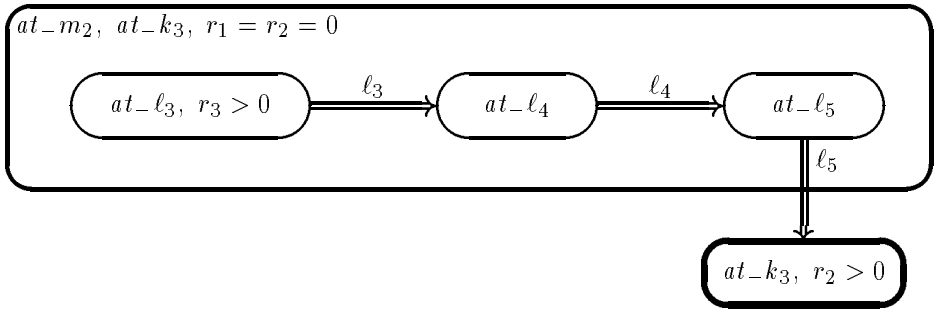


Fig. 3.9. CHAIN diagram for Lemma 1.1.

Note that due to χ_3, while P_1 is at ℓ_3 and P_2 is at m_2, r_3 is positive, which makes ℓ_3 immediately enabled. Thus, even though ℓ_3 is a compassionate transition we can prove for it immediate enableness.

Lemma 2 $\quad at\_m_3 \ \Rightarrow \ \diamondsuit(at\_m_3 \ \wedge \ r_3 > 0)$

This lemma proves the second enableness premise for the accessibility property. While P_2 is at m_3, the only way for r_3 to be nonpositive, according to χ_3, is that $r_3 = 0$, and P_1 is at $\ell_{4..6}$. The CHAIN diagram of Fig. 3.10 traces the progress of P_1 through $\ell_{4..6}$ until transition ℓ_6 increases r_3 to 1.

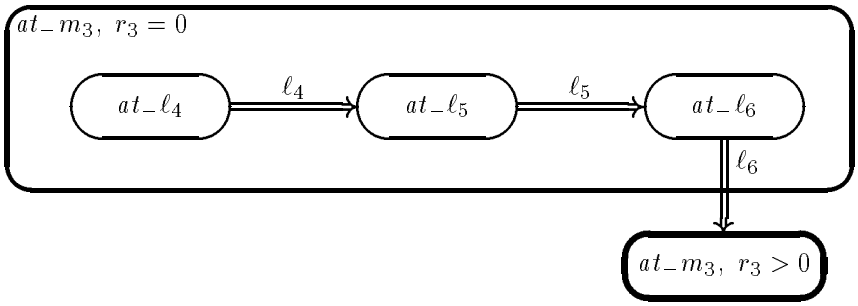


Fig. 3.10. CHAIN diagram for Lemma 2.

This concludes the proof of the lemmas, and establishes accessibility for P_2. Similarly, we can establish accessibility for P_1 and P_3. ∎

Recursiveness of the Fair Rules

The proof of accessibility for P_2 in the example above provides a second answer to the problem of recursiveness of the F-family of rules.

The recursiveness problem is that the enableness premise of the F-rules is a response formula, similar to the conclusions of these rules. The question, therefore, is how we escape circular reasoning trying to prove a formula using a rule in which one of the premises is not simpler than the conclusion.

The first answer we gave, which is also illustrated in the proof of Lemma 1, is that to prove a property by an F-rule, it is often possible to prove its eventual enableness premise F4 by J-rules. For example, the enableness premise for k_3, needed in the proof of Lemma 1, is proven in Lemma 1.1 by rule CHAIN-J (Fig. 3.9).

The second answer, which is illustrated in the complete proof of the accessibility property, is that even if we need to pursue several levels of proof, all of which use an F-rule, we are guaranteed that on each level we have to consider one process less than on the previous level.

Consider the proof of the accessibility property for program RES-3. It is mainly based on the progress of P_2 from m_2 to m_4. To complete the proof we need at m_2 an enableness premise that guarantees that the transition m_2 eventually becomes enabled. If it happens to be currently enabled, all the better. So the difficult case is to show that if P_2 is at m_2 and the transition at m_2 is currently *disabled*, it eventually becomes enabled. For that we use Lemma 1.

However, note that in all the subsequent situations considered in Lemma 1 or any of its subproofs, P_2 is disabled. This implies that, when dealing with this

case, we have one less process to consider. Lemma 1, recursively, uses an F-rule to follow the progress of P_3 from k_3 to k_6. In all situations arising throughout this progress, P_2 is continuously neutralized (frozen). Considering the situation of P_3 at k_3, we realize that the proof of another eventual enableness is called for, namely the eventual enableness of k_3. For that we use Lemma 1.1.

Since the interesting case for Lemma 1.1 is when k_3 is disabled, we enter this lemma with the assumption that two processes, P_2 and P_3 are disabled. Thus, Lemma 1.1 has only process P_1 to work with. And P_1 better help itself for all the enableness properties it needs, since P_2 and P_3 are paralyzed and cannot offer any help.

In general, a recursive rule or procedure is acceptable if we can identify a well-founded measure that decreases on any recursive invocation. The discussion above identifies this measure, in our case, as the number of processes which are still potentially enabled, since any additional level in the proof tree freezes an additional process. It is obvious that, when we reach a single remaining process, the property is valid only if it can be proven by the J-rules, since no other process is available to eventually improve the enableness status of transitions.

3.3 Compassion in Message Passing

The previous section considered compassionate transitions associated with semaphore statements. Here we consider communication statements, which are the other statements associated with compassion requirements. We present several examples of programs that use message passing and analyze their response properties, using rules RESP-C and CHAIN-F.

As examples, we present two algorithms for coordinating mutual exclusion by message passing which are based on compassion. We analyze these algorithms, and verify that they guarantee exclusion and individual accessibility.

Synchronous Message Passing

The first algorithm we consider is program MUX-SYNCH of Fig. 3.11. The program consists of two customer processes P_1, P_2, and an arbiter process A. The arbiter process accepts requests for entering the critical section from both processes. These requests are represented by T-messages on channels α_1, α_2. The arbiter chooses nondeterministically to communicate with one of them. This communication grants the selected customer process a permission to enter its critical section. After the customer exits from its critical section it returns this permission by sending an F-message on its α-channel to the arbiter that keeps waiting at k_2 or k_3, respectively, for this releasing communication.

We start by considering mutual exclusion. This can be established by the

local $\alpha_1,\ \alpha_2$: **channel of boolean**

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \alpha_1 \Leftarrow \text{T} \\ \ell_3\text{: } \textbf{critical} \\ \ell_4\text{: } \alpha_1 \Leftarrow \text{F} \end{array}\right] \end{array}\right]$$

$$\Big\|$$

$$A :: \left[\begin{array}{l} \textbf{local } y\text{: } \textbf{boolean} \\ k_0\text{: } \textbf{loop forever do} \\ \quad k_1\text{: } \left[\begin{array}{l} k_1^a\text{: } \alpha_1 \ \Rightarrow\ y;\ k_2\text{: } \alpha_1 \Rightarrow y \\ \quad \textbf{or} \\ k_1^b\text{: } \alpha_2 \ \Rightarrow\ y;\ k_3\text{: } \alpha_2 \Rightarrow y \end{array}\right] \end{array}\right]$$

$$\Big\|$$

$$P_2 :: \left[\begin{array}{l} m_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{noncritical} \\ m_2\text{: } \alpha_2 \Leftarrow \text{T} \\ m_3\text{: } \textbf{critical} \\ m_4\text{: } \alpha_2 \Leftarrow \text{F} \end{array}\right] \end{array}\right]$$

Fig. 3.11. Program MUX-SYNCH (mutual exclusion by synchronous communication).

following invariants

$$\chi_1\text{: } \quad at\_\ell_{3,4} \ \leftrightarrow\ at\_k_2$$

$$\chi_2\text{: } \quad at\_m_{3,4} \ \leftrightarrow\ at\_k_3.$$

From χ_1 and χ_2 we can infer the invariant

$$\neg(at\_\ell_{3,4} \ \wedge\ at\_m_{3,4}),$$

based on process A's inability to be simultaneously at k_2 and at k_3.

Next, we prove accessibility for P_1. Accessibility for P_1 is stated by

$$\underbrace{at\_\ell_2}_{p} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_3}_{q} \;.$$

It calls for a single application of rule RESP-C with

$$\varphi = p\colon \quad at\_\ell_2 \qquad \tau_h\colon \quad \langle \ell_2,\ k_1^a \rangle .$$

It is obvious that premises C1, C2, and C3 hold for this choice. Needing more attention is premise C4 requiring

$$\psi_1\colon \quad \underbrace{at\_\ell_2}_{\varphi} \;\Rightarrow\; \diamondsuit \Big(\cdots \;\vee\; \underbrace{at\_\ell_2 \;\wedge\; at\_k_1}_{En(\langle \ell_2, k_1^a \rangle)} \Big) \;.$$

This property is proved by rule CHAIN-J as shown in the CHAIN diagram of Fig. 3.12.

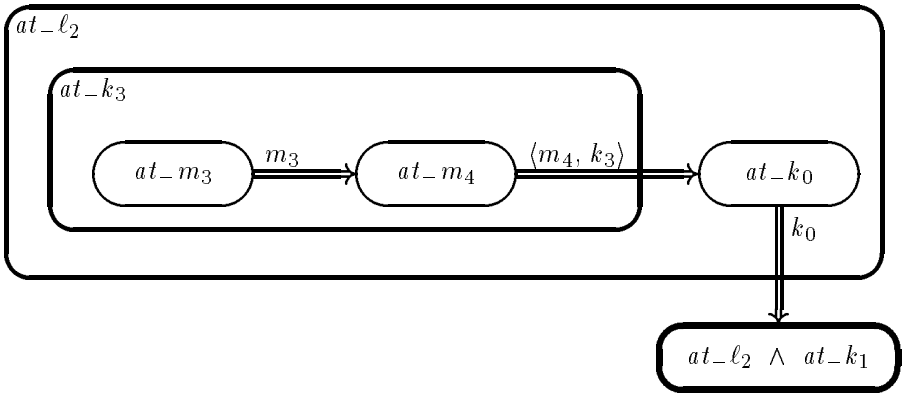


Fig. 3.12. CHAIN diagram for ψ_1.

The proof uses invariant χ_1 to infer that when P_1 is at ℓ_2, A cannot be at k_2, and invariant χ_2 to infer that when A is at k_3, P_2 can only be at m_3 or at m_4.

Accessibility for P_2 is proven in a completely symmetric way.

Asynchronous Message Passing

Program MUX-ASYNCH of Fig. 3.13 coordinates mutual exclusion by using asynchronous communication. The program uses channels which allow many readers but only one writer.

The arbiter process A starts by loading channel α with a single message, representing a permission to enter the critical section. The customer processes P_1 and P_2 compete on this single permission at locations ℓ_2 and m_2, respectively.

$$\textbf{local } \alpha,\ \beta_1,\ \beta_2\text{: } \textbf{channel } [..] \textbf{ of boolean}$$

$$P_1 :: \left[\begin{array}{l} \textbf{local } x_1\text{: } \textbf{boolean} \\ \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \alpha \Rightarrow x_1 \\ \ell_3\text{: } \textbf{critical} \\ \ell_4\text{: } \beta_1 \Leftarrow \text{T} \end{array}\right] \end{array}\right]$$

$$\Big\|$$

$$A :: \left[\begin{array}{l} \textbf{local } y\text{: } \textbf{boolean} \\ k_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} k_1\text{: } \alpha \Leftarrow \text{T} \\ k_2\text{: } \left[\begin{array}{l} k_2^a\text{: } \beta_1 \Rightarrow y \\ \quad \textbf{or} \\ k_2^b\text{: } \beta_2 \Rightarrow y \end{array}\right] \end{array}\right] \end{array}\right]$$

$$\Big\|$$

$$P_2 :: \left[\begin{array}{l} \textbf{local } x_2\text{: } \textbf{boolean} \\ m_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{noncritical} \\ m_2\text{: } \alpha \Rightarrow x_2 \\ m_3\text{: } \textbf{critical} \\ m_4\text{: } \beta_2 \Leftarrow \text{T} \end{array}\right] \end{array}\right]$$

Fig. 3.13. Program MUX-ASYNCH (mutual exclusion by asynchronous communication).

Whoever succeeds in reading the message from channel α proceeds to enter the critical section. On finishing, this process sends back a release message via β_1 or β_2, respectively. Meanwhile, process A keeps waiting at k_2 to collect this release message. On receiving the message, process A loops back and reloads channel α with a new permission.

To prove exclusion, we establish the invariant

$$\chi_1\colon \quad at\_k_{0,1} + |\alpha| + at\_\ell_{3,4} + at\_m_{3,4} + |\beta_1| + |\beta_2| \;=\; 1.$$

Clearly, this invariant implies that $at\_\ell_{3,4} + at\_m_{3,4} \leq 1$.

Accessibility for P_1 is expressed by

$$\underbrace{at\_\ell_2}_{p} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_3}_{q}\ .$$

This can be proven by rule RESP-C, taking

$$\varphi = p\colon\ at\_\ell_2 \qquad \tau_h\colon\ \ell_2.$$

The first three premises C1–C3 obviously hold. It only remains to show premise C4, the eventual enableness of ℓ_2. This is expressed by

$$\psi_2\colon \quad \underbrace{at\_\ell_2}_{\varphi} \;\Rightarrow\; \diamondsuit\Big(\cdots \;\vee\; \underbrace{at\_\ell_2 \;\wedge\; |\alpha| > 0}_{En(\ell_2)}\Big)\ .$$

A CHAIN-J proof of this property is presented in the CHAIN diagram of Fig. 3.14. The proof uses invariant χ_1 to infer that when $|\alpha| = 0$ then either A is at $k_{0,1}$, or it is at k_2 with $|\beta_2| = 1$, or at k_2 with $|\beta_2| = 0$ and P_2 at $m_{3,4}$.

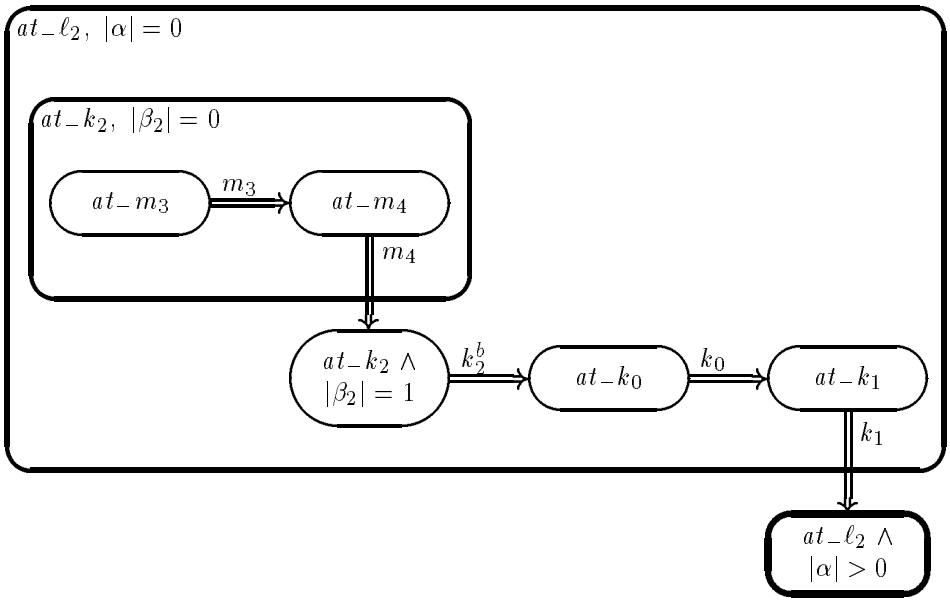


Fig. 3.14. CHAIN diagram for ψ_2.

3.4 Modeling Eventual Reliability

The preceding sections used compassion as an abstraction for a protocol that is implemented either by special hardware or by a software procedure. Our two main examples were semaphore and communication statements.

There are additional contexts and applications where compassion provides a useful abstraction. We introduce here one such application, in which compassion is used to model *eventual reliability* of a hardware component of the system.

Modeling Faulty Channels

Consider program REL-TRANS of Fig. 3.15. This program consists of process P_1 that sends messages $1, 2, \ldots$ on channel α, and process P_2 that reads these messages from channel α into y.

local α: **channel** [1..] **of integer**

$$P_1 :: \left[\begin{array}{l} \textbf{local } x\text{: } \textbf{integer where } x = 0 \\ \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } x := x + 1 \\ \ell_2\text{: } \alpha \Leftarrow x \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} \textbf{local integer } y \\ m_0\text{: } \textbf{loop forever do} \\ \quad m_1\text{: } \alpha \Rightarrow y \end{array}\right]$$

Fig. 3.15. Program REL-TRANS (reliable transmission).

It is a simple matter to verify that y assumes the values $1, 2, \ldots$.

Such verification depends, however, on the implicit assumption that channel α is reliable. This assumption is reflected in the definition of the transitions associated with the communication statements. For program REL-TRANS, these definitions imply that execution of ℓ_2: $\alpha \Leftarrow x$ appends x to the end of list α, and the message remains in α until removed by execution of m_1: $\alpha \Rightarrow y$.

In some cases, we may wish to study situations in which channel α is not reliable and may either lose some messages or transmit them with corrupted values. There are two possible approaches to modeling such faulty channels.

The first approach associates with communication statements over faulty channels different types of transitions. These transitions should represent the potentially faulty behavior of such channels.

Another approach, which is the one adopted here, is not to change the association of transitions to communication statements but to modify the program,

replacing each *send* statement by an appropriate selection statements. We will illustrate this approach for two types of faults.

Modeling Lossy Channels

The first fault we model is that of a channel that may lose some of its messages. Program LOSSY-TRANS of Fig. 3.16 represents a modified version of program REL-TRANS, which takes into account the possibility of lost messages.

$$\textbf{local } \alpha\text{: } \textbf{channel } [1..] \textbf{ of integer}$$

$$P_1 :: \left[\begin{array}{l} \textbf{local } x\text{: } \textbf{integer where } x = 0 \\ \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } x := x + 1 \\ \ell_2\text{: } \left[\begin{array}{l} \ell_2^g\text{: } \alpha \Leftarrow x \\ \quad \textbf{or} \\ \ell_2^s\text{: } \textbf{skip} \end{array}\right] \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} \textbf{local integer } y \\ m_0\text{: } \textbf{loop forever do} \\ \quad m_1\text{: } \alpha \Rightarrow y \end{array}\right]$$

Fig. 3.16. Program LOSSY-TRANS (transmission with lossy channels).

Program LOSSY-TRANS is obtained by replacing statement $\ell_2\text{: } \alpha \Leftarrow x$ of program REL-TRANS by the selection statement

$$[\ell_2^g\text{: } \alpha \Leftarrow x] \quad \textbf{or} \quad [\ell_2^s\text{: } \textbf{skip}].$$

This implies that, whenever process P_1 is ready to send x over channel α, there is a nondeterministic choice between taking the *good transition* ℓ_2^g which actually sends x on α, or taking the *skip* transition ℓ_2^s which does nothing and models a loss of the message. Thus, computations of program LOSSY-TRANS faithfully represent the situation that some messages are lost, and therefore y may assume only a subset of the values $1, 2, \ldots$.

Such a transformation is intended to model operations on an abstract faulty asynchronous channel $\widehat{\alpha}$. Taking either ℓ_2^g or ℓ_2^s represents sending x on $\widehat{\alpha}$: Taking ℓ_2^g represents a successful transmission on $\widehat{\alpha}$, while taking ℓ_2^s represents a loss of a message. Transition m_1 represents receiving a message from $\widehat{\alpha}$.

It is important to realize that the faulty channel $\widehat{\alpha}$ modeled by this transformation is *eventually reliable*. This means that, if infinitely many messages are sent, channel $\widehat{\alpha}$ cannot lose *all* of them from a certain point on. This excludes computations in which infinitely many messages are sent on $\widehat{\alpha}$ but all but finitely many of them are lost.

Eventual reliability of our model follows from the requirement of compassion for the communication statement ℓ_2^g: $\alpha \Leftarrow x$. If P_1 attempts to send a message infinitely many times (i.e., visits ℓ_2 infinitely many times), then transition ℓ_2^g is enabled infinitely many times and, by compassion, must eventually be taken. It follows that, eventually, some x will be placed on channel α, and process P_2 will read it.

Modeling Corrupt Channels

Another type of fault that may be displayed by unreliable (but eventually reliable) channels is corruption of messages. This fault transmits a sent message but corrupts its value. In program CORR-TRANS of Fig. 3.17 we model a channel that may lose some messages and corrupt some of the others but eventually also transmits some correct messages. The selection statement in ℓ_2 contains three substatements. Statement ℓ_2^g represents faithful transmission; ℓ_2^s represents a loss of a message; and statement ℓ_2^c represents transmission of a corrupted value: 0 instead of x.

local α: **channel** [1..] **of integer**

$$
P_1 :: \left[\begin{array}{l}
\textbf{local } x\textbf{: integer} \\
\quad \textbf{where } x = 0 \\
\ell_0\textbf{: loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\textbf{: } x := x + 1 \\
\quad \left[\begin{array}{l} \ell_2^g\textbf{: } \alpha \Leftarrow x \\ \quad \textbf{or} \\ \ell_2\textbf{: } \ell_2^s\textbf{: skip} \\ \quad \textbf{or} \\ \quad \ell_2^c\textbf{: } \alpha \Leftarrow 0 \end{array}\right]
\end{array}\right]
\end{array}\right]
\quad \| \quad
P_2 :: \left[\begin{array}{l}
\textbf{local integer } y \\
m_0\textbf{: loop forever do} \\
\quad \left[\begin{array}{l}
m_1\textbf{: } y := 0 \\
m_2\textbf{: while } y = 0 \textbf{ do} \\
\quad m_3\textbf{: } \alpha \Rightarrow y
\end{array}\right]
\end{array}\right]
$$

Fig. 3.17. Program CORR-TRANS (transmission with corrupting channels).

The receiving process P_2 in this program is aware of the possibility of corruption, and is programmed to keep receiving until it gets a nonzero message.

Again, due to compassion, the modeled channel is eventually reliable, excluding computations in which, from a certain point on, all messages are either lost or corrupted. Note, that a side effect of this modeling is that it also implies that some messages must eventually be correpted.

This approach to modeling faulty channels is also applicable to synchronous communication, using the same transformations.

Example (alternating bit protocol)

Consider program ALTER of Fig. 3.18. This program is designed to ensure reliable communication over a faulty channel. We assume a particular type of fault, by which each message is either transmitted intact or corrupted in a noticeable way, i.e., no messages are lost. The program uses synchronous channel *trans* to transmit messages from P_1 to P_2, and synchronous channel *ack* to send acknowledgements from P_2 to P_1.

Each message sent over channel *trans* is a record with three components: pt, d, and ct. As we will see, pt serves as an identification of the message, d is the data that should be reliably transmitted from P_1 to P_2, and ct is a boolean field indicating whether the message has been transmitted correctly (correct transmission) or corrupted by the faulty channel.

A message sent over channel *ack* is also a record with the two components *pa* and *ca*. Field *pa* identifies the message that is acknowledged. Boolean field *ca* indicates whether the message is transmitted correctly (correct acknowledgement) or corrupted.

The possibility of corruption of a transmitted message is represented in Fig. 3.18 as a nondeterministic but fair (due to compassion) choice of transmitting the intended message with an additional bit of either T or F value. The choice of T for the additional bit represents a successful transmission, and then the other fields of the message are reliably valid. The case of F for the added bit, represents a failed transmission, in which the other fields of the message are unreliable, except for the F bit which identifies the message as a corrupt one.

It is important to realize that Fig. 3.18 contains a mixture of a program for the protocol and additional transitions such as ℓ_1^c and m_4^c, which model the potentially erratic behavior of the eventually reliable channels. The protocol, which is the program that will run in an actual application, has the statements

$$\ell_1\colon\ trans \Leftarrow \langle S,\ X[i],\ \mathrm{T}\rangle \qquad m_4\colon\ ack \Leftarrow \langle R,\ \mathrm{T}\rangle$$

at locations ℓ_1 and m_4. The *selection* statements appearing at these locations in Fig. 3.18 are intended to model the behavior of *trans* and *ack* as eventually reliable channels.

- **The Protocol**

The main idea in the protocol is that it maintains an acknowledgement channel, called *ack*, in which the receiver reports whether the message has arrived correctly, or has been corrupted. In both cases, the message in the *ack* channel is a request for a next transmission. In the case that the last message has been correctly received, the request is for the next message. In the case that the last

$$
\begin{array}{ll}
\textbf{in} & X \;\; : \textbf{array}\ [1..]\ \textbf{of integer} \\
\textbf{local} & i,\ j \;\; : \textbf{integer where } i = j = 1 \\
& s,\ r,\ S,\ R,\ gt,\ ga\colon \textbf{boolean where } S = R = \textsc{t} \\
& trans\colon \textbf{channel of } \langle pt\colon \textbf{boolean},\ d\colon \textbf{integer},\ ct\colon \textbf{boolean}\rangle \\
& ack \;\; : \textbf{channel of } \langle pa\colon \textbf{boolean},\ ca\colon \textbf{boolean}\rangle \\
\textbf{out} & Y \;\; : \textbf{array}\ [1..]\ \textbf{of integer}
\end{array}
$$

$$
P_1 :: \left[\begin{array}{l}
\ell_0\colon \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_1\colon & \left[\begin{array}{l} \ell_1^g\colon\ trans \ \Leftarrow\ \langle S,\ X[i],\ \textsc{t}\rangle \\ \quad \textbf{or} \\ \ell_1^c\colon\ trans \ \Leftarrow\ \langle -,\ -,\ \textsc{f}\rangle \end{array}\right] \\
\ell_2\colon & ack \ \Rightarrow\ \langle r,\ ga\rangle \\
\ell_3\colon & \textbf{if } ga \ \wedge\ r \neq S \textbf{ then} \\
& \quad \ell_4\colon\ (i,\ S) := (i+1,\ \neg S)
\end{array}\right]
\end{array}\right]
$$

$$
\Big\|
$$

$$
P_2 :: \left[\begin{array}{l}
m_0\colon \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
m_1\colon & trans \ \Rightarrow\ \langle s,\ Y[j],\ gt\rangle \\
m_2\colon & \textbf{if } gt \ \wedge\ s = R \textbf{ then} \\
& \quad m_3\colon\ (j,\ R) := (j+1,\ \neg R) \\
m_4\colon & \left[\begin{array}{l} m_4^g\colon\ ack \ \Leftarrow\ \langle R,\ \textsc{t}\rangle \\ \quad \textbf{or} \\ m_4^c\colon\ ack \ \Leftarrow\ \langle -,\ \textsc{f}\rangle \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Fig. 3.18. Program ALTER (alternating bit protocol with synchronous communication).

message has been incorrectly received, the request is for a retransmission of the last message. Corruption is also possible in the acknowledgement message. Consequently, it can be that the last message has been correctly received, but the sender is unaware of this fact, and keeps retransmitting the old message until a correct acknowledgement arrives. This possibility is also handled by the protocol.

The interesting point about this protocol is that, under the above assumptions about the characteristics of the fault, only one bit of information, represented by a boolean value, is necessary to identify whether the sender is transmitting the

old message or a new message, and whether the receiver is asking for a transmission of a new message or for retransmission of an old one.

- **The Parity Bit**

It is possible to consider a simpler but less space-efficient algorithm in which fields *pt* and *pa* contain the full sequence number of the message with each message and each acknowledgement, i.e., i and j, respectively. In this simpler algorithm, the receiver knows immediately which message it is receiving, and the sender immediately recognizes which message is being acknowledged.

However, since messages are transmitted and acknowledged in sequence, and none are lost (by assumption), it is clear that the possibilities for confusion about which message is being transmitted or acknowledged are limited and can be resolved by one bit. When P_2 is awaiting message $X[j]$, and some message is successfully received, it can either be $X[j]$ or $X[j-1]$. Similarly, when P_1 is awaiting a request for $i+1$ (which indirectly acknowledges message i), it can receive a request either for $i+1$ or for i. The latter is a request for a retransmission of the old message.

Therefore, it is sufficient to enclose with each message the *parity bit* of the full sequence number (T for odd numbers, F for even ones). This will resolve the ambiguity between j and $j-1$, and between $i+1$ and i.

Indeed, initially $i = j = 1$ and the corresponding parity bits, S in P_1 (the sender parity bit) and R in P_2 (the receiver parity bit) are both T. Process P_1, when sending $X[i]$, includes S with it, which is the parity bit of i. When P_2 receives a message in m_1, it checks at m_2 for corruption and whether the sequence number of the received message, represented by its parity bit, matches P_2's expectations of receiving $X[j]$. If the message is not corrupt ($gt = \text{T}$) and the received parity bit matches R, then this is the expected message.

Consequently, both j and R are advanced in m_3 to the next sequence number and its corresponding parity bit. Otherwise j and R retain their old values.

Next, P_2 sends in m_4 a request for the next transmission which names $X[j]$ as the requested message. This is represented by sending R, the parity bit of j. In the case j has not been advanced, this is a request for retransmission of $X[j]$. In case j has been advanced, this is a request for the next message.

In ℓ_2, P_1 reads this request. In ℓ_3 it checks for a correct reception of the request message, and whether the request is for the next message, tested by $r \neq S$. If it is a request for a new message, then both i and S are advanced in ℓ_4. Otherwise they stay the same.

- **A Safety Property**

An appropriate safety property we wish to establish is that

> Messages approved by the receiver are identical to the original mes-

sages sent by the sender.

For convenience, we have represented the list of messages transmitted by the sender as an input array X, and the list of messages approved by the receiver as an output array Y. These arrays are not an integral part of the algorithm, and are only used for the purpose of the specification and verification of the program.

The main assertion, whose invariance implies the safety property described above, is

$$\psi_1: \quad Y[1..j-1] \;=\; X[1..j-1].$$

This assertion states that the segment of the array Y consisting of positions $1, \ldots, j-1$ is identical to the corresponding segment of the array X. Since at any point in the program for P_2, j denotes the sequence number of the message that P_2 attempts to receive, we can assume that it has already approved all the preceding messages, currently stored in $Y[1..j-1]$.

The proof of invariant ψ_1 uses the following auxiliary invariants:

$$I_1: \quad (S = R \;\wedge\; i = j) \;\vee\; (S \neq R \;\wedge\; i+1 = j)$$

$$I_2: \quad at\_\ell_3 \;\wedge\; ga \;\rightarrow\; r = R$$

$$I_3: \quad at\_m_2 \;\wedge\; gt \;\rightarrow\; s = S$$

$$I_4: \quad at\_\ell_4 \;\rightarrow\; S \neq R$$

$$I_5: \quad at\_m_3 \;\rightarrow\; S = R$$

$$I_6: \quad at\_\ell_2 \;\leftrightarrow\; at\_m_{2..4}.$$

In **Problem 3.1**, the reader is requested to prove the invariant ψ_1.

An alternative proof of invariant ψ_1 may be based on the assertion diagram of Fig. 3.19 which summarizes the states that may appear in a computation of program ALTER.

- **Proving Progress**

As previously explained, the nondeterministic selections at ℓ_1 and m_4 are not statements that actually exist in the protocol, but are intended to represent the unreliability of channels *trans* and *ack*. They present the possibility that instead of receiving a good record, containing $\langle S, X[i], \mathrm{T}\rangle$ as prepared by P_1, process P_2 may receive a faulty record with unpredictable values for the first two fields. The only alleviating assumption we make is that P_2 can recognize whether it has received a valid or faulty message. This is represented by the last field in the message, which is T for a valid message and F for a faulty one. The same assumptions are made about channel *ack*, which explains the nondeterministic selection at m_4.

The main response property that is natural for this problem is that any message that P_1 attempts to send is eventually correctly received by P_2. In view

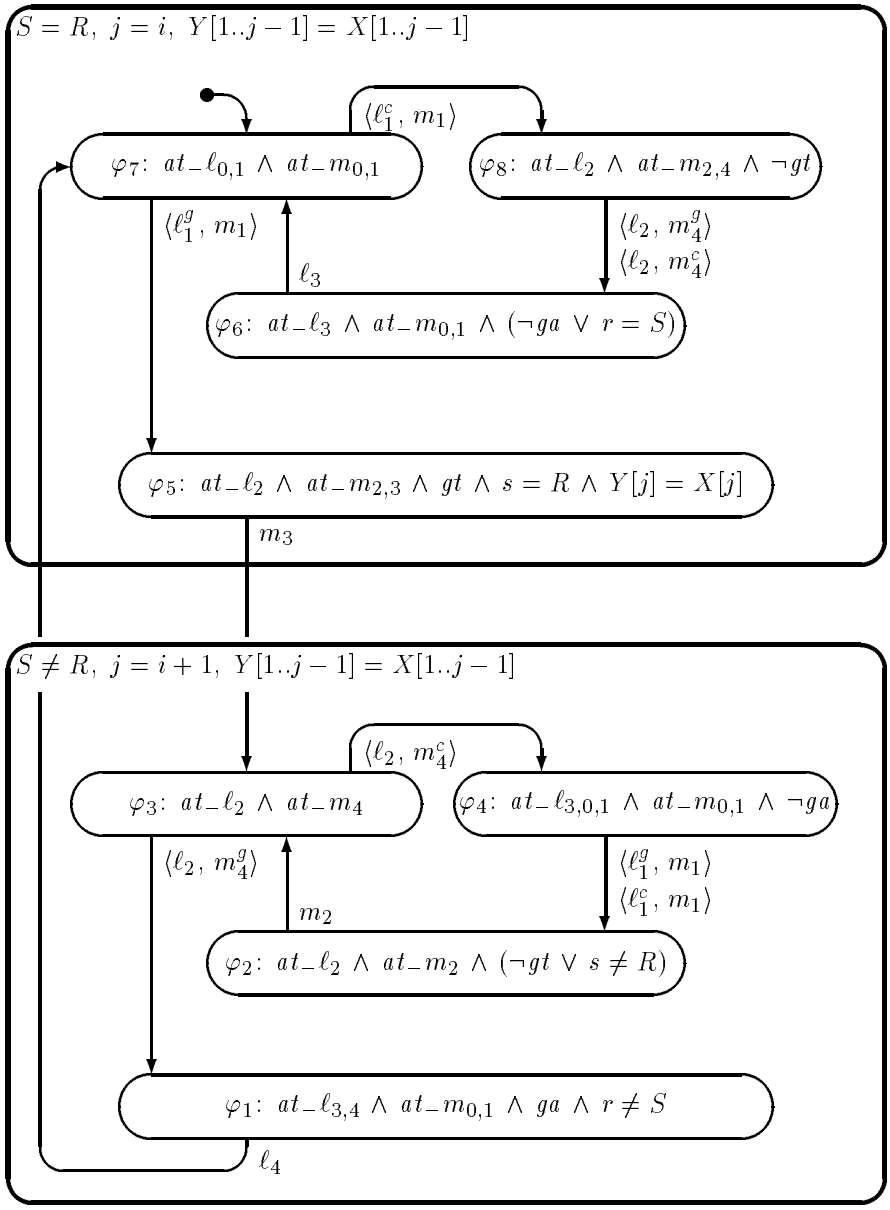


Fig. 3.19. Possible states of program ALTER.

of the safety property proved above, it is sufficient to show that j eventually increases. This can be expressed by the response property

$$\psi_2: \quad j = u \;\Rightarrow\; \diamondsuit(j = u + 1),$$

using rigid variable u to record the current value of j.

By invariant I_1, either $j = i$ or $j = i + 1$. Considering the two cases, it is possible to decompose ψ_2 into the two response properties

$$\psi_3: \quad i = j = u \;\Rightarrow\; \diamondsuit(i + 1 = j = u + 1)$$

$$\psi_4: \quad i + 1 = j = u \;\Rightarrow\; \diamondsuit(i = j = u)$$

and prove them separately.

Consulting Fig. 3.19, we see that the behavior of the program is partitioned into two alternating phases. The first phase, occupying the top of the diagram, is characterized by $j = i$. In this phase process P_1 repeatedly attempts to send message $X[i]$ to P_2. A successful attempt will lead to φ_5, which some steps later terminates the phase. Any failed attempt causes the system to traverse the cycle φ_7, φ_8, and φ_6 and return to φ_7. This cycle cannot repeat indefinitely since each visit to φ_7 involves a state in which the compassionate transition $\langle \ell_1^g, m_1 \rangle$ is enabled. By compassion, the system must eventually get to φ_5 and then proceed to φ_3. Response formula ψ_3 states the termination of the upper phase.

The second phase, occupying the bottom half of the diagram, is characterized by $j = i + 1$. In this phase, process P_2 attempts to acknowledge correct reception of message $X[j] = X[i + 1]$. Each failed attempt traverses the cycle φ_3, φ_4, φ_2, returning to φ_3 once. By compassion, eventually $\langle \ell_2, m_4^g \rangle$ must be taken, leading to φ_1 and the eventual termination of this phase, as required by ψ_1.

To summarize, it cannot be that one of the processes sends infinitely many messages and all of them, from a certain point, arrive corrupted. This implies that, since P_1 persistently sends message $m[u]$ until correctly acknowledged, eventually this message will arrive uncorrupted and P_2 will increase j to $u + 1$. Following this correct arrival, process P_2 persistently sends an acknowledgement for $m[u]$ (requesting $m[i + 1]$) until a correct $m[u + 1]$ arrives. Eventually, one of these acknowledgements will arrive correctly and P_1 will increase i to $u + 1$.

We proceed to substantiate this intuition by formally proving ψ_2. As suggested, we first establish

$$\psi_3: \quad (i = j = u) \;\Rightarrow\; \diamondsuit(i + 1 = j = u + 1).$$

The proof of this property, using rule CHAIN-F, is presented in the CHAIN-F diagram of Fig. 3.20.

It is easy to verify premises F1–F4 of rule CHAIN-F for assertions φ_1 and φ_2, appearing in Fig. 3.20.

It only remains to check eventual enableness of $\langle \ell_1^g, m_1 \rangle$ from φ_3-states.

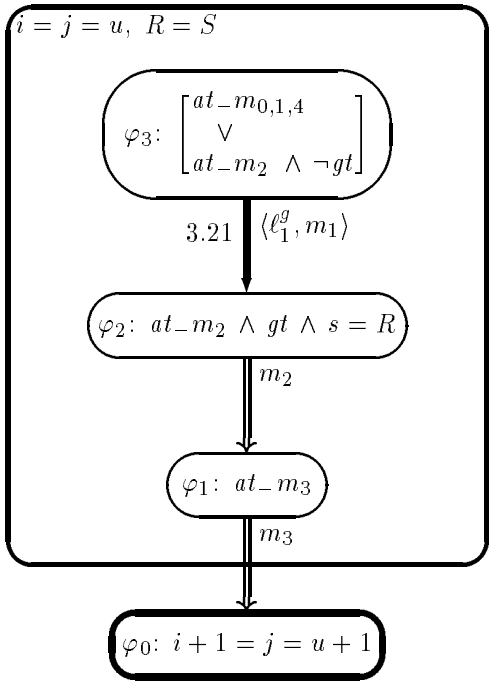


Fig. 3.20. Top-level CHAIN-F diagram for ψ_3.

This means that we have to show

$$at\_m_{0,1,4} \ \vee \ (at\_m_2 \ \wedge \ \neg gt) \ \ \Rightarrow \ \ \diamondsuit(at\_\ell_1 \ \wedge \ at\_m_1).$$

This is proven by CHAIN-J as presented in the CHAIN diagram of Fig. 3.21. The proof uses I_6 to infer that when P_2 is at m_2 or at m_4, P_1 must be at ℓ_2. It also uses I_2 and I_3 to infer that, while P_2 is at m_1 and $S = R$, P_1 can be only at ℓ_0, at ℓ_1, or at ℓ_3 with $\neg ga \vee r = S$.

This concludes the proof of the property

$$\psi_3: \quad i = j = u \ \Rightarrow \ \diamondsuit(i + 1 = j = u + 1).$$

The second response property ensures that eventually P_1 will also increase counter i

$$\psi_4: \quad i + 1 = j = u \ \ \Rightarrow \ \ \diamondsuit(i = j = u).$$

The proof of this property is presented in the CHAIN-F diagram of Fig. 3.22.

Eventual enableness of $\langle \ell_4, m_5 \rangle$ is proven by the CHAIN diagram of Fig. 3.23.

Combining properties ψ_3 and ψ_4, we obtain the main response property of

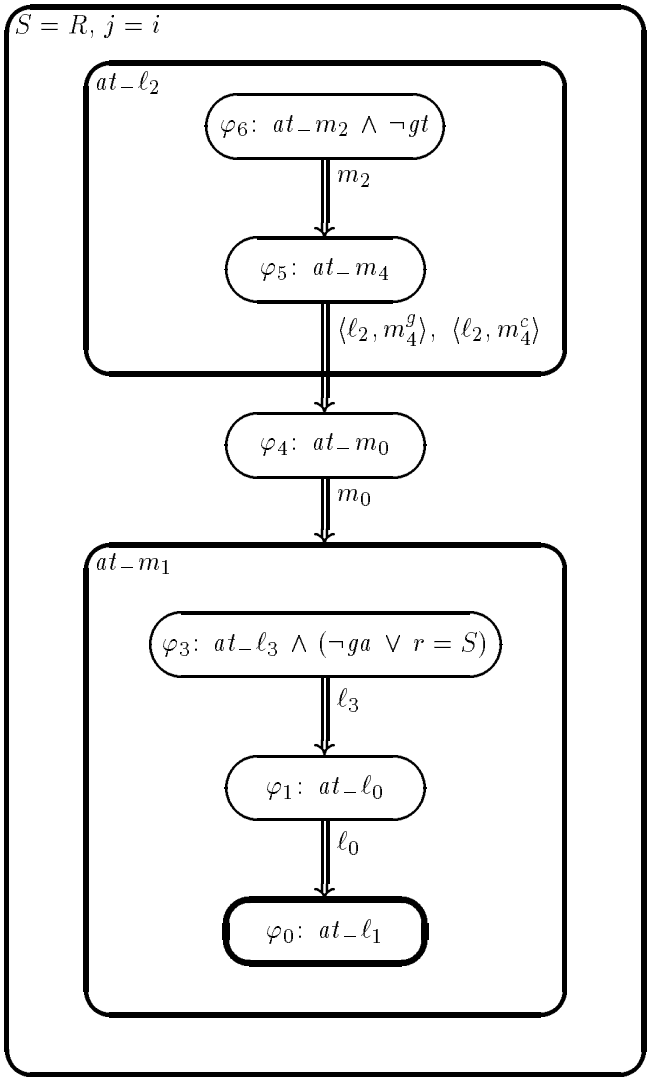


Fig. 3.21. CHAIN diagram for eventual enableness of $\langle \ell_1^g,\ m_1 \rangle$.

program ALTER

$$\psi_2\colon \quad j = u \ \Rightarrow\ \diamondsuit(j = u + 1). \qquad \blacksquare$$

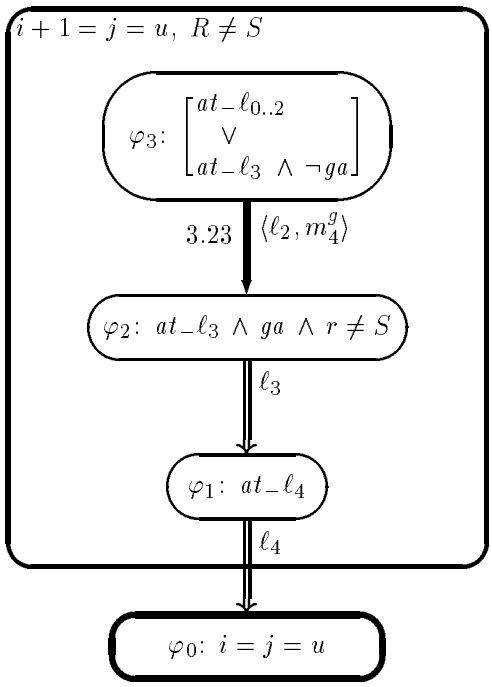


Fig. 3.22. Top-level CHAIN-F diagram for ψ_4.

3.5 Well-Founded Rule

The next rule we consider is the fair version of rule WELL-J (Fig. 1.26). This version can be obtained by replacing the temporal premise W2 of rule WELL (Fig. 1.23) by the appropriate premises of rule RESP-C for the case that τ_i is compassionate and the appropriate premises of rule RESP-J for a just τ_i. This leads to rule WELL-F (Fig. 3.24).

In applications of the rule, it is sufficient to establish premise FW2 for all $\tau \neq \tau_i$, since FW2 for $\tau = \tau_i$ is implied by FW3.

A verification diagram corresponding to rule WELL-F is called a RANK-F diagram.

Example (Producer Consumer)

Consider program PROD-CONS of Fig. 3.5. Using rule CHAIN-F, we proved accessibility for the producer, i.e., $at\_\ell_2 \Rightarrow \diamondsuit\, at\_\ell_4$. However, the main response property of this program is that any value produced by the producer is eventually consumed by the consumer. This property can be written as

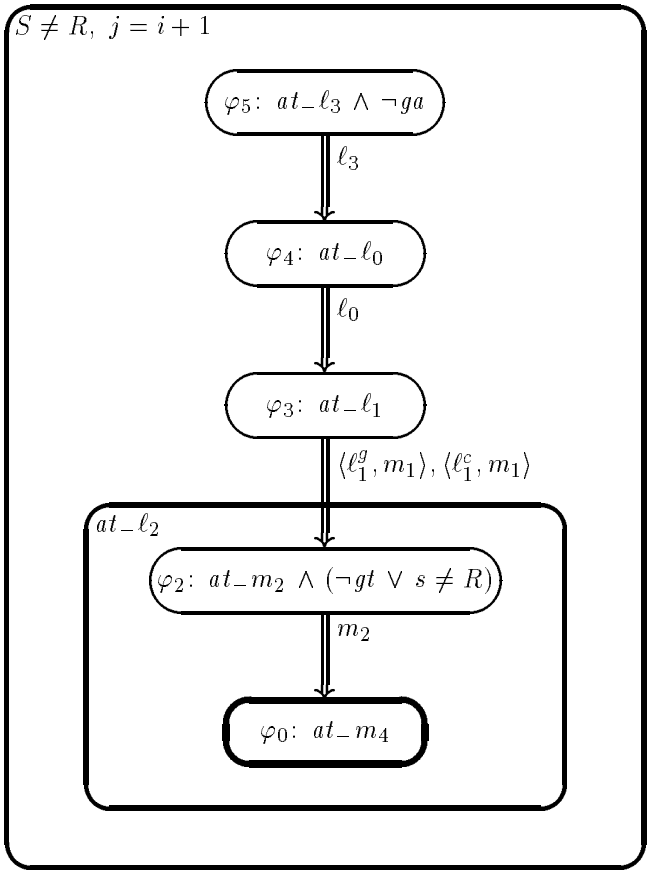


Fig. 3.23. CHAIN diagram for eventual enableness of $\langle \ell_2, m_4^g \rangle$.

$$prod(u) \ \Rightarrow \ \diamondsuit\, cons(u),$$

where we define

$$prod(u)\text{: } at\_\ell_2 \ \wedge \ x = u,$$

$$cons(u)\text{: } at\_m_6 \ \wedge \ y = u.$$

These two state formulas characterize, respectively, the situation that process *Prod* has just produced the value u at statement ℓ_1, and the situation that process *Cons* is about to consume the value u at statement m_6. Substituting the definitions for *prod* and *cons*, we obtain

$$\underbrace{at\_\ell_2 \ \wedge \ x = u}_{p} \ \Rightarrow \ \diamondsuit \underbrace{at\_m_6 \ \wedge \ y = u}_{q} \ .$$

For assertions p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J} \cup \mathcal{C}$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m$: $\mathcal{S} \mapsto \mathcal{A}$

FW1. $p \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j$

FW2. $\rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \left[\begin{array}{l} \bigvee_{j=0}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j) \\ \vee \;\; (\varphi'_i \;\wedge\; \delta_i = \delta'_i) \end{array}\right]$ for every $\tau \in \mathcal{T}$

FW3. $\rho_{\tau_i} \;\wedge\; \varphi_i \;\rightarrow\; \bigvee_{j=0}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j)$

} for $i = 1, \ldots, m$

FW4. If $\tau_i \in \mathcal{C}$, then

CW4. $\varphi_i \;\Rightarrow\; \diamondsuit\big(\neg\varphi_i \;\vee\; En(\tau_i)\big)$

Otherwise

JW4. $\varphi_i \;\rightarrow\; En(\tau_i)$

$p \;\Rightarrow\; \diamondsuit\, q$

Fig. 3.24. Rule WELL-F (well-founded rule under fairness).

We break the proof into three lemmas.

Lemma A $\quad at\_\ell_2 \;\wedge\; x = u \;\;\Rightarrow\;\; \diamondsuit(u \in b)$

This lemma states that if u is produced, it eventually will be placed in buffer b. The proof of this lemma closely resembles the proof of accessibility that we have proven before, and therefore will be omitted.

Lemma B $\quad u \in b \;\;\Rightarrow\;\; \diamondsuit(at\_m_4 \;\wedge\; y = u)$

This lemma claims that if message u is somewhere in the buffer, then eventually it will be read by process *Cons* and placed in y. The proof of this lemma is presented below.

Lemma C $at\_m_4 \ \wedge \ y = u \ \Rightarrow \ \diamondsuit(at\_m_6 \ \wedge \ y = u)$

This lemma completes the journey of u, by following *Cons* from m_4 to m_6. It is provable by a simple application of CHAIN-J, and the proof is omitted.

- *Proof of Lemma B*

  The proof of Lemma B is assisted by the invariants

  $$\chi_0\colon \quad r \geq 0 \ \wedge \ ne \geq 0 \ \wedge \ nf \geq 0$$
  $$\chi_1\colon \quad r + at\_\ell_{4,5} + at\_m_{3,4} = 1$$
  $$\chi_2\colon \quad ne + nf + at\_\ell_{3..6} + at\_m_{2..5} = N.$$

  The proof is based on rule WELL-F and is presented in the RANK-F diagram of Fig. 3.25. It uses intermediate assertions $\varphi_1, \ldots, \varphi_7$. As the well-founded domain we take the lexicographic product $\mathbb{N} \times [0..7]$. The ranking function for assertion φ_i, $i = 1, \ldots, 7$, is given by

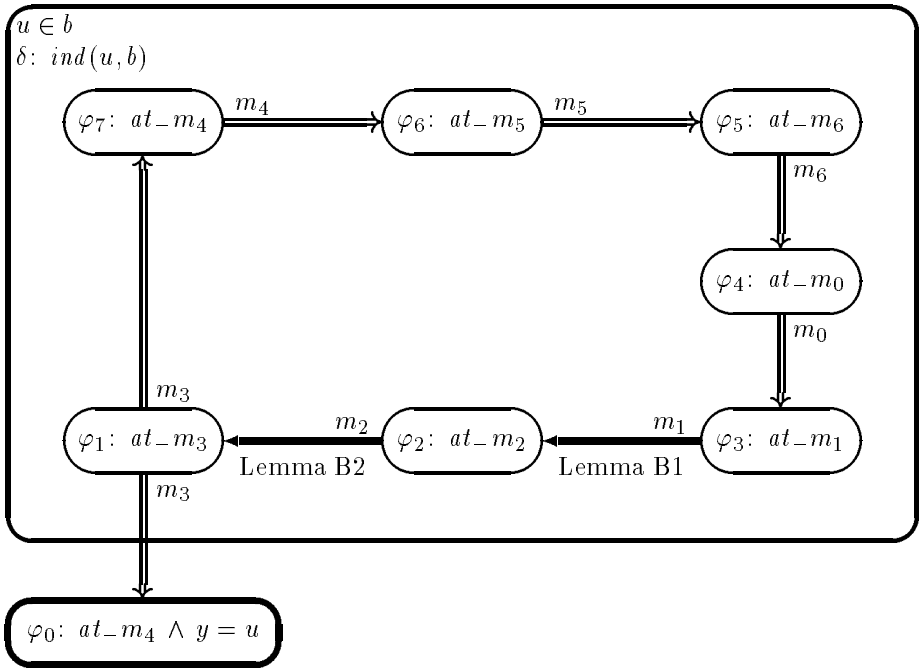


Fig. 3.25. RANK-F diagram for eventual transmission of messages.

$$\delta_i\colon \quad \big(ind(u, b),\ i\big),$$

where $ind(u,b)$ is defined as

$$ind(u,\,b) \;=\; min\{\, i \geq 1 \mid b[i] = u \,\}.$$

That is, $ind(u,b)$ is the smallest index i, $i \geq 1$, such that $b[i] = u$. Clearly, the value of $ind(u,b)$ keeps decreasing as *Cons* removes elements from b. We rely on the obvious implication

$$u \in b \;\;\rightarrow\;\; \exists i \geq 1\colon\; b[i] = u.$$

We adopt the convention by which if $u \notin b$ then $ind(u,b) = 0$. It follows that $ind(u,b) \geq 0$ in all cases.

It is not difficult to see that premises FW1–FW3 are satisfied by the assertions appearing in the diagram. In particular, we observe that no transition, except for m_3, can change the value of $ind(u,b)$, under the assumption of $u \in b$. This also holds for ℓ_4. Also, it is not difficult to see that the full ranking functions $\delta_i\colon \big(ind(u,b), i\big)$ decrease on any helpful transition, unless the goal $\varphi_0\colon\; at\_m_4 \;\wedge\; y = u$ is achieved. The transition m_3, either achieves φ_0 in the case that $ind(u,b) = 1$, or maintains $u \in b$, with $ind(u,b') < ind(u,b)$. All other helpful transitions preserve $ind(u,b)$, and lead from the rank $\big(ind(u,b), i\big)$ to the rank $\big(ind(u,b), i-1\big)$, for some i, $i > 0$.

Let us consider premise FW4. Most of the helpful transitions are just transitions for which we can easily establish JW4. The exceptions are the compassionate transitions m_1 and m_2 for which we will establish CW4.

To establish CW4, it is sufficient to show the eventual enableness of m_2 and m_3. These are proven by the two following lemmas.

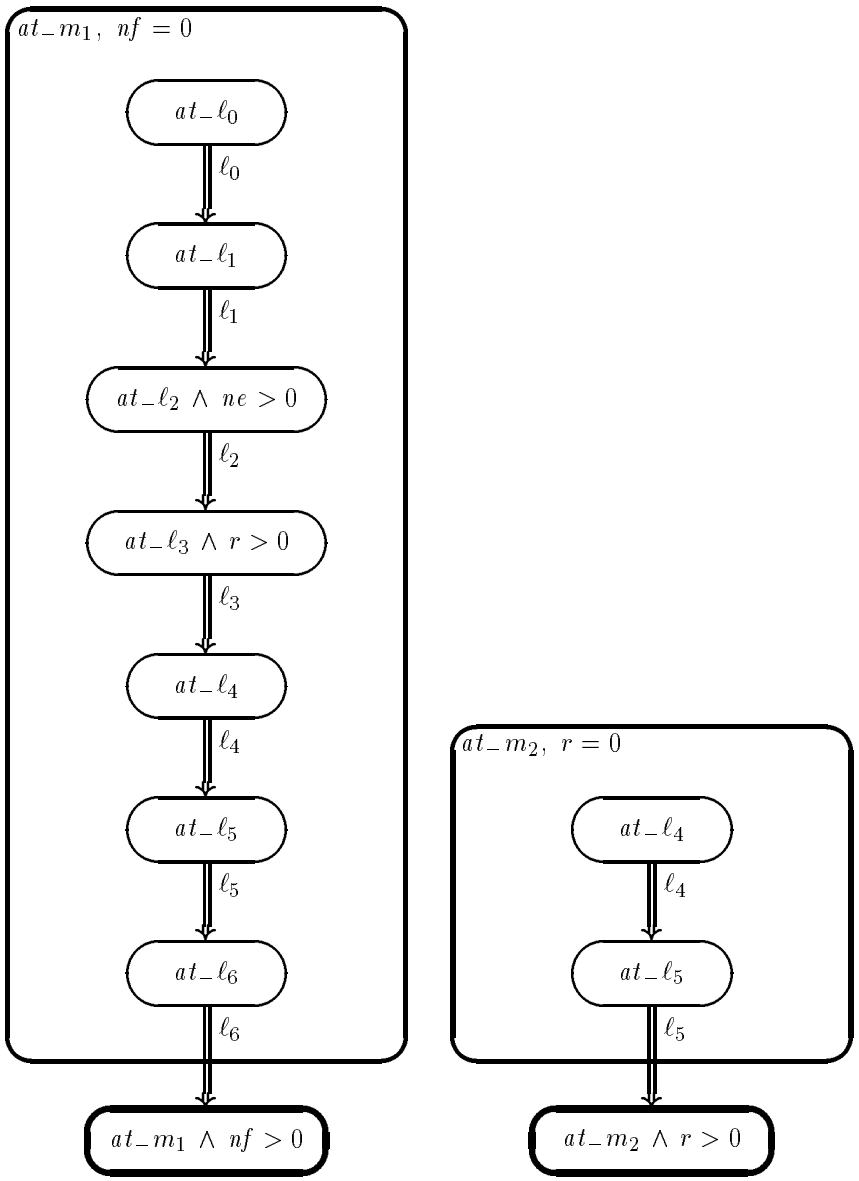
Lemma B1 $\quad at\_m_1 \;\Rightarrow\; \diamondsuit(at\_m_1 \;\wedge\; nf > 0)$

The lemma is proven by rule CHAIN-J, as presented in the CHAIN diagram of Fig. 3.26(a). Clearly if nf is not positive it must equal 0. We use invariant χ_2 and the fact that P_2 is at m_1 and $nf = 0$, to deduce that while P_1 is at ℓ_2, $ne = N > 0$. Similarly, we use χ_1 to deduce that while P_1 is at ℓ_3 and P_2 is at m_1, r is positive.

Lemma B2 $\quad at\_m_2 \;\Rightarrow\; \diamondsuit(at\_m_2 \;\wedge\; r > 0)$

The lemma is proven by rule CHAIN-J, as presented in the CHAIN diagram of Fig. 3.26(b). According to χ_0 and χ_1, if r is nonpositive while P_2 is at m_2, then r must equal 0, and P_1 must reside at ℓ_4 or at ℓ_5.

This concludes the proof of Lemma B, as well as the proof of the response property that ensures that any value produced by *Prod* eventually gets consumed by *Cons*. ∎



(a) Proof of Lemma B1. (b) Proof of Lemma B2.

Fig. 3.26. CHAIN diagrams.

3.6 The Dining Philosophers Problem

In Section 2.3 of the SAFETY book, we presented the dining philosophers problem and studied three proposed solutions to the problem given by programs DINE (Fig. 2.15), DINE-CONTR (Fig. 2.16), and DINE-EXCL (Fig. 2.17). In that section, we established the safety property of chopstick exclusion, by which no two adjacent processes (which share a chopstick) can reside in their critical section at the same time.

Here, we wish to study the progress property of these solutions, which is the property of accessibility. This property states that every philosopher who becomes hungry (exits its noncritical section) will eventually eat (enter its critical section).

As shown in Section 2.3 of the SAFETY book, program DINE may deadlock, which means that no accessibility property is valid for it. It remains to consider the two other programs: DINE-CONTR and DINE-EXCL.

Accessibility for Program DINE-CONTR

In Fig. 3.27 we reproduce program DINE-CONTR, first presented in Fig. 2.16 of the SAFETY book.

For this program, we established the following invariant assertions:

$\varphi_0[j]$: $c[j] \geq 0$, for every $j \in [1..M]$

$\varphi_1[j]$: $at\_\ell_{4..6}[j] + at\_\ell_{3..5}[j+1] + c[j+1] = 1$, for every $j \in [1..M-2]$

φ_2 : $at\_\ell_{4..6}[M-1] + at\_\ell_{4..6}[M] + c[M] = 1$

φ_3 : $at\_\ell_{3..5}[M] + at\_\ell_{3..5}[1] + c[1] = 1$.

We will show that the accessibility property

$$A[j]: \quad at\_\ell_2[j] \Rightarrow \diamondsuit\, at\_\ell_4[j]$$

is valid over program DINE-CONTR for each $j \in [1..M]$.

The accessibility property $A[j]$ can be decomposed into the two properties:

$$A_{2,3}[j]: \quad at\_\ell_2[j] \Rightarrow \diamondsuit\, at\_\ell_3[j]$$

$$A_{3,4}[j]: \quad at\_\ell_3[j] \Rightarrow \diamondsuit\, at\_\ell_4[j].$$

Obviously, the validity of $A_{2,3}[j]$ and $A_{3,4}[j]$ implies the validity of $A[j]$, over program DINE-CONTR. We intend to establish properties $A_{2,3}$ and $A_{3,4}$ in the following order:

- First, we will establish $A_{2,3}[j]$ for every $j \in [2..M-1]$.

in M: **integer where** $M \geq 2$
local c : **array** $[1..M]$ **of integer where** $c = 1$

$$\Big\|_{j=1}^{M-1} P[j] :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \textbf{request } c[j] \\ \ell_3\text{: } \textbf{request } c[j+1] \\ \ell_4\text{: } \textbf{critical} \\ \ell_5\text{: } \textbf{release } c[j] \\ \ell_6\text{: } \textbf{release } c[j+1] \end{array}\right] \end{array}\right]$$

$$\|$$

$$P[M] :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \textbf{request } c[1] \\ \ell_3\text{: } \textbf{request } c[M] \\ \ell_4\text{: } \textbf{critical} \\ \ell_5\text{: } \textbf{release } c[1] \\ \ell_6\text{: } \textbf{release } c[M] \end{array}\right] \end{array}\right]$$

Fig. 3.27. Program DINE-CONTR: solution with one contrary philosopher.

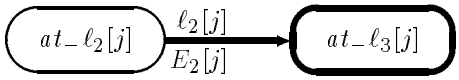
- Then, we establish by induction on decreasing j the properties $A_{3,4}[j]$ for $j = M, M-1, \ldots, 1$.
- Finally, we establish $A_{2,3}[1]$ and $A_{2,3}[M]$.

Proving $A_{2,3}[j]$ for $j \in [2..M-1]$

Let $j \in [2..M-1]$. The property

$$A_{2,3}[j]: \quad at\_\ell_2[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_3[j]$$

is proven by the following CHAIN-F diagram



The proof relies on the compassionate transition $\ell_2[j]$. As required for valid

diagrams involving compassionate transitions, we have to establish the eventual enableness of $\ell_2[j]$ stated by the formula

$$E_2[j]: \quad at\_\ell_2[j] \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_2[j] \;\wedge\; c[j] > 0}_{En(\ell_2[j])} \;.$$

Property $E_2[j]$ is proven by the CHAIN diagram of Fig. 3.28.

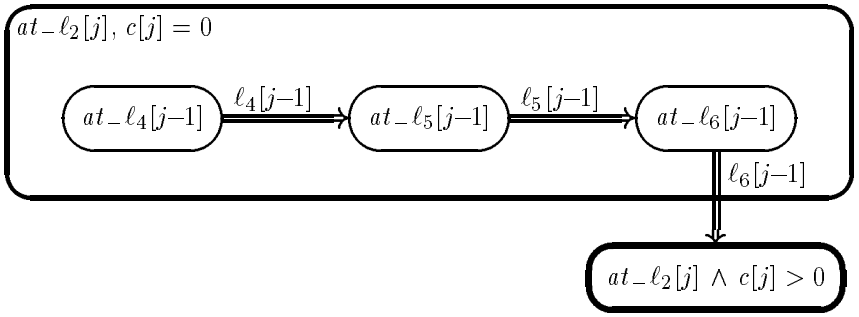


Fig. 3.28. CHAIN diagram for $E_2[j]$.

This proof relies on assertion $\varphi_0[j]$: $c[j] \geq 0$ to infer that $\neg(c[j] > 0)$ implies $c[j] = 0$. Then, we use assertion

$$\varphi_1[j-1]: \quad at\_\ell_{4..6}[j-1] \;+\; at\_\ell_{3..5}[j] \;+\; c[j] \;=\; 1$$

to infer that $at\_\ell_2[j] \wedge c[j] = 0$ implies $at\_\ell_{4..6}[j-1]$.

Proving $A_{3,4}[j]$ for $j = M, M-1, \ldots, 1$

Next, we establish the property

$$A_{3,4}[j]: \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_4[j]$$

by induction on decreasing $j = M, M-1, \ldots, 1$.

- *Induction Basis*

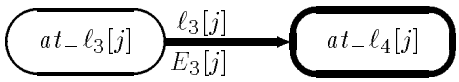
  For the induction basis, we prove

  $$A_{3,4}[j]: \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_4[j]$$

  for $j \in \{M-1, M\}$. We can establish $A_{3,4}$ for both $M-1$ and M in one proof because both processes compete on $c[M]$ at location ℓ_3 and release $c[M]$ at location ℓ_6.

  Let j denote one of the indices in $\{M-1, M\}$, and k denote the *other* index. Thus, if $j = M-1$ then $k = M$ and if $j = M$ then $k = M-1$.

Property $A_{3,4}[j]$ is proven by the following CHAIN-F diagram:



The diagram requires establishing the eventual enableness of $\ell_3[j]$ given by

$$E_3[j]: \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit\big(at\_\ell_3[j] \;\wedge\; c[M] > 0\big)$$

This is established by the CHAIN diagram of Fig. 3.29.

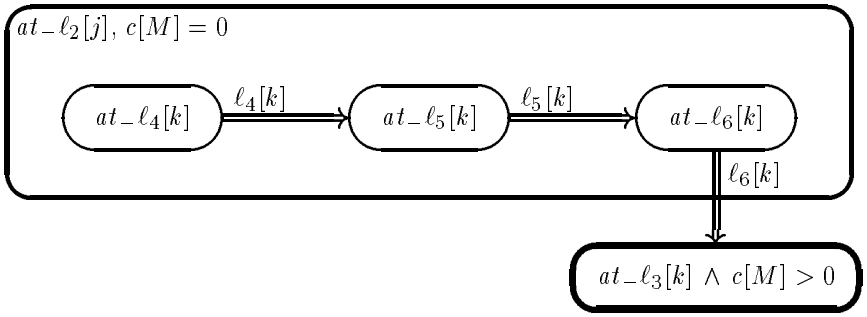


Fig. 3.29. CHAIN diagram for $E_3[j]$.

The proof uses $\varphi_0[M]$: $c[M] \geq 0$ to infer $c[M] = 0$ from $\neg\big(c[M] > 0\big)$ and the invariant

$$\varphi_2: \quad at\_\ell_{4..6}[M-1] \;+\; at\_\ell_{4..6}[M] \;+\; c[M] \;=\; 1$$

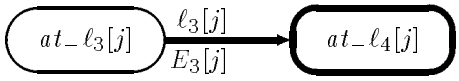
to infer $at\_\ell_{4..6}[M-1]$ from $at\_\ell_3[M] \wedge c[M] = 0$, and to infer $at\_\ell_{4..6}[M]$ from $at\_\ell_3[M-1] \wedge c[M] = 0$.

- *Induction Step*

We show that $A_{3,4}[j+1]$ implies $A_{3,4}[j]$ for each $j \in [1..M-2]$. Assume that $A_{3,4}[j+1]$ has been established. Property

$$A_{3,4}[j]: \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_4[j]$$

is proven by the following CHAIN-F diagram:



The required eventual enableness condition

$$E_3[j]: \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit\big(at\_\ell_3[j] \;\wedge\; c[j+1] > 0\big)$$

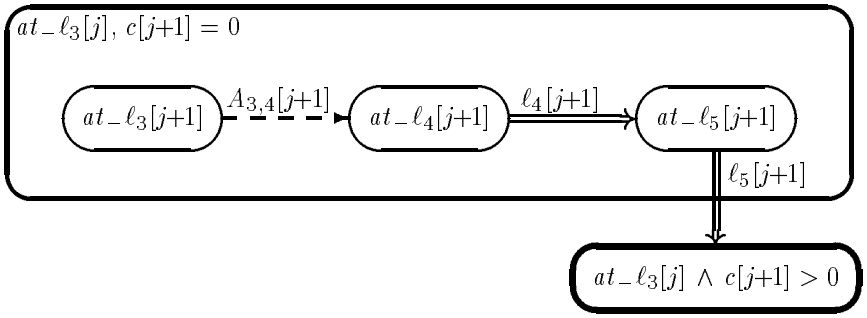


Fig. 3.30. CHAIN diagram for $E_3[j]$ (for the inductive step).

is proven by the CHAIN diagram of Fig. 3.30.

The proof uses assertion $\varphi_0[j+1]$: $c[j+1] \geq 0$ to infer $c[j+1] = 0$ from $\neg(c[j+1] > 0)$ and assertion

$$\varphi_1[j]: \quad at\_\ell_{4..6}[j] \; + \; at\_\ell_{3..5}[j+1] \; + \; c[j+1] \; = \; 1$$

to infer $at\_\ell_{3..5}[j+1]$ from $at\_\ell_3[j] \wedge c[j+1] = 0$.

It uses the assumption $A_{3,4}[j+1]$ to ensure that if $P[j+1]$ is at ℓ_3 it will eventually arrive to ℓ_4. In the diagram, this is represented by the dashed edge labeled by $A_{2,3}[j]$.

This concludes the proof of $A_{3,4}[j]$ for all $j \in [1..M]$.

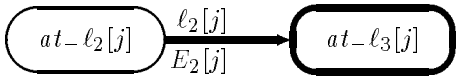
Proving $A_{2,3}[j]$ for $j \in \{1, M\}$

Finally, we prove

$$A_{2,3}[j]: \quad at\_\ell_2[j] \; \Rightarrow \; \diamondsuit \, at\_\ell_3[j]$$

for $j \in \{1, M\}$. Again, it is possible to present a single proof for these two indices since both $P[1]$ and $P[M]$ compete for $c[1]$ at location ℓ_2 and release $c[1]$ at location ℓ_5. As before, let j denote one of the two indices in $\{1, M\}$ and k denote the other index.

The proof of $A_{2,3}[j]$ is established by the following diagram:



The proof is completed by establishing the eventual enableness of $\ell_2[j]$, given by

$$E_2[j]: \quad at\_\ell_2[j] \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_2[j] \;\wedge\; c[1] > 0}_{En(\ell_2[j])} \;.$$

This property is proved by the CHAIN diagram of Fig. 3.31.

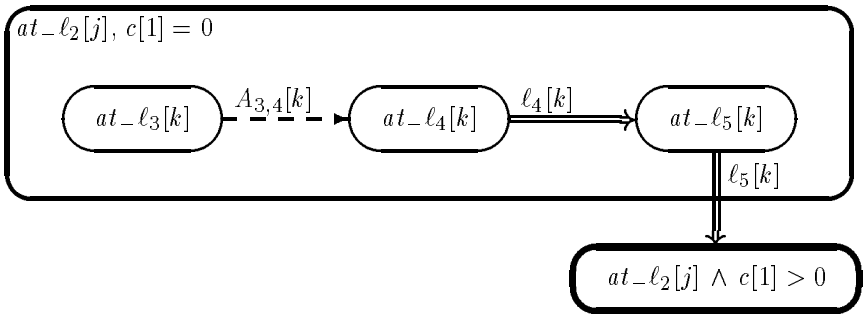


Fig. 3.31. CHAIN diagram for $E_2[j]$ for $j \in \{1, M\}$.

The proof uses assertion $\varphi_0[1]$: $c[1] \geq 0$ to infer $c[1] = 0$ from $\neg\big(c[1] > 0\big)$. It uses assertion

$$\varphi_3: \quad at\_\ell_{3..5}[M] \;+\; at\_\ell_{3..5}[1] \;+\; c[1] \;=\; 1$$

to infer $at\_\ell_{3..5}[M]$ from $at\_\ell_2[1] \wedge c[1] \;=\; 0$, and to infer $at\_\ell_{3..5}[1]$ from $at\_\ell_2[M] \wedge c[1] = 0$.

This concludes the proof of accessibility for all processes of program DINE-CONTR.

Accessibility for Program DINE-EXCL

In Fig. 3.32 we reproduce program DINE-EXCL, first presented in Fig. 2.17 of the SAFETY book. In contrast to program DINE-CONTR in which process $P[M]$ has a different program than all other processes, processes in program DINE-EXCL have identical programs. Symmetry is broken in program DINE-EXCL by the semaphore r which admits at most $M-1$ processes to the location range $\ell_{3..8}$.

In Section 2.3 of the SAFETY book, we established for program DINE-EXCL the following invariants

$\varphi_0[j]$: $c[j] \;\geq\; 0$ for all $j \in [1..M]$

$\varphi_1[j]$: $at\_\ell_{5..7}[j] \;+\; at\_\ell_{4..6}[j \oplus_M 1] \;+\; c[j \oplus_M 1] \;=\; 1$ for all $j \in [1..M]$

$\varphi_2 \;:\; r \;\geq\; 0$

$\varphi_3 \;:\; N_{3..8} \;+\; r \;=\; M-1.$

$$
\begin{array}{ll}
\textbf{in} & M\colon \textbf{integer where } M \geq 2 \\
\textbf{local } c & \colon \textbf{array } [1..M] \textbf{ integer where } c = 1 \\
r & \colon \textbf{integer where } r = M-1
\end{array}
$$

$$
\Big\|_{j=1}^{M} P[j] :: \left[\begin{array}{l}
\ell_0\colon \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\colon \textbf{noncritical} \\
\ell_2\colon \textbf{request } r \\
\ell_3\colon \textbf{request } c[j] \\
\ell_4\colon \textbf{request } c[j \oplus_M 1] \\
\ell_5\colon \textbf{critical} \\
\ell_6\colon \textbf{release } c[j] \\
\ell_7\colon \textbf{release } c[j \oplus_M 1] \\
\ell_8\colon \textbf{release } r
\end{array}\right]
\end{array}\right]
$$

Fig. 3.32. Program DINE-EXCL: one philosopher stays out.

Here we estabish the accessibility property

$$A[j]\colon \quad at\_\ell_2[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_5[j].$$

As before, we break the accessibility property into three steps

$$A_{2,3}[j]\colon \quad at\_\ell_2[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_3[j]$$
$$A_{3,4}[j]\colon \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_4[j]$$
$$A_{4,5}[j]\colon \quad at\_\ell_4[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_5[j].$$

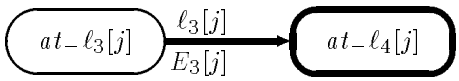
We establish $A_{3,4}[j]$ first, followed by $A_{4,5}[j]$, followed by $A_{2,3}[j]$.

Proving $A_{3,4}[j]$

The response property

$$A_{3,4}[j]\colon \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit\, at\_\ell_4[j]$$

is proven by the following CHAIN-F diagram:



Eventual enableness of compassionate transition $\ell_3[j]$ is stated by

$$E_3[j]\colon \quad at\_\ell_3[j] \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_3[j] \;\wedge\; c[j] > 0}_{En(\ell_3[j])}$$

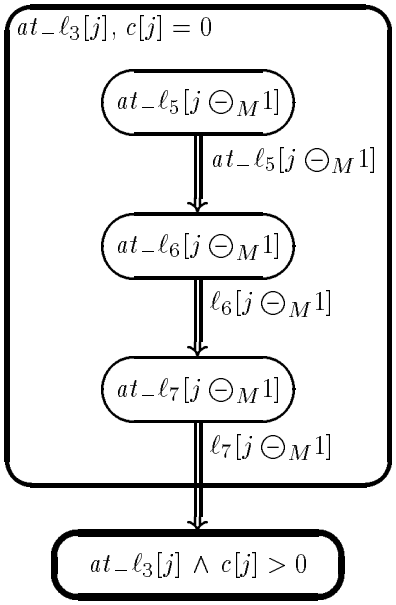


Fig. 3.33. CHAIN diagram of $E_3[j]$ (for program DINE-EXCL).

and is proven in the CHAIN diagram of Fig. 3.33.

The proof uses $\varphi_0[j]$: $c[j] \geq 0$ to infer $c[j] = 0$ from $\neg\big(c[j] > 0\big)$ and assertion

$$\varphi_1[j \ominus_M 1]: \quad at\_\ell_{5..7}[j \ominus_M 1] + at\_\ell_{4..6}[j] + c[j] = 1$$

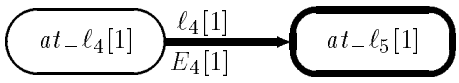
to infer $at\_\ell_{5..7}[j \ominus_M 1]$ from $at\_\ell_3[j] \wedge c[j] = 0$.

Proving $A_{4,5}[j]$

Since processes in program DINE-EXCL are fully symmetric it is sufficient to prove $A_{4,5}$ for a particular value of j, say $j = 1$

$$A_{4,5}[1]: \quad at\_\ell_4[1] \ \Rightarrow \ \diamondsuit\, at\_\ell_5[1].$$

The proof of $A_{4,5}[1]$ is presented in the following CHAIN-F diagram:



The required eventual enableness of $\ell_4[1]$ is stated by

$$E_4[1]: \quad at\_\ell_4[1] \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_4[1] \ \wedge \ c[2] > 0}_{En(\ell_4[1])}$$

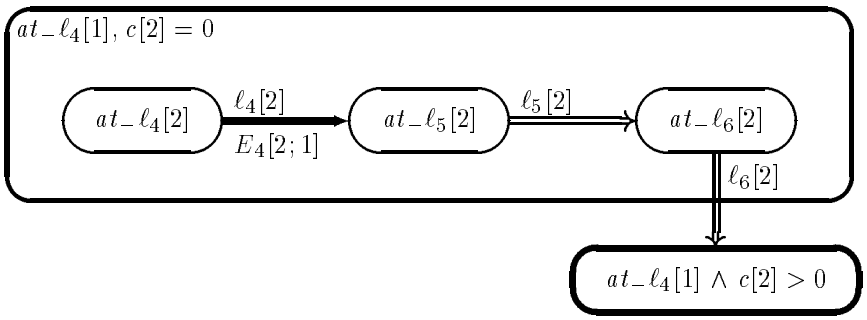


Fig. 3.34. CHAIN-F diagram of $E_4[1]$.

and can be proven by the CHAIN-F diagram of Fig. 3.34.

In this proof we use $\varphi_0[2]$ and $\varphi_1[1]$ to infer that $at\_\ell_4[1] \wedge \neg(c[2] > 0)$ implies $at\_\ell_{4..6}[2]$. The eventual enableness proof obligation for $\ell_4[2]$ is given by

$$E_4[2;1]: \quad at\_\ell_4[1] \ \wedge \ at\_\ell_4[2] \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_4[2] \ \wedge \ c[3] > 0}_{En(\ell_4[2])} .$$

We denote it by $E_4[2;1]$ because it claims the eventual enableness of $\ell_4[2]$ while $P[1]$ is also at ℓ_4.

Obviously, a proof of $E_4[2;1]$ will analyze the locations of process $P[3]$ and rely on an eventual enableness requirement given by

$$E_4[3;1,2]: \quad at\_\ell_4[1] \ \wedge \ at\_\ell_4[2] \ \wedge \ at\_\ell_4[3] \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_4[3] \ \wedge \ c[4] > 0}_{En(\ell_4[3])} .$$

Assume we constructed this sequence of verification diagrams reaching process $P[M-2]$. For this process, we need to establish

$$E_4[M-2;1,2,\ldots,M-3]: \quad at\_\ell_4[1] \ \wedge \ \cdots \ \wedge \ at\_\ell_4[M-2] \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_4[M-2] \ \wedge \ c[M-1] > 0}_{En(\ell_4[M-2])} .$$

This property can be established by the CHAIN-F diagram of Fig. 3.35.

The eventual enableness requirement is given by

$$E_4[M-1;1,\ldots,M-2]: \quad at\_\ell_4[1] \ \wedge \ \cdots \ \wedge \ at\_\ell_4[M-1] \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_4[M-1] \ \wedge \ c[M] > 0}_{En(\ell_4[M-1])} .$$

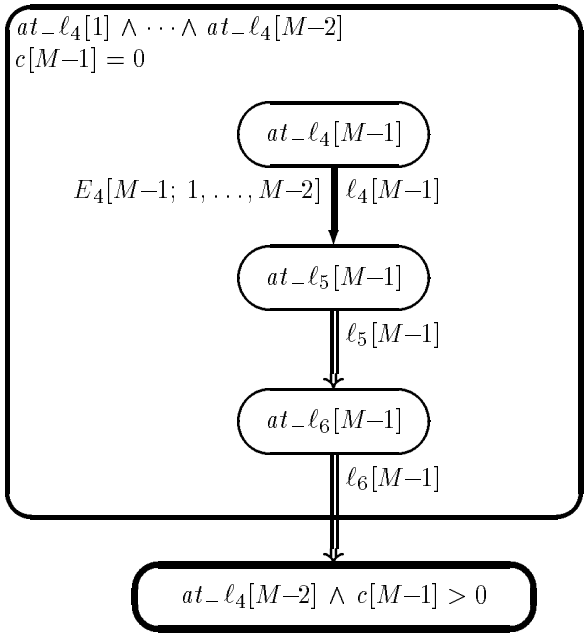


Fig. 3.35. CHAIN-F diagram of $E_4[M-2; 1, 2, \ldots; M-3]$.

However, in view of φ_2 and φ_3, $N_{3..8} \leq M-1$, which means that $at\_\ell_4[1] \wedge \cdots \wedge at\_\ell_4[M-1]$ implies $at\_\ell_{0..2}[M]$. Considering $\varphi_1[M-1]$, we see that $at\_\ell_4[M-1] \wedge at\_\ell_{0..2}[M]$ implies $c[M] = 1$. We conclude that the implication

$$at\_\ell_4[1] \ \wedge \ \cdots \ \wedge \ at\_\ell_4[M-1] \ \rightarrow \ at\_\ell_4[M-1] \ \wedge \ c[M] > 0$$

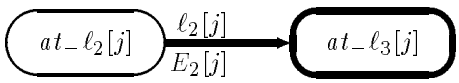
is P-state valid, implying the validity of $E_4[M-1; 1, \ldots, M-2]$.

Proving $A_{2,3}[j]$

Finally, we prove the response property

$$A_{2,3}[j]: \quad at\_\ell_2[j] \ \Rightarrow \ \diamondsuit \, at\_\ell_3[j].$$

This property can be proven by the CHAIN-F diagram:



It remains to discharge the proof obligation

$$E_2[j]: \quad at\_\ell_2[j] \ \Rightarrow \ \diamondsuit \big(at\_\ell_2[j] \ \wedge \ r > 0 \big).$$

Obviously, it suffices to prove the entailment starting from a state which satisfies $\neg(r > 0)$

$$at\_\ell_2[j] \ \wedge \ r = 0 \ \ \Rightarrow \ \ \diamondsuit\big(at\_\ell_2[j] \ \wedge \ r > 0\big).$$

By φ_2 and φ_3, it follows that

$$at\_\ell_2[j] \ \wedge \ r = 0 \ \ \rightarrow \ \ \exists k \in [1..M]\colon \ at\_\ell_2[j] \ \wedge \ at\_\ell_{3..8}[k],$$

stating that if r is not positive, some process $P[k]$ must be executing at some location in the range $[\ell_3..\ell_8]$.

Therefore, it is sufficient to prove the property

$$at\_\ell_2[j] \ \wedge \ at\_\ell_{3..8}[k] \ \Rightarrow \ \diamondsuit\big(at\_\ell_2[j] \ \wedge \ r > 0\big).$$

This proof is presented in the CHAIN diagram of Fig. 3.36.

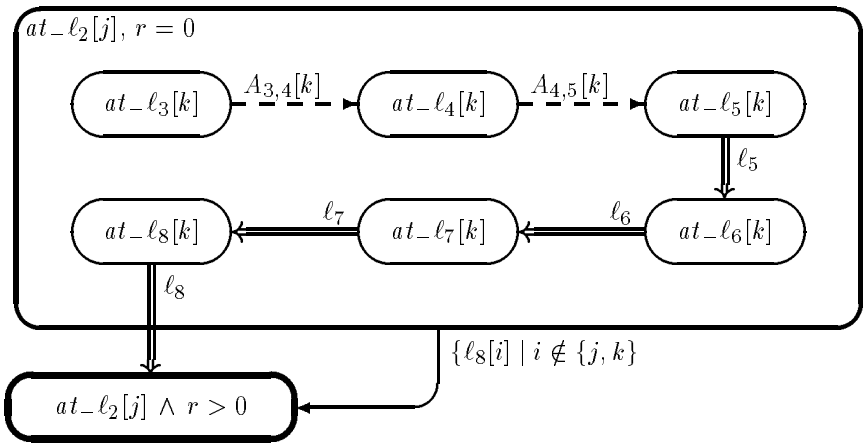


Fig. 3.36. CHAIN diagram for $E_2[j]$.

This concludes the proof of accessibility for program DINE-EXCL.

3.7 Allocation of Quantifiable Resources

One of the problems that requires coordination between concurrent processes is the management of limited resources. There are different types of resources in a system. Some resources, such as the ownership of a particular printer or a shared variable, form one indivisible unit that must be granted in its entirety to a single process at a time. Others, such as disk or memory space, consist of a large number

of identical units, which may be granted to several processes at the same time, provided the sum of granted quantities never exceeds a given available total.

Typically, before entering a particular activity, to which we refer as "a critical section," a process requests a certain quantity of units, and must wait for this quantity to be granted before entering the critical section.

Most of our previous examples considered resources of the atomic type. We now focus our attention on the allocation of resources of the second, *quantifiable*, type. To set a general framework, we consider $M > 0$ processes $P_1, \ldots, P_M$, and a single resource having $N > 0$ units. Assume that an array $X[1..M]$ represents the needs the processes in units of the resource. That is, process P_i needs $X[i]$ units of the resource in order to perform correctly its critical activity. Naturally, we have to assume that

$$X[i] \leq N \quad \text{for every} \quad i = 1, \ldots, M,$$

but this is the only assumption we make. This assumption states that there are sufficiently many units of the resource to satisfy the needs of each process separately.

In the following, we use our notation for parameterized programs, by which L_a denotes the set of indices of processes that currently execute at ℓ_a and $L_{a..b}$ denotes the set of indices of processes currently executing at any of $\ell_a, \ell_{a+1}, \ldots, \ell_b$. Similarly, $N_a = |L_a|$ and $N_{a..b} = |L_{a..b}|$.

A First Approximation

A first approximation to an algorithm that manages the allocation of a quantifiable resource is presented in parameterized program INADEQUATE of Fig. 3.37. In the critical section of $P[i]$, $X[i]$ units of the resource are used. The program uses the generalized semaphore statements

$$\textbf{request}\ (r,\ C),$$

whose effect is equivalent to

$$\langle \textbf{await}\ r \geq C;\ r := r - C \rangle,$$

and

$$\textbf{release}\ (r,\ C),$$

whose effect is equivalent to

$$r\ :=\ r + C.$$

The **request** (r, C) statement is associated with a compassion requirement, prescribing that it is impossible for the statement to be enabled infinitely many times, but executed only finitely many times.

- *Safety*

in $M,\ N$: **integer where** $M > 0,\ N > 0$
X : **array** $[1..M]$ **of integer where** $\forall i \in [1..M]\colon\ X[i] \leq N$
local r : **integer where** $r = N$

$$\prod_{i=1}^{M} P[i] :: \left[\begin{array}{l} \ell_0\colon \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\colon \textbf{noncritical} \\ \ell_2\colon \textbf{request}\ (r,\ X[i]) \\ \ell_3\colon \textbf{critical} \\ \ell_4\colon \textbf{release}\ (r,\ X[i]) \end{array}\right] \end{array}\right]$$

Fig. 3.37. Program INADEQUATE (inadequate allocation of quantifiable resource).

It is easy to see that the program of Fig. 3.37 has the right safety property of exclusion. Exclusion in this case means that the sum of allocated units of the resource never exceeds N. This can be expressed by

$$\left(\sum_{i \in L_3} X[i]\right)\ \leq\ N.$$

This formula takes the sum of all $X[i]$ for processes $P[i]$ that are currently at ℓ_3, and requires that the sum never exceeds N.

This safety property can be deduced from the two invariants

$$\chi_0\colon\quad r \geq 0$$

$$\chi_1\colon\quad \left(\sum_{i \in L_{3,4}} X[i]\right) + r\ =\ N.$$

It is clear that execution of $\ell_2[i]$ increments the sum by $X[i]$ and decrements r by the same amount. Similarly, transition $\ell_4[i]$ leaving the range $\ell_{3,4}$, decrements the sum and increments r by $X[i]$.

- *Individual accessibility not guaranteed*

The program of Fig. 3.37 does not ensure accessibility for each process wishing to enter ℓ_3, that is, it does not satisfy

$$at\_\ell_2[i]\ \Rightarrow\ \diamondsuit\ at\_\ell_3[i] \qquad \text{for every } i,\ 1 \leq i \leq M.$$

To show this, consider the particular case of three processes ($M = 3$), with $N = 2$ and the array of resource requirements given by $X[1] = X[2] = 1, X[3] = 2$.

Consider the computation

$$\left\langle \pi\colon \{\ell_0[1],\ \ell_0[2],\ \ell_0[3]\},\ r\colon 2 \right\rangle \ \cdots \xrightarrow{P[1],P[2],P[3]}$$
$$\cdots \left\langle \pi\colon \{\ell_2[1],\ \ell_2[2],\ \ell_2[3]\},\ r\colon 2 \right\rangle \ \cdots \xrightarrow{P[1],P[2]} \ \cdots$$

$$\left\langle \pi\colon \{\ell_3[1],\ \ell_3[2],\ \ell_2[3]\},\ r\colon 0 \right\rangle \ \cdots \xrightarrow{P[1]}$$
$$\cdots \left\langle \pi\colon \{\ell_2[1],\ \ell_3[2],\ \ell_2[3]\},\ r\colon 1 \right\rangle \ \cdots \xrightarrow{P[1]} \ \cdots$$

$$\left\langle \pi\colon \{\ell_3[1],\ \ell_3[2],\ \ell_2[3]\},\ r\colon 0 \right\rangle \ \cdots \xrightarrow{P[2]} \ \cdots \left\langle \pi\colon \{\ell_3[1],\ \ell_2[2],\ \ell_2[3]\},\ r\colon 1 \right\rangle \ \cdots \xrightarrow{P[2]}$$
$$\cdots \left\langle \pi\colon \{\ell_3[1],\ \ell_3[2],\ \ell_2[3]\},\ r\colon 0 \right\rangle \ \cdots .$$

This computation consists of a finite prefix reaching the state

$$\left\langle \pi\colon \{\ell_3[1],\ \ell_3[2],\ \ell_2[3]\},\ r\colon 0 \right\rangle,$$

followed by an infinite repetition of a segment. The segment starts with both $P[1]$ and $P[2]$ at ℓ_3. Then $P[1]$ performs a complete cycle and ends back at ℓ_3, followed by a similar cycle performed by $P[2]$. Throughout the segment, $P[3]$ waits at ℓ_2. The computation is compassionate with respect to $P[3]$, even though $P[3]$ continuously waits at ℓ_2. This is because transition $\ell_2[3]$ is never enabled in the infinitely repeating segment, having $r \leq 1$ at all states of this segment.

This computation illustrates the possibility of $P[1]$ and $P[2]$ conspiring against $P[3]$, leaving it stranded at ℓ_2.

In **Problem 3.3** the reader is requested to prove the weaker property of communal accessibility $N_2 > 0 \Rightarrow \diamondsuit(N_3 > 0)$ for program INADEQUATE.

An Improved Solution

To remedy the fault of the previous solution, we must be able to assign priorities to the different processes. If the assignment of priorities is fair and the process with highest priority is guaranteed to enter its critical section, then individual accessibility will be ensured. We can look at the possession of the highest priority as another resource, atomic this time, and implement its allocation by another semaphore.

This leads to program Q-RES of Fig. 3.38. This program uses semaphore variable p to grant high priority to precisely one process. The process that gains this priority is allowed to make a request of the quantifiable resource r, and waits until this request is granted. While this process is waiting, no other process can make a request to r. Consequently, the only possible changes to r while this process is waiting at ℓ_3 are caused by processes that exit their critical section and increment r by $X[i]$. Eventually r must become greater or equal to the quantity

$$\begin{array}{ll}\textbf{in} & M,\ N\colon \textbf{integer where } M>0,\ N>0\\ & X\ \ \colon \textbf{array } [1..M] \textbf{ of integer where } \forall i\in[1..M]\colon X[i]\le N\\ \textbf{local} & r,\ p\ \colon \textbf{integer where } r=N,\ p=1\end{array}$$

$$\prod_{i=1}^{M} P[i] :: \left[\begin{array}{l}\ell_0\colon \textbf{loop forever do}\\ \quad\left[\begin{array}{l}\ell_1\colon \textbf{noncritical}\\ \ell_2\colon \textbf{request } p\\ \ell_3\colon \textbf{request } (r,\ X[i])\\ \ell_4\colon \textbf{release } p\\ \ell_5\colon \textbf{critical}\\ \ell_6\colon \textbf{release } (r,\ X[i])\end{array}\right]\end{array}\right]$$

Fig. 3.38. Program Q-RES (allocation of quantifiable resource).

requested by the waiting process. At that stage, the waiting process is granted its request and relinquishes its high priority at ℓ_4 on the way to the critical section.

Let us verify the correctness of program Q-RES.

- *Safety*

  Exclusion requires the invariant

$$\left(\sum_{i\in L_5} X[i]\right)\ \le\ N.$$

This can be established by the invariants

$$\begin{array}{ll}\chi_0\colon & r\ge 0\ \wedge\ p\ge 0\\ \chi_1\colon & N_{3,4}+p\ =\ 1\\ \chi_2\colon & \left(\sum_{i\in L_{4..6}} X[i]\right)+r\ =\ N.\end{array}$$

In particular, exclusion is deducible from χ_2 and $r\ge 0$.

- *Individual accessibility*

  To show accessibility, we prove

$$at\_\ell_2[i]\ \Rightarrow\ \diamondsuit\, at\_\ell_5[i].$$

The CHAIN-F diagram of Fig. 3.39 presents the proof of this property by rule CHAIN-F.

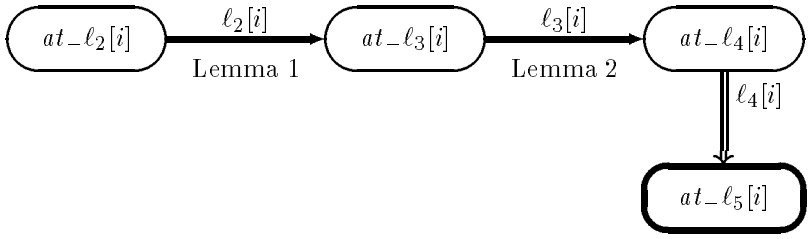


Fig. 3.39. CHAIN-F diagram for accessibility.

The more interesting part of this diagram is the proof of eventual enableness of transitions $\ell_2[i]$ and $\ell_3[i]$. This is proved by Lemmas 1 and 2.

Lemma 1 $\quad at\_\ell_2[i] \;\Rightarrow\; \diamondsuit\big(at\_\ell_2[i] \;\wedge\; p > 0\big)$

The nontrivial case is when $p = 0$ while $P[i]$ is at $\ell_2[i]$. Due to χ_1, this is possible only if $at\_\ell_{3,4}[j]$ for some $j \in [1..M]$.

Therefore, we prove

$$at\_\ell_2[i] \;\wedge\; at\_\ell_{3,4}[j] \;\;\Rightarrow\;\; \diamondsuit\big(at\_\ell_2[i] \;\wedge\; p > 0\big).$$

This is proven by the CHAIN-F diagram of Fig. 3.40.

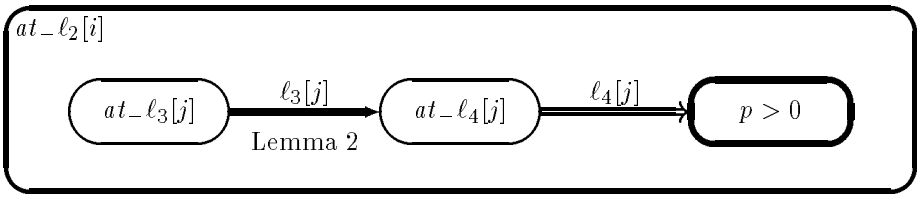


Fig. 3.40. CHAIN-F for Lemma 1.

Note that this proof also relies on Lemma 2 to provide the eventual enableness of $\ell_3[j]$.

Lemma 2 $\quad at\_\ell_3[i] \;\Rightarrow\; \diamondsuit\big(at\_\ell_3[i] \;\wedge\; r \geq X[i]\big)$

The proof of this lemma requires the use of the well-founded rule. The case we have to deal with is $at\_\ell_3[i] \wedge r < X[i]$. Since $X[i] \leq N$, it follows that $r < N$. According to χ_2, this can hold only if $N_{4..6} > 0$. Since $at\_\ell_3[i]$ and χ_1 imply $N_4 = 0$, it follows that $N_{5,6} > 0$.

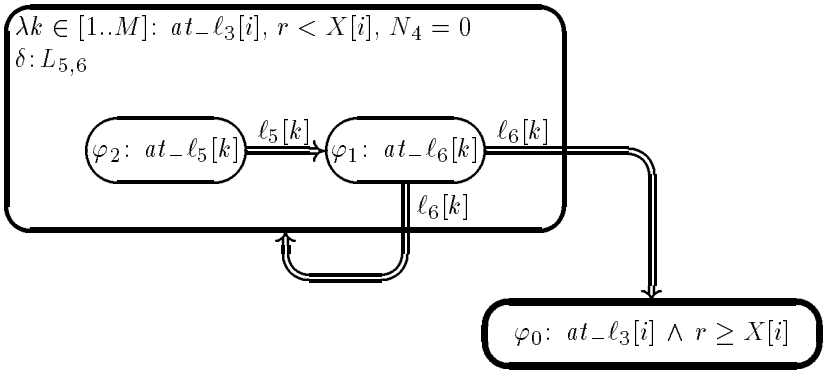


Fig. 3.41. CHAIN diagram for Lemma 2.

The lemma is proved by rule WELL-JP (Fig. 2.2), as shown in the CHAIN diagram of Fig. 3.41.

The double edge connecting φ_2: $at\_\ell_5[k]$ to φ_1: $at\_\ell_6[k]$ represents a transition of process $P[k]$ from ℓ_5 to ℓ_6. The two edges departing from φ_1: $at\_\ell_6[k]$ represent a movement of process $P[k]$ from ℓ_6 back to ℓ_0. Such movement decreases $L_{5,6}$ and increases r. If, as a result of the increase, r becomes greater or equal to $X[i]$, we have reached the goal φ_0. This possibility is represented by the edge connecting φ_1 to φ_0. In the other case, r is smaller than $X[i]$ even after the increase, and by the previous argument $r < X[i]$ and $at\_\ell_3[i]$ imply $L_{5,6} \neq \emptyset$, which means that, for some other $k \in [1..M]$, either $at\_\ell_5[k]$ or $at\_\ell_6[k]$ holds. This is represented by the edge connecting φ_1 to the compound node containing both φ_2 and φ_1.

Note that as long as $r < X[i]$, $P[i]$ cannot move from ℓ_3 to ℓ_4, and therefore N_4 remains zero and $N_{5,6}$ cannot increase.

This concludes the proof of accessibility. We remind the reader that most of the solutions to the *readers-writers* problem (Fig. 2.26) can be viewed as a special case of the quantifiable resource allocation. In these solutions, $M = N$ and $X[i] = 1$ for a reader, and $X[i] = M$ for a writer.

Consequently, the program in Fig. 3.38 also provides a solution to the *readers-writers* problem that ensures individual accessibility.

Response with Past Subformulas

Up to this point, we have only considered response formulas $p \Rightarrow \diamondsuit q$, in which p and q are assertions. The generalization to the case that p or q are past formulas is straightforward. As indicated in Section 1.6, this generalization replaces intermediate assertions by intermediate past-formulas, and implications by entailments.

In Fig. 3.42, we present the general version of rule WELL-F. The other F-rules can be generalized in a similar way.

For past formulas p and $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J} \cup \mathcal{C}$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m$: $\mathcal{S} \mapsto \mathcal{A}$

FW1. $p \;\Rightarrow\; \bigvee_{j=0}^{m} \varphi_j$

FW2. $\rho_\tau \;\wedge\; \varphi_i \;\Rightarrow\; \left[\begin{array}{l} \bigvee_{j=1}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j) \\ \vee \;\; (\varphi'_i \;\wedge\; \delta_i = \delta'_i) \end{array}\right]$ for every $\tau \in \mathcal{T}$

FW3. $\rho_{\tau_i} \;\wedge\; \varphi_i \;\rightarrow\; \bigvee_{j=0}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j)$

} for $i = 1, \ldots, m$

FW4. If $\tau_i \in \mathcal{C}$, then

CW4. $\varphi_i \;\Rightarrow\; \diamondsuit\big(\neg\varphi_i \;\vee\; En(\tau_i)\big)$

Otherwise

JW4. $\varphi_i \;\Rightarrow\; En(\tau_i)$

$p \;\Rightarrow\; \diamondsuit\, q$

Fig. 3.42. General version of rule WELL-F.

** 3.8 Completeness: State Response

In this and the following section we will show that rule WELL-F (Fig. 3.42) is complete for proving P-validity of a given response formula

$$p \;\Rightarrow\; \diamondsuit\, q.$$

We follow a strategy similar to the one taken in Chapters 1 and 4 of the SAFETY

book and divide the proof into two steps. In the first step, covered in this section, we show WELL-F to be complete for state response formulas, i.e., the case that p and q are assertions. In a second step, covered in the next section, we use the statification techniques introduced in Chapter 4 of the SAFETY book to show that completeness for the assertional case implies completeness for the general case in which p and q are past formulas.

The precise statement of completeness for state response formulas is given by the following theorem.

Theorem 3.2 (completeness of rule WELL-F, state response)

For every two assertions p and q such that

$$p \Rightarrow \diamondsuit q$$

is P-valid, there exist
assertions $q = \varphi_0, \varphi_1, \ldots, \varphi_m$,
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J} \cup \mathcal{C}$,
a well-founded domain $\{\mathcal{A}, \succ\}$, and
ranking functions $\delta_0, \ldots, \delta_m$: $S \mapsto \mathcal{A}$,
such that the premises of rule WELL-F (Fig. 3.24) are provable from state validities.

We prove the theorem in this section.

Preliminaries

Let P be a fair transition system with justice set $\mathcal{J}$ and compassion set $\mathcal{C}$. We refer to the union of $\mathcal{J}$ and $\mathcal{C}$ as the *fairness set* of P and denote it by $\mathcal{F} = \mathcal{J} \cup \mathcal{C}$. Assume that $\mathcal{F} = \{\tau_1, \ldots, \tau_m\}$.

For the case that p and q are assertions, we will show that for every p and q there exist intermediate assertions φ_i, one for each fair transition $\tau_i \in \mathcal{F}$, $i = 1, \ldots, m$, a well-founded domain $(\mathcal{A}, \succ)$, and ranking functions $\delta_1, \ldots, \delta_m$ such that premises FW1–FW3 and subpremise JW4 are satisfied as P-state valid implications, and subpremise CW4 is P-valid. From the corollary to the completeness theorem of state invariances (Section 2.5 of the SAFETY book) it follows that premises FW1–FW3 and subpremise JW4 are provable from state validities. Subpremise CW4 will be shown to be provable from state validities in the course of the completeness proof.

An interesting conclusion of the completeness proof is that, while a first-order language over the integers and additional data domains is adequate for expressing all the constructs required by the safety rules, it is no longer adequate for expressing the necessary response constructs. Several extensions to the assertion

language have been suggested. We will use the extension that allows notions such as ordinals and fixpoints to write formulas that quantify over predicates.

In the construction of ranking functions, we intend to construct a single ranking function δ: $\mathcal{S} \mapsto \mathcal{A}$, and take $\delta_1 = \cdots = \delta_m = \delta$.

The Kernel of the Intermediate Assertions

The intermediate assertions $\varphi_1, \ldots, \varphi_m$ we intend to construct have the common form

$$\varphi_i: \quad \varphi \ \wedge \ \psi_i, \quad \text{for } i = 1, \ldots, m.$$

Assertion φ is a common conjunct appearing in all φ_i's, to which we refer as the *kernel* of $\varphi_1, \ldots, \varphi_m$.

Here, we define the common assertion φ and argue that it can be expressed in a first-order assertion language.

We define a (P-)*segment* to be a finite sequence of states $[s_a, \ldots, s_b]$ such that $a \leq b$ and for every i, $a \leq i < b$, s_{i+1} is a τ-successor of s_i for some transition τ of P. A segment $[s_a, \ldots, s_b]$ is called (P-)*accessible* if state s_a is P-accessible. Obviously, if $[s_a, \ldots, s_b]$ is accessible then there exists a P-computation containing $[s_a, \ldots, s_b]$ as a segment. A segment $[s_a, \ldots, s_b]$ is called *q-free* if no s_i, $a \leq i \leq b$, satisfies q.

We say that $[s_a, \ldots, s_b]$ is a *(p, q)-segment* if it is an accessible q-free segment such that $s_a \models p$. Based on this notion, we define an assertion Q such that

$$s \models Q \quad \leftrightarrow \quad \text{there exists a } (p, q)\text{-segment } [s_a, \ldots, s_b] \text{ where } s_b = s.$$

Thus, Q characterizes all states that can appear as last states in a (p, q)-segment. The definition implies that s satisfies Q if and only if there exists a prefix $[s_0, \ldots, s_b]$ of a P-computation, such that $s_b = s$ and there exists a position a, $0 \leq a \leq b$, such that p holds at a and no position i, $a \leq i \leq b$, satisfies q.

Our interest in Q is motivated by the following claim.

Claim 3.3 (characterizes q-pending states)

Assume that $p \Rightarrow \diamondsuit q$ is P-valid and that σ is a P-computation such that Q holds at position $i \geq 0$. Then q holds at some position $k \geq i$ (in fact $k > i$).
Or equivalently:
if $p \Rightarrow \diamondsuit q$ is P-valid then so is the response formula $Q \Rightarrow \diamondsuit q$.

Thus, Q characterizes all states that must have a following q in all computations.

Justification To justify the claim, let $s = s_i$ be the state appearing at position i of σ. Consider the infinite sequence $s_{i+1}, s_{i+2}, \ldots$ obtained by removing the

prefix $s_0, \ldots, s_i$ from σ. By definition of Q, there exists a prefix $\widehat{s}_0, \ldots, \widehat{s}_b$ such that $\widehat{s}_b = s$, $\widehat{s}_a \Vdash p$ for some a, $0 \leq a \leq b$, and $\widehat{s}_j \not\Vdash q$ for all j, $a \leq j \leq b$. Consider the spliced sequence

$$\widetilde{\sigma}: \quad \widehat{s}_0, \ \ldots, \ \widehat{s}_b, \ s_{i+1}, \ s_{i+2}, \ \ldots .$$

Since $\widehat{s}_b = s = s_i$, this sequence is obviously a computation and p holds at position a of $\widetilde{\sigma}$. Since $p \Rightarrow \diamondsuit q$ is P-valid there exists some position $r \geq a$ such that q holds at position r of $\widetilde{\sigma}$. Since q cannot hold at any j, $a \leq j \leq b$, it follows that $r > b$. It follows that q holds at position k of the original computation σ, where $k = r - b + i > i$. ∎

Using the techniques of Section 2.5 of the SAFETY book, we can write a first-order formula $\varphi(V)$ using dynamic arrays which expresses Q. We take $\varphi = Q$ to be the common part of the intermediate assertions.

Example Consider program UP-DOWN of Fig. 3.43 (see also Fig. 1.27). The response property

$$\underbrace{y = 10}_{p} \quad \Rightarrow \quad \diamondsuit \underbrace{y = 5}_{q}$$

is valid over this program.

local $x, \ y$: **integer where** $x = y = 0$

$$P_1 :: \begin{bmatrix} \ell_0\text{: } \textbf{while } x = 0 \textbf{ do} \\ \quad \ell_1\text{: } y := y + 1 \\ \ell_2\text{: } \textbf{while } y > 0 \textbf{ do} \\ \quad \ell_3\text{: } y := y - 1 \\ \ell_4\text{:} \end{bmatrix} \quad \| \quad P_2 :: \begin{bmatrix} m_0\text{: } x := 1 \\ m_1\text{:} \end{bmatrix}$$

Fig. 3.43. Program UP-DOWN.

The kernel assertion Q, relative to p: $y = 10$ and q: $y = 5$, can be given by

$$Q: \quad acc_{\,P} \ \wedge \ y > 5,$$

where

$$acc_{\,P}: \quad (at\_m_0 \wedge at\_\ell_{0,1} \wedge x = 0 \quad \vee \quad at\_m_1 \wedge at\_\ell_{0..4} \wedge \ x = 1) \ \wedge$$
$$y \geq at\_\ell_3 \ \wedge \ (at\_\ell_4 \ \rightarrow \ y = 0)$$

characterizes all the accessible states for this program.

Note that if we observe a state s in which $y = 6$ in a computation σ, it does not necessarily imply that a prefix containing a $(y = 10)$-position is part of that computation. The value 6 may well be the highest value attained by y in σ. We are only guaranteed that there exists some prefix (possibly not present in σ) containing a $(y = 10)$-position and ending in our $(y = 6)$-state without encountering a $(y = 5)$-position in between. However, the preceding claim guarantees that every $(y = 6)$-position is followed by a $(y = 5)$-position. ▟

We denote by $\|Q\|$ the set of all states satisfying Q. A segment $\sigma\colon [s_a, \ldots, s_b]$ is called a *Q-segment* if $s_i \models Q$ for every $i \in [a..b]$.

A Primary Order

The identification of the ranking functions, mapping states into a well-founded domain, is based on the identification of a well-founded ordering over the states of the program.

For a segment $\sigma\colon [s_a, \ldots, s_b]$, we say that σ *leads* from s_a to s_b. Transition $\tau_i \in \mathcal{F}$ is said to be *gratified* in a segment $\sigma\colon [s_a, \ldots, s_b]$ if

- τ_i is taken in σ, or
- $\tau_i \in \mathcal{J} - \mathcal{C}$ and τ_i is disabled on some s_j, $j \in [a..b]$.

We denote by $grat(\sigma)$ the index set of the transitions that are gratified in σ

$$grat(\sigma)\colon \quad \{i \mid \tau_i \in \mathcal{F} \text{ is gratified in } \sigma\}.$$

A segment σ is called *fair* if it gratifies all fair transitions, i.e., $grat(\sigma) = [1..m]$. Note that a compassionate transition $\tau \in \mathcal{C}$ is considered to be gratified in σ if it is taken in σ. In this, the compassionate transitions differ from just transitions which can be gratified also by being disabled somewhere in σ.

Example Consider program UP-DOWN of Fig. 3.43. The set of fair transitions for this program consists of the just transitions ℓ_0, ℓ_1, ℓ_2, ℓ_3, and m_0. The segment

$$\sigma_1\colon \quad \langle \pi\colon \{\ell_0, m_0\},\ x\colon 0,\ y\colon 5\rangle,\ \langle \pi\colon \{\ell_1, m_0\},\ x\colon 0,\ y\colon 5\rangle$$

is not fair. Segment σ_1 gratifies transitions ℓ_0, ℓ_1, ℓ_2, and ℓ_3 but does not gratify m_0. Note that ℓ_0–ℓ_3 are gratified since each has a state within σ_1 on which the transition is disabled.

On the other hand, the following extension of σ_1,

$$\sigma_2\colon \quad \langle \pi\colon \{\ell_0, m_0\},\ x\colon 0,\ y\colon 5\rangle,\ \langle \pi\colon \{\ell_1, m_0\},\ x\colon 0,\ y\colon 5\rangle,\ \langle \pi\colon \{\ell_1, m_1\},\ x\colon 1,\ y\colon 5\rangle$$

is fair since it gratifies all fair transitions. ▟

We define a binary relation $\sqsupset$ between states in $\|Q\|$ as follows:

$$s_1 \sqsupset s_2 \quad \text{iff} \quad \text{there exists a fair } Q\text{-segment leading from } s_1 \text{ to } s_2.$$

Claim 3.4 (well-foundedness of $\sqsupset$)

The relation $\sqsupset$ is a well-founded order over $\|Q\|$.

Justification We start by showing that $\sqsupset$ is a well-founded relation over $\|Q\|$.

Assume to the contrary, that there exists an infinite sequence of states $s^1, s^2, \ldots \in \|Q\|$, such that

$$s^1 \sqsupset s^2 \sqsupset \cdots.$$

By the definition of Q and the meaning of $\sqsupset$, there exists an infinite sequence of states as follows

$$\sigma\colon \quad s_0,\ \ldots,\ \underbrace{s_a,\ \ldots,\ \overbrace{s^1}^{q\text{-free and fair}}}_{q\text{-free}},\ \ldots,\ \underbrace{s^2,\ \ldots,\ s^3}_{q\text{-free and fair}},\ \ldots$$

where $s_a \vDash p$. This sequence satisfies the fairness requirement since in each segment $[s^i, \ldots, s^{i+1}]$ each transition $\tau_i \in \mathcal{C}$ is either taken at least once or is disabled on all states in the segment, and each transition $\tau_i \in \mathcal{J} - \mathcal{C}$ is either taken or disabled on some state in the segment. On the other hand, p holds at position a of σ and q does not hold at any position $j \geq a$, in violation of our assumption that $p \Rightarrow \diamondsuit q$ is P-valid.

Next, we show that $\sqsupset$ is an ordering relation over $\|Q\|$. Relation $\sqsupset$ is anti-symmetric. This is because the existence of s_1, s_2 such that $s_1 \sqsupset s_2$ and $s_2 \sqsupset s_1$ gives rise to the infinitely descending sequence

$$s_1 \sqsupset s_2 \sqsupset s_1 \sqsupset s_1 \sqsupset \cdots,$$

which violates the just-proven well-foundedness of $\sqsupset$. Irreflexivity is proven in the same way by taking $s_1 = s_2$. Transitivity follows from the definition of $\sqsupset$. Namely, if there exists a fair Q-segment $s_a, \ldots, s_b$ and a fair Q-segment $s_b, \ldots, s_c$, then the Q-segment $s_a, \ldots, s_b, \ldots, s_c$ obtained by dove-tailing these two segments is fair and establishes $s_a \sqsupset s_c$. ∎

Example Consider program SKIP2 presented in Fig. 3.44.

The set of fair (just) transitions for this program is:

$$\mathcal{F}\colon \quad \{\tau_1\colon \ell_1, \quad \tau_2\colon \ell_2^{\mathrm{E}}, \quad \tau_3\colon \ell_3, \quad \tau_4\colon \ell_4, \quad \tau_5\colon \ell_2^{\mathrm{X}}\},$$

where ℓ_2^{E} and ℓ_2^{X} are the entry and exit transitions from the cooperation statement ℓ_2.

The response property we investigate for program SKIP2 is

$$\left[\begin{array}{l} \ell_1\colon \textbf{skip} \\ \ell_2\colon \left[\ [\ell_3\colon \textbf{skip};\ \ell_5\colon\ ]\ \ \|\ \ [\ell_4\colon \textbf{skip};\ \ell_6\colon\ ]\ \right] \\ \ell_7\colon \end{array}\right]$$

Fig. 3.44. Program SKIP2.

$$\underbrace{at\_\ell_3\ \wedge\ at\_\ell_4}_{p}\ \Rightarrow\ \diamondsuit\ \underbrace{at\_\ell_7}_{q}\ .$$

The corresponding assertion Q should characterize all the states which can be reached from accessible states satisfying p: $at\_\ell_3 \wedge at\_\ell_4$ without passing through a state satisfying q: at_ℓ_7. Therefore, Q is given by

$$Q\colon\quad at\_\ell_{3,5}\ \wedge\ at\_\ell_{4,6}.$$

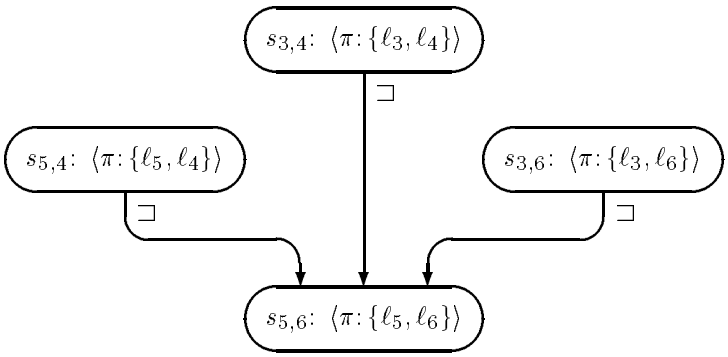


Fig. 3.45. Q-states of program SKIP2 related by $\sqsupset$.

In Fig. 3.45, we present the four Q-states of this program and their ordering by relation $\sqsupset$. For example, segment $[s_{5,4}, s_{5,6}]$ is fair, i.e., gratifies all fair transitions τ_1–τ_5: it gratifies τ_1, τ_2, τ_3, and τ_5 (i.e., ℓ_1, ℓ_2^E, ℓ_3, and ℓ_2^X) because they are all disabled on $s_{5,4}$; it gratifies $\tau_4\colon \ell_4$ by taking this transition in the step from $s_{5,4}$ to $s_{5,6}$.

On the other hand, the segment $[s_{3,4}, s_{5,4}]$ is not fair since it does not gratify ℓ_4, which is enabled on all states in the segment but not taken there. This is why there is no $\sqsupset$-labeled edge connecting $s_{3,4}$ to $s_{5,4}$ in Fig. 3.45, nor is there a $\sqsupset$-labeled edge from $s_{3,4}$ to $s_{3,6}$. ∎

The Primary Order is Too Coarse

Order $\sqsupset$ is a good first approximation to the order we need. It is well-founded and measures progress in the sense that when the system moves from state s_1 to state s_2 such that $s_1 \sqsupset s_2$, a progress step has been taken. This is because the system cannot take infinitely many such steps.

Unfortunately, the order $\sqsupset$ only provides a coarse measure of progress. Reconsider program SKIP2 presented in Fig. 3.44 and the $\sqsupset$-ordering of its Q-states in Fig. 3.45.

Clearly, every step leading from one Q-state to another represents progress towards the goal q: at_ℓ_7. However, the order $\sqsupset$ identifies only segments leading to $s_{5,6}$ as observable progress. In particular, it does not identify the moves from $s_{3,4}$ to either $s_{5,4}$ or $s_{3,6}$ as observable progress. It is possible to conclude that $\sqsupset$ successfully measures coarse progress but is insensitive to finer progress.

As a first step in the refinement of $\sqsupset$, we find a ranking function into a well-founded domain, whose ordering is compatible with $\sqsupset$.

Mapping into the Ordinals

As a first approximation of the well-founded domain $(\mathcal{A}, \succ)$, we take $(\mathcal{O}, >)$, where $\mathcal{O}$ is the domain of the *ordinals* and $>$ is the natural order over the ordinals.

We remind the reader that the ordinals are obtained by extending the sequence of natural numbers beyond the finite naturals. There are very few facts about the ordinals that are required in our proof here. Following is a list of some of these properties:

- Every natural number is an ordinal. We refer to the naturals as the *finite ordinals*.
- There is a total order $<$ on the ordinals which generalizes the "smaller than" ordering over the naturals.
- Some ordinals β are the immediate successors of another ordinal α, and we write $\beta = \alpha + 1$. Others are *limit ordinals* and are defined as the smallest ordinal greater than (least upper bound of) all ordinals in a given set. For example, ω is the limit ordinal obtained as the least upper bound of all finite ordinals, and is usually described as the first infinite ordinal. The ordinal $\omega + 1$ is a non-limit ordinal and is the immediate successor of ω.
- The order $>$ ("greater than") is well-founded over the ordinals.
- There exists an induction principle over the ordinals, called *transfinite induction*, which generalizes the complete induction principle over the naturals. It can be stated as

$$\forall\alpha\Big((\forall\beta<\alpha\colon\varphi(\beta))\rightarrow\varphi(\alpha)\Big)\quad\rightarrow\quad\forall\gamma\colon\varphi(\gamma).$$

This principle claims that if we can prove the induction step then the property φ holds for every ordinal. The induction step requires showing, for every α, that if $\varphi(\beta)$ for all $\beta<\alpha$ then φ holds for ordinal α.

Ordinals are not so mysterious as they may sometimes appear to be. In fact, we have already introduced and extensively used structures that are isomorphic to the low infinite ordinals.

A lexicographic pair of natural numbers (m,n) can be faithfully considered as an isomorphic representation of the ordinal $\omega\cdot m+n$. To see this, we observe that lexicographic pairs of the form $(0,n)$ are isomorphic to the naturals, with $(0,n)$ corresponding to n. The pair $(1,0)$, like ω, is the smallest pair which is bigger than all the pairs in the infinite set $\{(0,n)\mid n\geq 0\}$.

Our proof relies on the following general theorem.

Theorem 3.5 (embedding within the ordinals)

Let $\sqsupset$ be a well-founded ordering over a set S. Then, there exists a (ranking) function $\delta_\pi\colon S\mapsto\mathcal{O}$ such that:

(a) $s\sqsupset s'$ implies $\delta_\pi(s)>\delta_\pi(s')$.

(b) If $\delta_\pi(s)=\beta$ and $\alpha<\beta$, then there exists an element s' such that $\delta_\pi(s')=\alpha$ and $s\sqsupset s'$.

(c) If $s'\sqsupset s''$ implies $s\sqsupset s''$ for every $s''\in S$, then $\delta_\pi(s)\geq\delta_\pi(s')$.

The theorem can be proven, using a transfinite construction that assigns a value $\delta_\pi(s)$ to an element s, based on the values $\delta_\pi(s')$ assigned to all $s'\sqsubset s$. The constrruction is defined as follows:

- $\delta_\pi(s)=0$ for all $\sqsupset$-minimal elements of S, i.e., elements s such that there exists no $s'\sqsubset s$.
- For all other $s\in S$, $\delta_\pi(s)$ is the least upper bound of $\delta_\pi(s')+1$ for all $s'\sqsubset s$.

Example Consider system LEX presented in Fig. 3.46. Transition τ_1 of this system sets x to zero and sets y to an arbitrary positive integer value. Transition τ_1 can be taken only once since it sets x to 0 which disables τ_1 for any future activation. Transition τ_2 can be taken as long as $y>0$ and always sets y to a nonnegative value which is smaller than its value before the transition. Obviously, any computation of this systems takes τ_1 once and takes τ_2 finitely many times until it terminates in the state $\langle x\colon 0,\ y\colon 0\rangle$.

Assume we wish to prove for this system the following response property.

$$\underbrace{x=y=1}_{p=\Theta}\quad\Rightarrow\quad\diamondsuit\Big(\underbrace{x=y=0}_{q}\Big)\ .$$

V: $\{x, y : \textbf{integer}\}$

Θ: $x = y = 1$

$\mathcal{T}$: $\{\tau_I, \tau_1, \tau_2\}$ where $\rho_{\tau_1}: x > 0 \ \wedge \ x' = 0 \ \wedge \ y' > 0$
$\rho_{\tau_2}: x = 0 \ \wedge \ y > 0 \ \wedge \ x' = x \ \wedge \ 0 \leq y' < y$

$\mathcal{J}$: $\{\tau_1, \tau_2\}$

$\mathcal{C}$: $\emptyset$

Fig. 3.46. System LEX.

States of this system consist of pairs of integers (m, n), where m and n are the current values of x and y, respectively. Clearly, the set of Q-states is given by

$$\|Q\|: \quad \big\{(1,1)\big\} \ \cup \ \big\{(0,n) \mid n > 0\big\}.$$

Consider the primary ordering $\sqsupset$, holding between (m_1, n_1) and $(m_2, n_2) \in \|Q\|$ if there exists a fair Q-segment connecting (m_1, n_1) to (m_2, n_2). It can be shown that the order $\sqsupset$ is given by

$$(m_1, n_1) \ \sqsupset (m_2, n_2): \quad (m_1, n_1) \succ (m_2, n_2) \ \wedge \ m_2 = 0 \ \wedge \ n_2 > 0,$$

where $\succ$ is the lexicographic ordering over $\|Q\|$. The condition $m_2 = 0$ follows from the observation that any fair segment must either take τ_1 which sets x to 0, implying $m_2 = 0$, or have τ_1 disabled on some state of the segment, which also implies $m_2 = 0$. The consequence of this condition is that any two states of the form $(1, n_1)$ and $(1, n_2)$ are unrelated by $\sqsupset$. The condition $n_2 > 0$ excludes from consideration the state $(0, 0)$ which satisfies q.

Applying the described bottom-up construction of the mapping δ_π, we observe the following:

- $(0, 1)$ is the least element of $\|Q\|$, hence $\delta_\pi\big((0,1)\big) = 0$.
- $\delta_\pi\big((0,2)\big)$ is the least upper bound of $\delta_\pi\big((m,n)\big) + 1$ for all $(m, n) \prec (0, 2)$. Since the only element smaller than $(0, 2)$ is $(0, 1)$, we compute

  $$\delta_\pi\big((0,2)\big) \ = \ \textit{least-upper-bound}\big(\{1\}\big) \ = \ 1.$$

  In a similar way, we can show that

  $$\delta_\pi\big((0, n+1)\big) \ = \ n$$

  for every natural n.
- Next consider $\delta_\pi\big((1,1)\big)$. State $(1, 1)$ dominates all states $(0, n)$. Consequently, denoting *least-upper-bound* by *l.u.b*, we obtain

$$\delta_\pi\big((1,1)\big) \;=\; l.u.b\Big(\big\{\delta_\pi((0,n+1))+1 \mid n \in \mathbb{N}\big\}\Big) \;=\; lub\{n+1 \mid n \in \mathbb{N}\} \;=\; \omega.$$

Let us return to the general case of the primary order $\sqsupset$ holding between states s and s' if there exists a fair Q-segment connecting s to s'. Applying the embedding theorem to $\sqsupset$, we obtain a mapping δ_π from the program states to the ordinals $\mathcal{O}$.

The following statement identifies a property satisfied by the mapping δ_π.

Property P0 If $s \in \|Q\|$ and s' is a P-successor of state s, not satisfying q, then $s' \in \|Q\|$ and $\delta_\pi(s) \geq \delta_\pi(s')$.

Justification From the definition of Q it follows that if s', a P-successor of some $s \in \|Q\|$, does not satisfy q it is reachable by a (p,q)-prefix and, therefore, belongs to $\|Q\|$. Let s'' be such that $s' \sqsupset s''$. By definition of $\sqsupset$, there exists a fair Q-segment σ: $[s', \ldots, s'']$. Obviously, the segment $[s, s', \ldots, s'']$ is also a fair Q-segment, implying $s \sqsupset s''$. Thus, $s' \sqsupset s''$ implies $s \sqsupset s''$ for every s''. By clause (c) of the embedding theorem, it follows that $\delta_\pi(s) \geq \delta_\pi(s')$. ∎

The Secondary Ranking

We can view the mapping δ_π as a first approximation to our desired ranking function. Unfortunately, it did not remedy our complaint about $\sqsupset$ being insensitive to fine progress. For example, the mapping δ_π, corresponding to the fundamental order presented in Fig. 3.45, is the following.

$$\delta_\pi(s_{3,4}) = \delta_\pi(s_{5,4}) = \delta_\pi(s_{3,6}) = 1, \quad \delta_\pi(s_{5,6}) = 0.$$

This primary rank is presented in Fig. 3.47.

As we see, there is no observable progress (according to the measure δ_π) when the program moves from $s_{3,4}$ to $s_{5,4}$ or from $s_{3,4}$ to $s_{3,6}$.

We proceed to define a secondary ranking which, together with δ_π, will form our final ranking function. Looking at the diagram of Fig. 3.47, we see that the three states $s_{3,4}$, $s_{5,4}$, and $s_{3,6}$ are situated on a plateau defined by $\delta_\pi = 1$. However, in the order of execution, $s_{5,4}$ and $s_{3,6}$ are "closer to the edge" of the plateau than state $s_{3,4}$. How can we measure this closeness? A possible measure could be the "size" of possible paths that are all located within the tableau. Thus, $s_{3,4}$ is further away from the edge of the plateau since it has a path of length 2, traversing states that all have $\delta_\pi = 1$. States $s_{5,4}$ and $s_{3,6}$ are closer to the edge since they only have paths of length 1. ∎

In the simple example shown here, we could take the "size" of the path to be simply its length. However, in many cases this is not the right measure, since

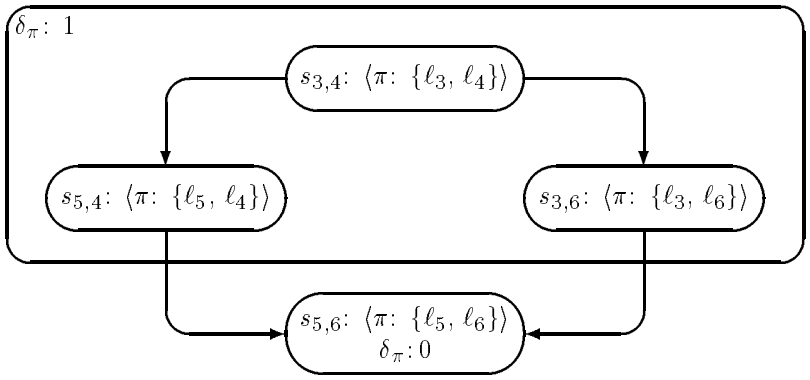


Fig. 3.47. Q-states of program SKIP2 ranked by δ_π.

states may have infinite paths all with the same value of δ_π. The relevant "size" of a path in our case is the number of transitions it gratifies.

For a segment σ, we define the *deficit* of σ, denoted by $\Delta(\sigma)$, to be the smallest natural number $i \geq 1$ such that $i \notin \mathit{grat}(\sigma)$, i.e., τ_i is not gratified by σ. If σ is fair we define $\Delta(\sigma) = m + 1$.

We use the deficit as the measure of the "size" of a segment. Note that it is related to the number of transitions gratified by a segment, since if $\mathit{grat}(\sigma) \subseteq \mathit{grat}(\sigma')$ then $\Delta(\sigma) \leq \Delta(\sigma')$.

A (Q-)segment σ: $[s_a, \ldots, s_b]$ is called *leveled* if $\delta_\pi(s_a) = \delta_\pi(s_{a+1}) = \cdots = \delta_\pi(s_b)$. This corresponds to the previously discussed notion of a segment that is fully contained in a plateau, defined by the value of $\delta_\pi(s_a)$ which is common to all states in the segment. We define the *height* of a state s, denoted by $h(s)$, by

$$h(s) \;=\; \max\Big\{\Delta\big([s = s_a, \ldots, s_b]\big) \mid \delta(s_a) = \cdots = \delta(s_b)\Big\}.$$

That is, $h(s)$ is the maximal deficit over all leveled segments originating at s. This corresponds to the intuition of the distance from the edge of the plateau along leveled segments.

Example Consider the ranking by δ_π on the states presented in Fig. 3.47. Segments $[s_{3,4}, s_{5,4}]$ and $[s_{3,4}, s_{3,6}]$ are leveled. Segments $[s_{5,4}, s_{5,6}]$ and $[s_{3,4}, s_{5,4}, s_{5,6}]$ are not leveled.

There are three leveled Q-segments originating at $s_{3,4}$: $[s_{3,4}]$, $[s_{3,4}, s_{5,4}]$, and $[s_{3,4}, s_{3,6}]$. Segment $[s_{3,4}]$ gratifies all pair transitions except for τ_3: ℓ_3 and τ_4: ℓ_4. Consequently, its deficit is $\Delta\big([s_{3,4}]\big) = 3$. Segment $[s_{3,4}, s_{5,4}]$ gratifies all fair transitions except for τ_4: ℓ_4. Consequently, its deficit is $\Delta\big([s_{3,4}, s_{5,4}]\big) = 4$. Segment $[s_{3,4}, s_{3,6}]$ gratifies all fair transitions except for τ_3: ℓ_3. Consequently

$\Delta\big([s_{3,4}, s_{3,6}]\big) = 3$. Taking the maximum over $\{3, 4\}$, we obtain the height of $s_{3,4}$ as $h(s_{3,4}) = 4$.

The only leveled Q-segment originating at $s_{5,4}$ is the singleton segment $[s_{5,4}]$ which gratifies all fair transitions except for $\tau_4{:}\,\ell_4$ and $\tau_5{:}\,\ell_2^X$. Consequently, $h(s_{5,4}) = 4$.

In a similar way, $h(s_{3,6}) = 3$ and $h(s_{5,6}) = 5$. ∎

The Final Ranking

The final ranking δ that is intended to satisfy premises FW2 and FW3 combines the primary ranking δ_π with the secondary ranking induced by comparing state heights.

As the well-founded domain, we take the lexicographic product

$$(\mathcal{A},\ \succ){:}\quad (\mathcal{O},\ >) \times \big([1..m+1],\ >\big).$$

Thus, elements of $\mathcal{A}$ are pairs (α, k), where $\alpha \in \mathcal{O}$ is an ordinal and $k \in [1..m+1]$. The ranking function, mapping states into elements of $\mathcal{A}$, is given by

$$\delta(s){:}\quad \big(\delta_\pi(s),\ h(s)\big).$$

This definition of $\delta(s)$ applies only to states which satisfies Q. To complete the definition, we define

$$\delta(s)\ =\ (0, 0),$$

for all states that do not satisfy Q. An immediate consequence of this definition is that

$$\delta(s_1)\ \succ\ \delta(s_2)$$

for every $s_1 \in \|Q\|$ and $s_2 \notin \|Q\|$. This is because $\delta_0(s) \geq 0$ and $h(s) > 0$ for every $s \in \|Q\|$.

Rule WELL-F allows us to have a different ranking function δ_i for each $i = 0, 1, \ldots, m$. However, in the proof of completeness we do not use this option, and take $\delta_i = \delta$ for all $i = 0, \ldots, m$.

Example We return to program SKIP2. In Fig. 3.48 we present the ordering between the states as determined by the complete rank δ.

As we see in the diagram, states $s_{3,4}$ and $s_{5,4}$ have equal ranks $(1, 4)$. This is because they both have $\delta_\pi = 1$ and have height 4. State $s_{3,4}$ is higher than $s_{3,6}$ because $\delta_\pi(s_{3,4}) = \delta_\pi(s_{3,6}) = 1$ but $h(s_{3,4}) = 4 > 3 = h(s_{3,6})$. States $s_{5,4}$ and $s_{3,6}$ are higher than $s_{5,6}$ because $\delta(s_{5,4}) = (1, 4) > (0, 5) = \delta_\pi(s_{5,6})$ and $\delta(s_{3,6}) = (1, 3) \succ (0, 5) = \delta(s_{5,6})$.

Thus, the full ranking δ identifies the step from $s_{3,4}$ to $s_{3,6}$ as observable

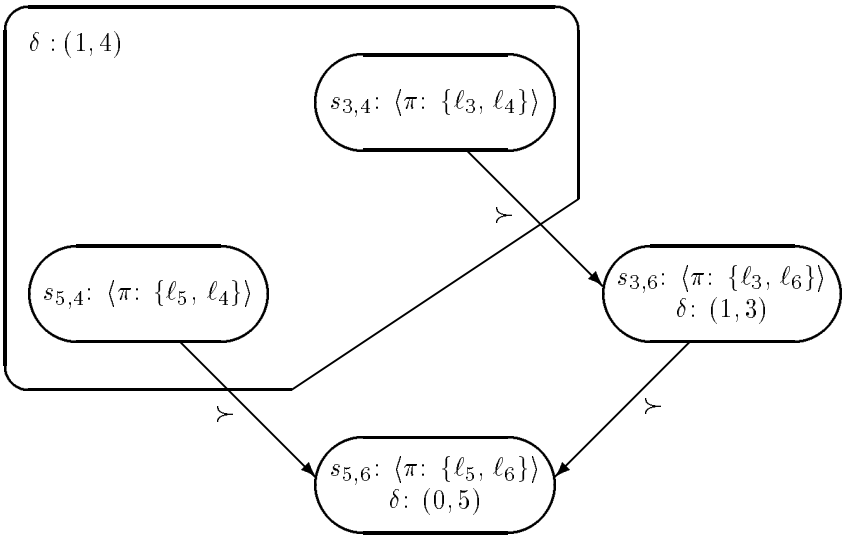


Fig. 3.48. States of program SKIP2 ranked by δ.

progress but not the step from $s_{3,4}$ to $s_{5,4}$. The reason for this lack of symmetry will become apparent later. ▟

There are several properties that the defined constructs satisfy.

Property P1 If $s \in \|Q\|$ then $h(s) \leq m$.

This is because $h(s) = m + 1$ implies the existence of a leveled Q-segment $s = s_a, \ldots, s_b$ which gratifies all the fair transitions. However, in this case, $s_a, \ldots, s_b$ is fair and therefore $s_a \sqsupset s_b$, contradicting the assumption that $s_a, \ldots, s_b$ is leveled.

Property P2 If $s, s' \in \|Q\|$ and s' is a successor of s then $\delta(s) \succcurlyeq \delta(s')$.

Property P0 already established $\delta_\pi(s) \geq \delta_\pi(s')$. If $\delta_\pi(s) > \delta_\pi(s')$ we are done.

It remains to consider the case $\delta_\pi(s) = \delta_\pi(s')$ and show that $h(s) \geq h(s')$. Let $h(s') = j$. By definition of the height h, there exists a leveled Q-segment σ: $[s'..s_b]$ such that $\Delta(\sigma) = j$. The segment $\widetilde{\sigma}$: $s, s', \ldots, s_b$ is a leveled $\big(\delta_\pi(s) = \delta_\pi(s')\big)$ Q-segment originating at s. Since $\widetilde{\sigma}$ extends σ, $grat(\widetilde{\sigma}) \supseteq grat(\sigma)$ and therefore $\Delta(\widetilde{\sigma}) \geq \Delta(\sigma) = j$. Since $h(s)$ is the maximal deficit over all leveled Q-segments originating at s, and $\widetilde{\sigma}$ is one of them, it follows that $h(s) \geq \Delta(\widetilde{\sigma}) \geq j = h(s')$, leading to the required inequality.

Property P3 If $s, s' \in \|Q\|$, $h(s) = i$, and s' is a τ_i-successor of s, then $\delta(s) \succ \delta(s')$.

By P2, $\delta(s) \succcurlyeq \delta(s')$. If $\delta_\pi(s) > \delta_\pi(s')$, we are done. Therefore, consider the case $\delta_\pi(s) = \delta_\pi(s')$ and let $h(s') = j$. Since $\delta(s) \succcurlyeq \delta(s')$, it follows that $i \geq j$. It remains to show that $i \neq j$. Assume, to the contrary, that $i = j$. As before, let σ: $[s', \ldots, s_b]$ be a leveled Q-segment realizing the deficit $\Delta(\sigma) = j$. This means that σ gratifies all fair transitions τ_k, $k < j$, but does not gratify τ_j. Consider again the extended segment $\tilde{\sigma}$: $[s, s', \ldots, s_b]$. This segment gratifies all the transitions gratified by σ and, in addition, also gratifies $\tau_i = \tau_j$ since τ_i is taken in the step from s to s'. It follows that $\Delta(\tilde{\sigma}) > i$. This contradicts the definition of $h(s)$ as being the maximal deficit of all leveled Q-segments originating at s.

The Intermediate Assertions

In the next step of the proof, we construct the intermediate assertions $\varphi_1, \ldots, \varphi_m$, required by rule WELL-F. For each $i = 1, \ldots, m$, we define

$$\varphi_i: \quad Q \ \wedge \ h = i.$$

Thus, φ_i is satisfied by all Q-states whose height is precisely i.

To complete the definition, we take

$$\varphi_0: \ q.$$

The following property is a consequence of the definition of the intermediate assertions.

Property P4 If $s \in \|Q\|$, then s satisfies one of $\varphi_1, \ldots, \varphi_m$.

To prove this statement, let s be a state satisfying Q. It has a defined height $h(s) = j$ which, by P1, is in the range $j \in [1..m]$. Consequently, s satisfies φ_j: $Q \wedge h = j$.

Example For the response property $at\_\ell_3 \wedge at\_\ell_4 \Rightarrow \diamondsuit\, at\_\ell_7$ of program SKIP2, we obtain

$$\begin{array}{lcll}
\varphi_0 = q: & at_\ell_7 & & \\
\varphi_1 : & \underbrace{at_\ell_{3,5} \ \wedge \ at_\ell_{4,6}}_{Q} \ \wedge \ h = 1 & \sim & \text{F} \\
\varphi_2 : & at_\ell_{3,5} \ \wedge \ at_\ell_{4,6} \ \wedge \ h = 2 & \sim & \text{F} \\
\varphi_3 : & at_\ell_{3,5} \ \wedge \ at_\ell_{4,6} \ \wedge \ h = 3 & \sim & at_\ell_3 \ \wedge \ at_\ell_6 \\
\varphi_4 : & at_\ell_{3,5} \ \wedge \ at_\ell_{4,6} \ \wedge \ h = 4 & \sim & at_\ell_{3,5} \ \wedge \ at_\ell_4
\end{array}$$

$$\varphi_5 \quad : \quad at\_\ell_{3,5} \ \wedge \ at\_\ell_{4,6} \ \wedge \ h = 5 \quad \sim \quad at\_\ell_5 \ \wedge \ at\_\ell_6.$$

The assertions are simplified based on the full knowledge of the four Q-states and their heights. ∎

Claim 3.3 implies that the response formula $Q \ \Rightarrow \ \diamondsuit\, q$ is P-valid. Since φ_i: $Q \wedge h = i$ implies Q we conclude that for every $i = 1, \ldots, m$, the response formula

$$\varphi_i \ \Rightarrow \ \diamondsuit\, q$$

is also P-valid. ∎

Proving the Premises

The construction of $\varphi_0, \ldots, \varphi_m$ concludes the development of all the constructs needed for rule WELL-F.

We proceed to show that all premises are satisfied by these constructs. All premises, excluding CW4, are implications $\varphi \ \rightarrow \ \psi$ that have to be shown as P-state valid. To prove this, it is sufficient to show that any P-accessible state satisfying φ also satisfies ψ.

- Premise FW1. This premise requires

$$p \ \rightarrow \ \bigvee_{j=0}^{m} \varphi_j.$$

Let s be an accessible state satisfying p. If s satisfies $q = \varphi_0$, we are done. Otherwise, s satisfies assertion Q since it is reachable by the singleton (p, q)-segment $[s]$. By P4, s satisfies φ_j: $Q \wedge h = j$ for some $j \in [1..m]$.

- Premise FW2. This premise requires

$$\rho_\tau \ \wedge \ \varphi_i \quad \rightarrow \quad \bigvee_{j=0}^{m} (\varphi'_j \ \wedge \ \delta \succ \delta') \ \vee \ (\varphi'_i \ \wedge \ \delta = \delta').$$

Let s be a state satisfying φ_i: $Q \wedge h = i$ and let s' be a τ-successor of s. If s' satisfies $q = \varphi_0$, we are done. Otherwise, s' satisfies Q, then, by P4, φ'_j holds for some $j \in [1..m]$ and, by P2, $\delta(s) \succcurlyeq \delta(s')$ which means that either $\delta_\pi(s) > \delta_\pi(s')$ or $\delta_\pi(s) = \delta_\pi(s')$ and $h(s) \geq h(s')$. If $\delta_\pi(s) > \delta_\pi(s')$ or $\delta_\pi(s) = \delta_\pi(s')$ and $h(s) > h(s')$, then $\delta(s) \succ \delta(s')$. Otherwise, $\delta_\pi(s) = \delta_\pi(s')$ and $h(s) = h(s')$ which implies $\delta(s) = \delta(s')$ and that $\varphi'_i = \varphi_i(s') = Q(s') \wedge h(s') = i$ holds.

- Premise FW3. This premise requires

$$\rho_{\tau_i} \ \wedge \ \varphi_i \quad \rightarrow \quad \bigvee_{j=0}^{m} (\varphi'_j \ \wedge \ \delta \succ \delta').$$

Let s be a state satisfying φ_i: $Q \wedge h = i$ and s' a τ_i-successor of s. If s' satisfies $q = \varphi_0$, we are done. Otherwise s' satisfies Q and, therefore, by P4, φ'_j for some $j \in [1..m]$. Also, by P3, $\delta(s) \succ \delta(s')$.

Premise FW4 consists of the two subpremises: CW4 and JW4. We consider JW4 first.

- Subpremise JW4. This subpremise requires

$$\varphi_i \;\rightarrow\; En(\tau_i).$$

Let s be a state satisfying φ_i: $Q \wedge h = i$. By the definition of h, there exists a leveled Q-segment σ: $[s, \ldots, s_b]$ whose deficit is $\Delta(\sigma) = i$. This implies that τ_i is not gratified in σ.

Transition τ_i must be enabled on s because otherwise, τ_i would have been gratified in σ.

- Subpremise CW4. This subpremise requires a proof of

$$\varphi_i \;\Rightarrow\; \diamondsuit\big(\neg\varphi_i \;\vee\; En(\tau_i)\big).$$

All other premises considered previously were implications of state formulas. Since our statement of completeness is relative to state validities it was enough to show that premises FW2, FW3, and subpremise JW4 are state valid and that premise FW1 is P-state valid.

Subpremise CW4 is a temporal formula. Consequently, we have to show that its proof can be syntactically reduced, perhaps using rule WELL-F again, to state and P-state validities.

We begin by showing that $\varphi_i \Rightarrow \diamondsuit\big(\neg\varphi_i \vee En(\tau_i)\big)$ is P^{-i}-valid where P^{-i} is the fair transition system obtained from P by removing transition τ_i from the compassion set $\mathcal{C}_P$. This is motivated by the observation that, if we are currently at position j where τ_i is disabled and are working towards the first position $k > j$ where τ_i becomes enabled, then τ_i cannot be taken between j and k. Therefore, the computation segment between j and k is also a computation segment of the simpler system P^{-i}.

Assume to the contrary, that $\varphi_i \Rightarrow \diamondsuit\big(\neg\varphi_i \vee En(\tau_i)\big)$ is not P^{-i}-valid. Then, there exists a computation σ of P^{-i} and a position $j \geq 0$, such that φ_i holds and τ_i is disabled at all positions $k \geq j$. Since τ_i is continually disabled beyond j, σ is also a computation of the complete system P. In particular, σ is fair with respect to τ_i which is never taken due to the continual disableness of τ_i. Furthermore, φ_i implies Q which implies $\neg q$. It follows that q never holds beyond j leading to the fact that $Q \Rightarrow \diamondsuit\, q$ does not hold on σ, a computation of P. This contradicts a corollary of Claim 3.3. We therefore conclude that $\varphi_i \Rightarrow \diamondsuit\big(\neg\varphi_i \vee En(\tau_i)\big)$ is P^{-i}-valid.

Next, we show that subpremise CW4 is not only P^{-i}-valid but can be proven from

state validities. This is the time to explain to the reader that the completeness proof is done by induction on κ, the number of compassionate transitions.

If $\kappa = 0$ then we are never called to establish subpremise CW4. Therefore all necessary premises and subpremises are state formulas and it is sufficient to show that they are state or P-state valid as we have done in the previous discussion.

If $\kappa > 0$ then system P^{-i} has only $\kappa - 1 < \kappa$ compassionate transitions since τ_i is compassionate. By the induction hypothesis, rule WELL-F is complete for such systems. It follows that the response formula $\varphi_i \Rightarrow \diamondsuit\big(\neg\varphi_i \vee En(\tau_i)\big)$, which has been shown to be P^{-i}-valid, is provable by a (recursive use of) rule WELL-F.

This concludes the proof of completeness of rule WELL-F for proving response formulas $p \Rightarrow \diamondsuit\, q$ for the case that p and q are assertions.

Example We illustrate the generation of the constructs needed for rule WELL-F on an example. In particular, we wish to illustrate the principle of inductive reduction by which compassionate transitions are removed one at a time.

Consider program SIMPLE-MUX presented in Fig. 3.49.

$$\textbf{local } y\colon \textbf{integer where } y = 1$$

$$P_1 :: \left[\begin{array}{l} \ell_0\colon \textbf{while } y = 1 \textbf{ do} \\ \qquad \left[\begin{array}{l} \ell_1\colon\ y := y - 1 \\ \ell_2\colon\ \textbf{critical} \\ \ell_3\colon\ y := y + 1 \end{array}\right] \\ \ell_4\colon \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0\colon \textbf{request } y \\ m_1\colon \end{array}\right]$$

Fig. 3.49. Program SIMPLE-MUX.

This program can be viewed as a simplified version of program MUX-SEM presented in Fig. 3.1.

The property in which we are interested is

$$\psi\colon\quad \underbrace{at\_m_0}_{p} \quad \Rightarrow \quad \diamondsuit \underbrace{at\_m_1}_{q}\ ,$$

claiming that process P_2 eventually terminates. This formula is obviously valid over SIMPLE-MUX. We follow the process outlined in the completeness proof to show how formula ψ can be verified over SIMPLE-MUX, using rule WELL-F.

Constructing Assertion Q

In this simple case, we can explicitly enumerate the set of accessible states. In Fig. 3.50 we present a state-transition graph which displays the accessible states of program SIMPLE-MUX and the transitions between them.

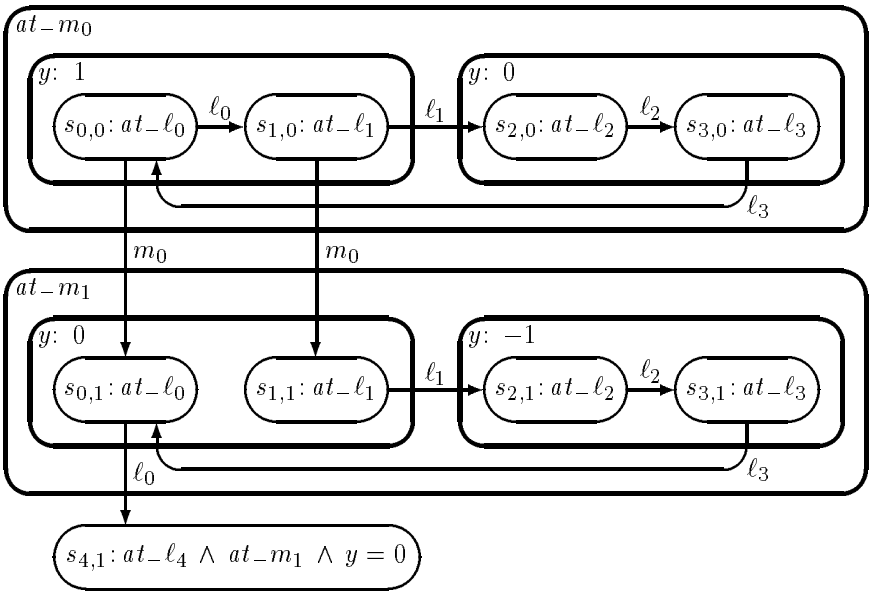


Fig. 3.50. A state-transition graph for program SIMPLE-MUX.

Assertion Q characterizes all the states which are reachable from an accessible $(p\colon\ at\_m_0)$-state by a $(q\colon\ at\_m_1)$-free segment. This includes the states

$$\|Q\|\colon\quad \{s_{0,0},\ s_{1,0},\ s_{2,0},\ s_{3,0}\}.$$

Therefore, we can take for Q the assertion

$$Q\colon\quad (at\_\ell_{0,1}\ \wedge\ y = 1\ \vee\ at\_\ell_{2,3}\ \wedge\ y = 0)\ \wedge\ at\_m_0.$$

Constructs $\sqsupset$, δ_π, h, δ, φ_i, and τ_i

The fair transitions of program SIMPLE-MUX consist of the just transitions ℓ_0, ℓ_1, ℓ_2, ℓ_3 and the compassionate transition m_0. We list them in the following order:

$$\mathcal{F}\colon\quad \{\tau^1\colon \ell_0,\ \ \tau^2\colon \ell_1,\ \ \tau^3\colon \ell_2,\ \ \tau^4\colon \ell_3,\ \ \tau^5\colon m_0\}.$$

Examining the graph of Fig. 3.50, where the set $\|Q\|$ occupies the top row of states, we observe that there is no Q-segment which gratifies all the five fair transitions.

In particular, no Q-segment gratifies $\tau^5\colon m_0$. It follows that the order $\sqsupset$ is empty, i.e.,

$$\sqsupset\colon \emptyset.$$

This makes the task of computing δ_π trivial, leading to

$$\delta_\pi(s) = 0 \quad \text{for all } s \in \|Q\|.$$

Next, we compute the height function $h(s)$. Since $\delta_\pi(s) = 0$ for all Q-states, there is a leveled Q-segment departing from each Q-state which gratifies the transitions $\{\ell_0, \ell_1, \ell_2, \ell_3\}$. The deficit of such a segment is 5, corresponding to the ungratified transition $\tau^5\colon m_0$. We conclude that

$$h(s) = 5 \quad \text{for all } s \in \|Q\|.$$

It follows that the full ranking function is

$$\delta(s) = (0, 5) \quad \text{for all } s \in \|Q\|.$$

Consequently, there exists a single intermediate assertion $\varphi = Q$ with a corresponding helpful transition $\tau = \tau^5\colon m_0$.

It is not difficult to check that all the premises of rule WELL-F, except CW4, are P-state valid. It only remains to prove

CW4. $$\underbrace{(at\_\ell_{0,1} \ \wedge \ y = 1 \ \vee \ at\_\ell_{2,3} \ \wedge \ y = 0) \ \wedge \ at\_m_0}_{\varphi = Q} \ \Rightarrow \ \diamondsuit\Big(\neg Q \ \vee \ \underbrace{at\_m_0 \ \wedge \ y = 1}_{En(m_0)}\Big).$$

Proving CW4

To prove CW4, we reapply rule WELL-F with

$$p^{-5} \ = \ Q\colon \quad (at\_\ell_{0,1} \ \wedge \ y = 1 \ \vee \ at\_\ell_{2,3} \ \wedge \ y = 0) \ \wedge \ at\_m_0$$
$$q^{-5} \quad : \quad \neg Q \ \vee \ (at\_m_0 \ \wedge \ y = 1)$$

over program SIMPLE-MUX^{-5} which is identical to program SIMPLE-MUX, except that $\tau^5\colon m_0$ is no longer considered compassionate. The removal of m_0 from the compassion list does not change the behavior of the program in the segments of interest, but has an effect on the auxiliary constructs.

Assertion Q^{-5}

Assertion Q^{-5} should characterize all states which are reachable from an accessible p^{-5}-state by a q^{-5}-free segment. It is obvious that

$$\|Q^{-5}\| \ = \ \{s_{2,0}, \ s_{3,0}\}.$$

Consequently,

$$Q^{-5}: \quad at\_\ell_{2,3} \ \wedge \ at\_m_0 \ \wedge \ y = 0.$$

Constructs $\sqsupset^{-5}$, δ_π^{-5}, h^{-5}, δ^{-5}, φ_i^{-5}, and τ_i^{-5}

Consider the q^{-5}-free segments departing from Q^{-5}-states. State $s_{2,0}$ initiates the segment $[s_{2,0}, s_{3,0}]$ which gratifies all the remaining four transitions ℓ_0, ℓ_1, ℓ_2, and ℓ_3. Consequently, we set

$$\sqsupset^{-5} \ = \ (s_{2,0}, \ s_{3,0})$$

which implies $s_{2,0} \sqsupset^{-5} s_{3,0}$.

Computing δ_π^{-5}, we get

$$\delta_\pi^{-5}(s_{2,0}) \ = \ 1 \qquad \text{and} \qquad \delta_\pi^{-5}(s_{3,0}) \ = \ 0.$$

It follows that $s_{2,0}$ has $[s_{2,0}]$ as the only departing leveled Q-segment with deficit 3, corresponding to the ungratified transition $\tau^3{:}\,\ell_2$. State $s_{3,0}$ has the departing leveled Q-segment $[s_{3,0}]$ with deficit 4, corresponding to $\tau^4{:}\,\ell_3$.

Consequently, we obtain two intermediate assertions with the following associated constructs:

i	φ_i^{-5}	δ^{-5}	Helpful transition τ_i^{-5}
1	$at\_\ell_2 \ \wedge \ at\_m_0 \ \wedge \ y = 0$	$(1, 3)$	ℓ_2
2	$at\_\ell_3 \ \wedge \ at\_m_0 \ \wedge \ y = 0$	$(0, 4)$	ℓ_3

It is not difficult to check that all premises of rule WELL-F are satisfied for this choice. Since the two helpful transitions are just, it is only necessary to check subpremise JW4. ∎

In **Problem 3.4**, we ask the reader to apply the construction outlined in the completeness proof to the full mutual-exclusion program MUX-SEM of Fig. 3.1.

Expressing the Constructs by a Formal Language

The completeness proof specified the necessary constructs by semantic means. We now show that all constructs can be expressed within an appropriate assertional language. We use some of the notation introduced in Section 2.5 of the SAFETY book.

The Kernel Assertion Q

Recall that a state s satisfies assertion Q if there exists a P-prefix $[s_1, \ldots, s_n]$, such that

- $s_n = s$, and
- for some $k \in [1..n]$, s_k satisfies p, and
- for all $i \in [k..n]$, s_i does not satisfy q.

We use the notations introduced in Section 2.5 of the SAFETY book to express Q for the case that the system variables of program P, $\overline{y} = \{y_1, \ldots, y_r\}$, range over data domains that are either natural numbers or encodable as natural numbers. We use a two-dimensional dynamic array $a \in \mathcal{A}^2(\mathbb{N})$ to encode sequences of states intended to represent prefixes and segments.

The complete formula is given by

$$Q(\overline{y}): \quad \exists n > 0: \exists a \in \mathcal{A}^2(\mathbb{N}): \Big(\mathit{reach}(\overline{y}) \ \wedge \ \exists k \in [1..n]: \big(p(a[k]) \ \wedge \ \mathit{q\text{-}free}\big)\Big),$$

where

$$\mathit{reach}(\overline{y}): \quad |a| = (n, r) \ \wedge \ \Theta\big(a[1]\big) \ \wedge \ a[n] = \overline{y} \ \wedge \ \mathit{evolve}(a, n)$$

$$\mathit{evolve}(a, n): \quad \forall i (1 \leq i < n): \bigvee_{\tau \in \mathcal{T}^-} \rho_\tau\big(a[i], a[i+1]\big)$$

$$\mathit{q\text{-}free}: \quad \forall i \in [k..n]: \neg q\big(a[i]\big).$$

The free variables in Q are the list of system variables $\overline{y} = y_1, \ldots, y_r$, representing the current state.

Segments and the Transitions they Gratify

In the following discussion we use two-dimensional dynamic arrays $b \in \mathcal{A}^2(\mathbb{N})$ of shape $|b| = (n, r)$ to represent segments.

The formula $\mathit{Qseg}(b, n, \overline{u})$ states that b encodes a Q-segment of length n, originating at state $\overline{u}$

$$\mathit{Qseg}(b, n, \overline{u}): \quad |b| = (n, r) \ \wedge \ b(1) = \overline{u} \ \wedge \ \mathit{evolve}(b, n) \ \wedge \ \forall i \in [1..n]: Q\big(b[i]\big).$$

The formula $\mathit{gratify}(b, i)$ states that segment b gratifies transition τ_i:

$$\begin{aligned}\mathit{gratify}(b, i): \quad \exists j \in [1..n-1]: \ & \rho_{\tau_i}\big(b[j], b[j+1]\big) \ \vee \\ & \Big(\tau_i \in \mathcal{C} \ \wedge \ \forall j \in [1..n]: \neg \mathit{En}\big(\tau_i, b[j]\big)\Big) \ \vee \\ & \Big(\tau_i \in \mathcal{J} - \mathcal{C} \ \wedge \ \exists j \in [1..n]: \neg \mathit{En}\big(\tau_i, b[j]\big)\Big),\end{aligned}$$

where

$$\mathit{En}(\tau_i, \overline{y}): \quad \exists \overline{y}': \ \rho_{\tau_i}(\overline{y}, \overline{y}').$$

The formula $\mathit{fair}(b)$ states that segment b is fair, i.e., gratifies all fair transitions

$$fair(b): \quad \bigwedge_{i=1}^{m} grat(b,i).$$

The Primary Order $\sqsupset$

Recall that $s_1 \sqsupset s_2$ if there exists a fair Q-segment leading from s_1 to s_2. Representing the state-variables at s_1 and at s_2 by $\overline{u}$ and $\overline{v}$, respectively, we define the corresponding formula $\overline{u} \sqsupset \overline{v}$

$$\overline{u} \sqsupset \overline{v}: \quad \exists n > 0: \ \exists b \in \mathcal{A}^2(\mathbb{N}): \ \Big(Qseg(b,n,\overline{u}) \ \wedge \ b[n] = \overline{v} \ \wedge \ fair(b) \Big).$$

The Primary Ranking δ_π

When we attempt to express the primary ranking δ_π, we find that the assertional language we used so far, a first-order language over the integers, is no longer adequate. Our failure to express the ranking δ_π by a first-order language is not accidental and there are theoretical arguments which shows that some extensions to the first-order language are necessary.

The extension we choose is one which leads to the most natural formalization of the construction of δ_π. We allow *least fixed points* over recursive function definitions. With this extension we can express the ranking δ_π as the least fixed point of the recursive equation

$$F(\overline{u}) \quad = \quad lub\big\{ F(\overline{v}) + 1 \mid \overline{v} \sqsubset \overline{u} \big\}.$$

Note that if $\overline{u}$ represents a $\sqsupset$-minimal state, then the least upper bound is taken over an empty set. The least ordinal which is bigger than all elements of the empty set is 0. This shows that $\delta_\pi(s) = 0$ for all $\sqsupset$-minimal states s.

Expressing the Height of a State

The formula $Lseg(b,n,\overline{u})$ identifies array b as representing a leveled Q-segment of length n, originating at state s, which is represented by system variables $\overline{u}$.

$$Lseg(b,n,\overline{u}): \quad Qseg(b,n,\overline{u}) \ \wedge \ \forall i \in [1..n-1]: \ \delta_\pi\big(a[i]\big) = \delta_\pi\big(a[i+1]\big).$$

The formula $has\text{-}def(\overline{u},j)$ states the existence of a leveled Q-segment originating at $\overline{u}$ with deficit $j \in [1..m]$

$$has\text{-}def(\overline{u},j): \quad \exists n > 0 \, \exists b \in \mathcal{A}^2(\mathbb{N}): \ Lseg(b,n,\overline{u}) \ \wedge \ \neg gratify(b,j) \ \wedge \ \bigwedge_{k<j} gratify(b,k).$$

That is, τ_j is not gratified in b but every τ_k, $1 \le k < j$, is gratified in b.

The height function $h(\overline{u})$ can now be written as follows

$$h(\overline{u}): \quad max\big\{ j \in [1..m] \mid has\text{-}def(\overline{u},j) \big\}.$$

This formula computes the maximal deficit over all leveled Q-segments departing from $\overline{u}$.

The Final Rank and Assertions

The final rank δ is given by

$$\delta(\overline{u}): \quad \big(\delta_\pi(\overline{u}),\ h(\overline{u})\big).$$

The intermediate assertions $\varphi_i(\overline{u})$, $i = 1, \ldots, m$, can be expressed by the following

$$\varphi_i(\overline{u}): \quad Q \ \wedge \ h(\overline{u}) = i.$$

** 3.9 Completeness: General Response

In the general case, we consider a response formula

$$p \ \Rightarrow \ \diamondsuit q,$$

where p and q are past formulas. In this case, the intermediate formulas $\varphi_1, \ldots, \varphi_m$ are also past formulas and all premises are to be written as entailments $\Rightarrow$. Note that the ranking function δ may also depend on the past.

Example As our running example for the general case, consider system INC presented in Fig. 3.51 (see also Fig. 4.2 of the SAFETY book).

V: $\{x$: **integer**$\}$
Θ: $x = 0$
$\mathcal{T}$: $\{\tau_I, \tau\}$ where ρ_τ: $x' = x + 1$
$\mathcal{J}$: $\{\tau\}$
$\mathcal{C}$: $\{\ \}$

Fig. 3.51. System INC.

We wish to establish for it the response property

$$\diamondminus(x = 5) \ \Rightarrow \ \diamondsuit \diamondminus(x = 10).$$

This property states that every position that has $x = 5$ in its past is followed by a position that has $x = 10$ in its past. ∎

The strategy we intend to follow in proving completeness for the general case is identical to the one followed in Section 4.9 of the SAFETY book, where we established completeness for past invariances.

Namely, given a response formula $p \Rightarrow \diamondsuit q$, where p and q are past formulas, and a program P, we construct statified versions of p, q, and P, denoted by $\widehat{p}$, $\widehat{q}$, and $\widehat{P}$, respectively. Assuming that $p \Rightarrow \diamondsuit q$ is P-valid, we conclude that $\widehat{p} \Rightarrow \diamondsuit \widehat{q}$ is $\widehat{P}$-valid. We invoke the proof of completeness for response over state formulas to construct intermediate assertions $\widehat{\varphi}_1, \ldots, \widehat{\varphi}_m$ and ranking $\widehat{\delta}$, satisfying premises FW1, FW2, FW3, and FW4 of rule WELL-F. As a last step we unstatify $\widehat{\varphi}_1, \ldots, \widehat{\varphi}_m$, and $\widehat{\delta}$ to obtain past formulas $\varphi_1, \ldots, \varphi_m$ and a (possibly) past dependent ranking δ.

Since these steps are very similar to the corresponding steps taken in Section 4.9 of the SAFETY book, we provide only a sketch and concentrate on a few minor differences between the two cases.

Statifying p, q, and Program P

The base for statification in Section 4.9 of the SAFETY book was the closure Φ_ψ which contains all subformulas of ψ whose principal operator is a past operator. Here, we need to statify both p and q at the same time. Consequently, we base statification on the set $\Phi_{p,q}$ which contains all subformulas of p and q with a past principal operator.

Using $\Phi_{p,q}$ we statify p, q, and P.

Example Consider property $\underbrace{\diamondminus(x = 5)}_{p} \Rightarrow \diamondsuit \underbrace{\diamondminus(x = 10)}_{q}$ for system INC. The closure of p and q is given by

$$\Phi_{p,q}: \quad \{\diamondminus(x = 5),\ \diamondminus(x = 10)\}.$$

The statification transformation can be defined as

$$stat(\varphi): \quad \varphi[b_5/\diamondminus(x = 5),\ b_{10}/\diamondminus(x = 10)].$$

Consequently, we have $\widehat{p}$: b_5 and $\widehat{q}$: b_{10}.

The statified system $\widehat{\text{INC}}$ is presented in Fig. 3.52.

Thus, we have reduced the task of proving $\diamondminus(x = 5) \Rightarrow \diamondsuit \diamondminus(x = 10)$ over INC to the task of proving $b_5 \Rightarrow \diamondsuit b_{10}$ over $\widehat{\text{INC}}$. ∎

Invoking the Completeness Proof for State Response

Faced with the task of verifying $\widehat{p} \Rightarrow \diamondsuit \widehat{q}$ over $\widehat{P}$, we invoke the completeness proof for state response formulas. This invocation results in intermediate assertions $\widehat{\varphi}_1, \ldots, \widehat{\varphi}_m$, and ranking $\widehat{\delta}$ satisfying premises FW1, FW2, FW3, and FW4 of rule WELL-F.

$\widehat{V}$: $\{x$: **integer**; $b_5,\ b_{10}$: **boolean**$\}$

$\widehat{\Theta}$: $x = 0 \ \wedge \ b_5 = b_{10} = \text{F}$

$\widehat{\mathcal{T}}$: $\{\widehat{\tau}_I, \widehat{\tau}\}$ with transition relation (after simplification)

$\rho_{\widehat{\tau}}$: $x' = x + 1 \ \ \wedge \ \ b'_5 = (x = 4 \ \vee \ b_5) \ \ \wedge \ \ b'_{10} = (x = 9 \ \vee \ b_{10})$

$\widehat{\mathcal{J}}$: $\{\widehat{\tau}\}$

$\widehat{\mathcal{C}}$: $\emptyset$

Fig. 3.52. Statified system $\widehat{\text{INC}}$.

Example Applying the completeness proof to property $b_5 \Rightarrow \diamondsuit\, b_{10}$ over system $\widehat{\text{INC}}$ may result in a single intermediate assertion $\widehat{\varphi}$ and the ranking $\widehat{\delta}$

$$\widehat{\varphi}: \quad x \geq 0 \ \wedge \ (b_5 \ \leftrightarrow \ x \geq 5) \ \wedge \ (b_{10} \ \leftrightarrow \ x \geq 10) \ \wedge \ (b_5 \ \wedge \ \neg b_{10})$$

$$\widehat{\delta}(x, b_5, b_{10}): \quad \text{if } \widehat{\varphi} \text{ then } (9 - x, 1) \text{ else } (0, 0).$$

Note that if we were doing a manual proof of the considered property, it would have been sufficient to take $5 \leq x < 10$ for $\widehat{\varphi}$ and $max(9 - x, 0)$ for $\widehat{\delta}(x)$. The constructs $\widehat{\varphi}$ and $\widehat{\delta}$ displayed above are (equivalent to) the ones that are actually constructed by the completeness proof. For example, they only allow accessible states in $\widehat{\varphi}$. ∎

Besides providing the constructs $\widehat{\varphi}_1, \ldots, \widehat{\varphi}_m$, and $\widehat{\delta}$, the state-response completeness proof provided us with a detailed deductive proof of $\widehat{p} \Rightarrow \diamondsuit\, \widehat{q}$ over program $\widehat{P}$. This proof consists of repeated applications of rule WELL-F, which establish the $\widehat{P}$-state validity of various instances of premises FW1, FW2, FW3, and subpremise JW4, all of which are state implications. Consequently, the steps in the proof of $\widehat{P} \vDash \widehat{p} \Rightarrow \diamondsuit\, \widehat{q}$ alternate between establishing the $\widehat{P}$-state validity of a state implication and invocations of rule WELL-F.

Unstatifying

As the final step, we unstatify the constructs $\widehat{\varphi}_1, \ldots, \widehat{\varphi}_m$, and $\widehat{\delta}$ into $\varphi_1, \ldots, \varphi_m$, and δ. Subsequently, we show that each of the premises, previously shown to be a $\widehat{P}$-state valid implication, is transformed into an entailment of past formulas which can be shown to be P-valid using inverse statification, rule IGEN, and entailment reasoning. This step is very similar to the corresponding step in Section 4.9 of the SAFETY book. Details are therefore omitted here.

Example Unstatifying the obtained constructs for system $\widehat{\text{INC}}$, we obtain the following

$$\varphi\colon\quad x \geq 10 \ \wedge\ \big(\diamondsuit(x = 5) \ \leftrightarrow\ x \geq 5\big) \ \wedge\ \big(\diamondsuit(x = 10) \ \leftrightarrow\ x \geq 10\big) \wedge \big(\diamondsuit(x = 5) \ \wedge\ \neg\, \diamondsuit(x = 10)\big)$$

$$\delta(x)\colon\quad \text{if } \varphi \text{ then } (9 - x, 1) \text{ else } (0, 0).$$ ∎

3.10 Finite-State Algorithmic Verification: State Response

In this and the next section we present an algorithm for checking whether a finite-state program satisfies a response property $p \Rightarrow \diamondsuit\, q$. Consistent with our general strategy, we consider in this section the case that p and q are assertions. In the next section, we study the extensions necessary to handle the more general case in which p and q may be general past formulas.

The algorithm searches for a counter-example. That is, it looks for a computation σ and a position $j \geq 0$ such that p holds at j but q holds at no position $k \geq j$.

Even for finite-state programs, a computation is an infinite object and, usually, there are uncountably many different computations. Consequently, we cannot simply enumerate all possible computations and check each of them for being a counter-example. Instead, we will analyze the state-transition graph G_P, which is always finite for a finite-state program P. Recall that G_P is a graph containing all the P-accessible states as nodes, and edges connecting s_1 to s_2 if s_2 is a τ-successor of s_1 for some $\tau \in \mathcal{T}$. In this section, we only consider states appearing in G_P. Therefore, any reference to a state s implies that s is P-accessible.

Let σ be a computation. Define $\mathit{Inf}(\sigma)$, called the *infinity set* of σ, to be the set of states that appear at infinitely many positions of σ. Since there are only finitely many P-accessible states, there exists a position $f \geq 0$ in σ such that

$$s \in \mathit{Inf}(\sigma) \quad \text{if and only if} \quad s = s_j \text{ for some position } j \geq f \text{ in } \sigma.$$

Recall that a subgraph $S \subseteq G_P$ is called a *strongly connected subgraph* (SCS) if, for every two distinct states $s_1, s_2 \in S$, there exists a path in S, leading from s_1 to s_2.

Since $\mathit{Inf}(\sigma)$ contains only P-accessible states, we can view it as a subgraph of G_P, called the subgraph *induced* by σ.

Claim 3.6 (from computations to subgraphs)

$\mathit{Inf}(\sigma)$ is an SCS.

Justification Let $s, \widehat{s} \in \mathit{Inf}(\sigma)$ be two different states. Since $s \in \mathit{Inf}(\sigma)$, there exists a position $j \geq f$ such that $s_j = s$. As $\widehat{s}$ also appears infinitely many times

in σ, there exists a later position $k > j$ such that $s_k = \widehat{s}$. Consider the segment $s{=}s_j, s_{j+1}, \ldots, s_k{=}\widehat{s}$ in σ. For every $i \in [j..k]$, $i \geq f$. Consequently, $s_i \in \mathit{Inf}(\sigma)$ for every $i \in [j..k]$ and the path $[s_j, \ldots, s_k]$ leading from s to $\widehat{s}$ is contained in $\mathit{Inf}(\sigma)$. ∎

Example Consider system INC2 presented in Fig. 3.53.

V: $\{x\colon [0..7]\}$

Θ: $x = 0$

$\mathcal{T}$: $\{\tau_I, \tau_1, \tau_2, \tau_3\}$ where $\rho_{\tau_1}\colon\ x' = x \oplus_8 1$
$\rho_{\tau_2}\colon\ even(x) \ \wedge\ x' = x \oplus_8 2$
$\rho_{\tau_3}\colon\ odd(x) \ \wedge\ x' = x \oplus_8 2$

$\mathcal{J}$: $\{\tau_1, \tau_2\}$

$\mathcal{C}$: $\{\tau_3\}$

Fig. 3.53. System INC2.

In Fig. 3.54, we present the state transition graph for INC2.

Consider the computation

$$\sigma\colon\quad s_0[s_2,\ s_4,\ s_6,\ s_0]^k\ [s_2,\ s_3,\ s_5, s_7,\ s_0]^\omega$$

which consists of s_0, followed by $k \geq 0$ repetitions of the segment $[s_2, s_4, s_6, s_0]$, followed by infinitely many repetitions of the segment $[s_2, s_3, s_5, s_7, s_0]$. Independent of how large k is,

$$\mathit{Inf}(\sigma)\ =\ \{s_0,\ s_2,\ s_3,\ s_5,\ s_7\}$$

since these are the states that appear infinitely many times in σ. ∎

Fair Subgraphs

Not every SCS can be obtained as the infinity set $\mathit{Inf}(\sigma)$ of a computation σ. Consider, for example, the set S: $\{s_0, s_2, s_4, s_6\}$ of Fig. 3.54. This set cannot be the infinity set of a computation. This is because every sequence σ such that $\mathit{Inf}(\sigma) = S$ has transition τ_1 continuously enabled and never taken beyond position f (the position such that $s_j \in \mathit{Inf}(\sigma)$ for every $j \geq f$). Thus, any run σ such that $\mathit{Inf}(\sigma) = S$ is necessarily unjust towards transition τ_1.

Consequently, we characterize those SCS's that can be $\mathit{Inf}(\sigma)$ for some computation σ.

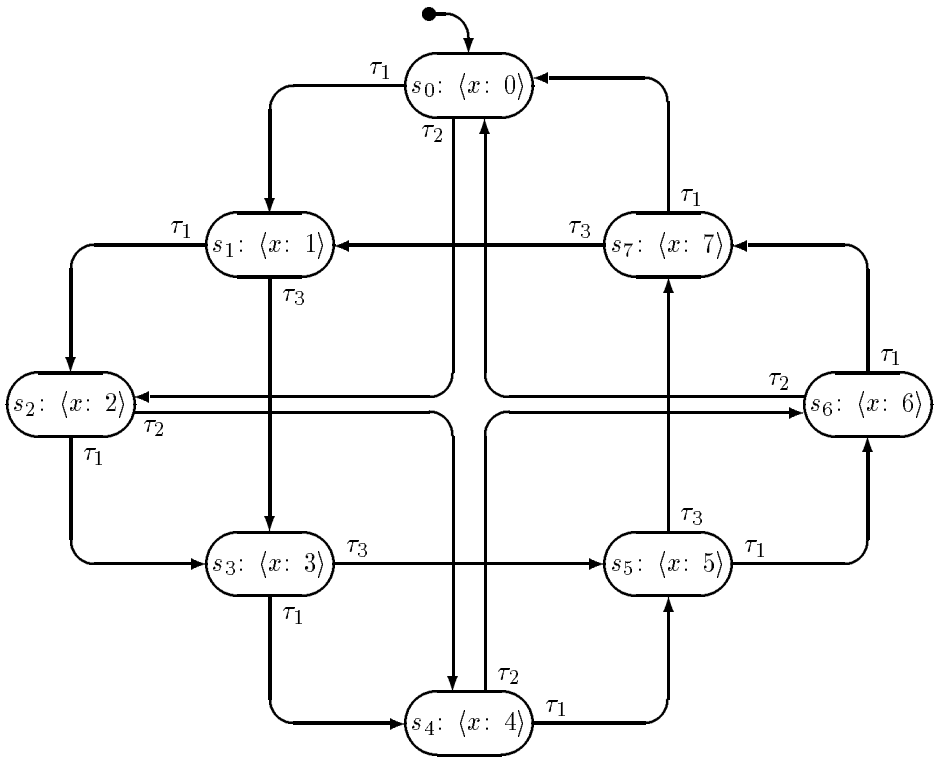


Fig. 3.54. State transition graph for INC2.

Let S be a subgraph of G_P and τ a transition. We say that τ is *taken in* S if S contains two states, s_1 and s_2, such that s_1 is connected by a directed edge to s_2 and $s_2 \in \tau(s_1)$. Transition τ is said to be *enabled* (*disabled*) *in* S if there exists an $s \in S$ such that τ is enabled (disabled) on s.

A subgraph S is called *just* if every just transition $\tau \in \mathcal{J}$ is either taken in S or is disabled in S. For example, subgraph $\{s_0, s_1, s_3, s_5, s_7\}$ of Fig. 3.54 is just. It is just towards τ_1 because $s_1 \in \tau_1(s_0)$. It is just towards τ_2 since τ_2 is disabled on s_1.

As previously observed, subgraph $\{s_0, s_2, s_4, s_6\}$ is not just towards τ_1, which is enabled on all of its states but not taken.

A subgraph S is called *compassionate* if every compassionate transition $\tau \in \mathcal{T}$ is either taken in S or is not enabled in S. *Not enabled* means that τ is disabled on all states $s \in S$.

For example, subgraph $\{s_0, s_2, s_4, s_6\}$ is compassionate but not just. It fulfills

its obligations to the only compassionate transition τ_3, since τ_3 is disabled on all states in the subgraph. On the other hand, subgraph $\{s_0, s_1, s_2, s_4, s_6\}$ is just but not compassionate. This is because τ_3 is enabled on state s_1 of the subgraph but is not taken in the subgraph.

A subgraph is called *fair* if it is both just and compassionate. Thus, subgraph $\{s_0, s_1, s_3, s_5, s_7\}$ of Fig. 3.54 is fair.

Traversing Cycle

Let S be an SCS of G_P. A path $\sigma: [s_a, \ldots, s_b]$ in the subgraph S is called a *traversing cycle* of S if it satisfies the following requirements:

- $s_a = s_b$.
- For every $s \in S$, $s = s_i$ for some $i \in [a..b]$.
- For every pair of states $s, \widehat{s} \in S$ that are connected by an edge, there exists an $i \in [a..b-1]$ such that $s = s_i$ and $\widehat{s} = s_{i+1}$.

Thus, σ visits every state $s \in S$ at least once and traverses every edge connecting two states in S at least once.

Consider, for example, subgraph $\{s_0, s_1, s_2, s_4, s_6\}$. The shortest traversing cycle originating at s_0 is

$$s_0,\ s_1,\ s_2,\ s_4,\ s_6,\ s_0,\ s_2,\ s_4,\ s_6,\ s_0.$$

Note that we have to go twice around the subgraph in order to traverse both edge $\langle s_0, s_1 \rangle$ and $\langle s_0, s_2 \rangle$.

It is not too difficult to see that every SCS with n nodes has a traversing cycle whose size has a worst case complexity of $O(n^3)$.

The following claim establishes a correspondence between computations and fair subgraphs.

Claim 3.7 (fair SCS's and computations)

An SCS is the infinity set of a computation iff it is fair.

Justification Let σ be a computation and $S = Inf(\sigma)$ its infinity set. We will show that S is fair.

Consider a just transition $\tau \in \mathcal{J}$. Being a computation, σ is just towards τ which means that either τ is disabled on a state s and $s = s_i$ for infinitely many i's, or there exists a pair of states $\widehat{s} \in \tau(\widetilde{s})$ such that $\widetilde{s} = s_i$ and $\widehat{s} = s_{i+1}$ for infinitely many i's. In the first case, $s \in S$ and, therefore, τ is disabled in S. In the second case, both $\widetilde{s}, \widehat{s} \in S$ and τ is taken in S. We conclude that S is just.

Next, consider a compassionate transition $\tau \in \mathcal{T}$. As σ is compassionate towards τ, either τ is continually disabled from some point on, or τ is taken infinitely many times in σ. In the first case, τ is disabled on all states in $S = \mathit{Inf}(\sigma)$. In the second case, S contains a pair of states $\widehat{s} \in \tau(\widetilde{s})$, which means that τ is taken in S. We conclude that S is compassionate.

Consequently, $S = \mathit{Inf}(\sigma)$ is fair.

In the other direction, consider a fair strongly connected subgraph S. Let it have the traversing cycle

$$s_a,\ \ldots,\ s_b\ =\ s_a\,.$$

Since all states in G_P are accessible, there exists a path

$$s_0,\ s_1,\ \ldots,\ s_a$$

leading from some initial state s_0 to s_a.

Consider the state sequence

$$\sigma\colon\quad s_0,\ \ldots,\ s_a,\ [s_{a+1},\ \ldots,\ s_b]^\omega$$

which consists of the initial prefix $s_0, \ldots, s_a$ followed by endless repetition of the segment $s_{a+1}, \ldots, s_b$. It is not difficult to verify that σ is a computation of program P. In particular, the fact that σ satisfies the requirements of justice and compassion follows from the fairness of S and the property that $s_a, \ldots, s_b$ visits every state in S and traverses every edge in S at least once. ∎

For example, an INC2-computation σ such that $\mathit{Inf}(\sigma) = \{s_0, s_1, s_3, s_5, s_7\}$ can be given by

$$\sigma\colon\quad s_0,\ [s_1,\ s_3,\ s_5,\ s_7,\ s_1,\ s_3,\ s_5,\ s_7,\ s_0]^\omega.$$

Computation σ satisfies the fairness requirement by taking τ_1 and τ_3 infinitely many times. Just transition τ_2 is not taken at all, but this is allowed since it is infinitely often disabled — at all visits to states s_1, s_3, s_5, and s_7.

Subgraphs Corresponding to Counter-Examples

The correspondence between computations and fair SCS's raises the possibility that instead of examining uncountably many computations we may have to inspect only the fair subgraphs of G_P, whose number is finite.

A path $[s_a, \ldots, s_b]$ in graph G_P is called q-free, if s_i satisfies q for no $i \in [a..b]$. A state s is called *q-pending* if there exists a q-free path $[s_a, \ldots, s_b]$ such that $s_b = s$ and s_a satisfies p.

Assume that property $p \Rightarrow \diamondsuit\, q$ is P-valid. An important property of q-pending states is that if state s_i in a computation of P is q-pending it must be followed by a later state s_j, $j > i$, satisfying q. The reason for this is that these

states characterize situations in which we have already observed an occurrence of p but not yet the corresponding q, promised by the P-validity of $p \Rightarrow \diamondsuit q$. In that respect, they generalize the family of p-states, since for all of these states there is still a pending occurrence of q.

Example Consider the response property

$$\underbrace{x = 0}_{p} \quad \Rightarrow \quad \diamondsuit \underbrace{5 \leq x \leq 6}_{q}$$

for system INC2 (Fig. 3.53). The set of q-pending states for this choice of p and q is

$$\{s_0,\ s_1,\ s_2,\ s_3,\ s_4\}.$$

States s_5 and s_6 are not included since they satisfy q. State s_7 is not included since every path connecting s_0 to s_7 must pass through either s_5 or s_6. ▟

When we want to check whether $p \Rightarrow \diamondsuit q$ is P-valid, we refer to a computation σ that does not satisfy $p \Rightarrow \diamondsuit q$ as a *counter-example*. It is not difficult to see that

σ is a counter-example iff
all states from a certain position on are q-pending iff
all states in $Inf(\sigma)$ are q-pending.

Since we are interested in counter-examples, we define the *Q-graph* of program P, denoted Q_P, to be the subgraph of G_P consisting of the q-pending states in G_P. The following algorithm describes an efficient calculation of Q_P, given G_P.

Algorithm Q-GRAPH (calculation of Q_P)

Let Q_P be a copy of G_P.

Remove from Q_P all q-states.

Remove from Q_P all states not reachable from a p-state.

Example Let us calculate Q_P for system INC2 for p: $x = 0$ and q: $5 \leq x \leq 6$. As a first step, we remove the q-states s_5 and s_6. Next we identify the p-reachable states by successive marking. This procedure marks states according to the sequence s_0, s_1, s_2, s_3, s_4. State s_7 remains unmarked and is removed. We remain with Q_P: $\{s_0, s_1, s_2, s_3, s_4\}$. ▟

The preceding discussion and sequence of claims lead to a conclusion that is summarized by the following proposition.

Claim 3.8 (checking for validity)

The graph Q_P contains a fair SCS iff the property $p \Rightarrow \diamondsuit q$ is not P-valid.

Justification Assume that Q_P contains a fair SCS S. Following the construction described in Claim 3.7, there exists a P-computation σ such that $Inf(\sigma) = S$. Since all but finitely many states in σ are q-pending, σ is a counter-example which violates $p \Rightarrow \diamondsuit q$.

In the other direction, if σ is a counter-example which violates $p \Rightarrow \diamondsuit q$ then $S = Inf(\sigma)$ is a fair SCS of G_P consisting of q-pending states. Consequently, it is also a subgraph of Q_P. ∎

Example Consider the property

$$\varphi: \quad \underbrace{x = 0}_{p} \quad \Rightarrow \quad \diamondsuit \underbrace{x = 4}_{q}$$

for program INC2 (Fig. 3.53). We wish to check whether property φ is P-valid. Calculation of Q_P shows that all states, excluding s_4, are q-pending. Consequently, by Claim 3.8, φ is valid over INC2 iff there does not exist a fair SCS of $Q_P = G_P - \{s_4\}$. Obviously, $\{s_0, s_1, s_3, s_5, s_7\}$ is such a subgraph. We conclude that $(x = 0) \Rightarrow \diamondsuit(x = 4)$ is not valid over INC2 and can actually point to the computation

$$\sigma: \quad [s_0,\ s_1,\ s_3,\ s_5,\ s_7]^\omega$$

as a counter-example.

On the other hand, let us consider another property

$$\psi: \quad \underbrace{x = 0}_{p} \quad \Rightarrow \quad \diamondsuit \underbrace{odd(x)}_{q} \ .$$

Calculating Q_P we obtain Q_P: $\{s_0, s_2, s_4, s_6\}$. The only SCS of Q_P is Q_P itself. However, Q_P is not fair since the just transition τ_1 is enabled on all states of Q_P but is not taken in Q_P.

We conclude that $(x = 0) \Rightarrow \diamondsuit\ odd(x)$ is valid over INC2. ∎

In principle, we can consider the problem solved. Since Q_P contains only finitely many subgraphs, and each can be checked for being strongly connected and fair in a finite number of steps, we can suggest an algorithm based on this procedure.

Unfortunately, the resulting algorithm will not be very efficient, since a graph of size n may contain an exponential number of SCS's. We therefore study a more efficient algorithm.

An Efficient Algorithm

Let S be a strongly connected graph. We wish to check whether S is fair or contains a fair SCS.

It is straightforward to check whether S is just. For each $\tau \in \mathcal{J}$, we check that τ is disabled in S or taken in S. If S is not just then no subgraph of S can be just. To see this, we observe that if some subgraph $S' \subseteq S$ treats τ justly, that is, τ is disabled or taken in S', then so does S. Consequently, if $S' \subseteq S$ is just then so is S and, equivalently, if S is unjust it cannot contain a just subgraph.

Assume we found S to be just. Next, we check whether S is compassionate, i.e., if it is the case that every compassionate transition $\tau \in \mathcal{C}$ is either taken in S or is disabled on all states of S. If S is compassionate, we conclude that it is fair. However, if S is not compassionate it may still, unlike the justice case, contain a compassionate subgraph.

For example, considering graph G_P of Fig. 3.54, the subgraph S: $\{s_0, s_1, s_2, s_4, s_6\}$ is not compassionate towards τ_3 which is enabled on s_1 but not taken in S. On the other hand, the subgraph of S, S': $\{s_0, s_2, s_4, s_6\}$ is compassionate since τ_3 is disabled on all of its states.

If S was found to be uncompassionate, there exists some transition $\tau_k \in \mathcal{C}$ which is not taken in S but is enabled on some states in S. Let $EN(\tau_k, S)$ denote the set of S-states on which τ_k is enabled. Clearly, no compassionate subgraph $S' \subseteq S$ can contain any of these states. Therefore, it is safe to remove $EN(\tau_k, S)$ from our consideration. Let $U = S - EN(\tau_k, S)$. We should check whether U contains a fair SCS. Removing the set $EN(\tau_k, S)$ from S may lead to a subgraph U which is no longer strongly connected. Therefore, before proceeding, we decompose U into a sequence of maximal strongly connected subgraphs (MSCS's) $U_1, \ldots, U_k$. There exist several efficient algorithms for such decomposition, and we refer by the generic name DECOMPOSE to any of them. We used such an algorithm already in Section 3.6 of the SAFETY book. We proceed to test recursively each U_i, $i = 1, \ldots, k$, checking whether it contains a fair SCS.

We summarize these steps in algorithm FAIR-SUB which accepts as input an SCS S and produces as an output a fair SCS $S' \subseteq S$ if one exits, or the empty set $\emptyset$ if S does not contain a fair SCS.

Algorithm FAIR-SUB (fair SCS's)

Recursive Algorithm *fair-sub*(S: subgraph) **returns** subgraph

If S is not just **return** $\emptyset$ — failure

If S is compassionate **return** S — S is fair

Otherwise, there exists $\tau \in \mathcal{C}$ such that τ is not taken in S and $EN(\tau, S) \neq \emptyset$

Let $U = S - EN(\tau, S)$

Decompose U into MSCS's $U_1, \ldots, U_k$

For each $i = 1, \ldots, k$ **do**
  if $V_i = \mathit{fair\text{-}sub}(U_i) \neq \emptyset$ **then return** V_i — fair subgraph

end-for
return $\emptyset$. — no fair subgraphs.

Justification It is not difficult to see that if *fair-sub*(S) returns a nonempty subgraph $S' \subseteq S$ then S' is fair.

We concentrate on showing that if S contains a fair subgraph $S' \subseteq S$ then *fair-sub*(S) returns some fair subgraph $S'' \subseteq S$, which need not be identical to S'. We prove this by induction on $n = |S|$, the number of states in S.

For $|S| = n = 1$, S can contain a fair subgraph $S' \subseteq S$ only if $S' = S$ itself is fair. For this case, the first two tests in *fair-sub* will find that S is both just and compassionate and return it as the result.

Assume that the inductive claim is true for all subgraphs of size not exceeding n, and consider a subgraph S of size $n+1$ that contains a fair subgraph $S' \subseteq S$. If S itself is fair then *fair-sub* will return S after the first two tests. Otherwise, the algorithm will find S to be just but not compassionate and identify a compassionate transition $\tau \in \mathcal{C}$ which is not taken in S and such that $En(\tau, S) \neq \emptyset$. Subgraph S' being fair, it must treat τ with compassion, which implies that S' cannot contain any state $s \in En(\tau, S)$. Consequently, S' is contained in $U = S - En(\tau, S)$. Decomposing U into MSCS's $U_1, \ldots, U_k$, we observe that S', which is strongly connected, must be fully contained in U_j, for some $j \in [1..k]$. Since $|U_j| \leq |S - En(\sigma, S)| < n+1$, the induction hypothesis holds for U_j which contains a fair subgraph. Consequently, if the algorithm does not return a nonempty result for some $i < j$, it will invoke *fair -sub*(U_i) and obtain, by the induction hypothesis, a nonempty fair subgraph V_i as a result. This V_i is the result of *fair -sub*(S). ∎

Example Consider the state-transition graph presented in Fig. 3.55.

This graph represents a finite-state fair transition system whose fairness sets are

$$\mathcal{J}\colon \{\tau_1\}$$
$$\mathcal{C}\colon \{\tau_2,\ \tau_3\}.$$

We wish to check whether the property

$$\underbrace{x = 0}_{p} \ \Rightarrow \ \diamondsuit \underbrace{x = 5}_{q}$$

is valid over system ROUND. As a first step, we identify the set of q-pending states, which is $Q_P\colon \{s_0, s_1, s_2, s_3, s_4\}$. Since Q_P is strongly connected we can apply Algorithm FAIR-SUB. We find that Q_P is just but not compassionate. In particular it does not treat τ_3 with compassion. Transition τ_3 is enabled on s_3 but not taken in Q_P. Consequently, we consider $U = Q_P - En(\tau, Q_P) = Q_P - \{s_3\} = \{s_0, s_1, s_2, s_4\}$. Subgraph U is not strongly connected and it is decomposed, using

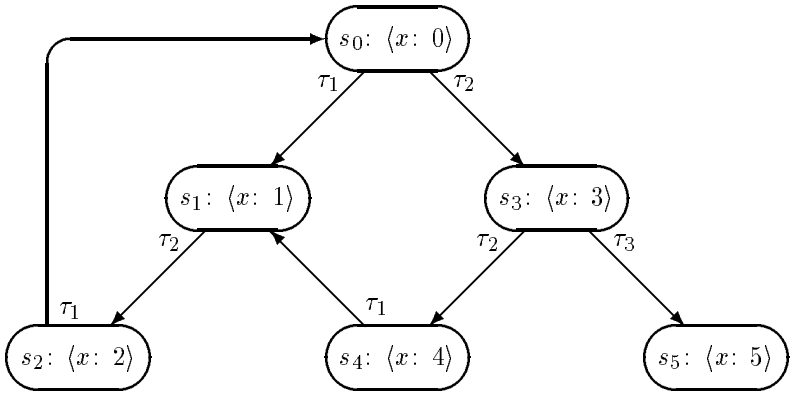


Fig. 3.55. System ROUND.

Algorithm DECOMPOSE, into the MSCS's U_1: $\{s_4\}$ and U_2: $\{s_0, s_1, s_2\}$.

In the next step, we apply FAIR-SUB to U_1: $\{s_4\}$. This subgraph is not just since τ_1 is enabled on all of its states but not taken in U_1. Consequently, $\mathit{fair\text{-}sub}(\{s_4\})$ returns $\emptyset$, denoting a failure to locate a fair SCS within U_1.

Applying FAIR-SUB to U_2: $\{s_0, s_1, s_2\}$ we observe that this subgraph is both just and compassionate and is therefore fair. It is compassionate towards τ_3 by having τ_3 disabled on all states of U_2.

We therefore conclude that property $x = 0 \Rightarrow \diamondsuit(x = 5)$ is not valid over system ROUND. A counter-example is provided by any sequence that endlessly traverses U_2. For example, σ: $[s_0, s_1, s_2]^\omega$ is a computation not satisfying $x = 0 \Rightarrow \diamondsuit(x = 5)$. ▟

We can now combine all the constituents into algorithm STATE-RESP for checking validity of a response formula over a finite state system.

Algorithm STATE-RESP (algorithmic verification of state formulas)

To check whether formula φ: $p \Rightarrow \diamondsuit q$ is valid over finite-state system P,

- Construct the state-transition graph G_P.
- Construct Q_P, the graph of q-pending states, using Algorithm Q-GRAPH.
- Decompose Q_P into MSCS's $S_1, \ldots, S_t$.
- For each $i = 1, \ldots, t$, apply $\mathit{fair\text{-}sub}(S_i)$.
  If any of these applications returns with a nonempty result, formula φ is not P-valid. A counter-example can be constructed from the returned

fair subgraph.
If all applications return the empty set as result, formula φ is P-valid.

Example (checking accessibility for MUX-SEM)

Let us consider the finite-state program MUX-SEM of Fig. 3.56 (see also Fig. 3.1). We wish to model-check whether it satisfies the property of accessibility

local y: **integer where** $y = 1$

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \textbf{request } y \\ \ell_3\text{: } \textbf{critical} \\ \ell_4\text{: } \textbf{release } y \end{array}\right] \end{array}\right] \;\Big\|\; P_2 :: \left[\begin{array}{l} m_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{noncritical} \\ m_2\text{: } \textbf{request } y \\ m_3\text{: } \textbf{critical} \\ m_4\text{: } \textbf{release } y \end{array}\right] \end{array}\right]$$

Fig. 3.56. Program MUX-SEM (mutual exclusion by semaphores).

$$\underbrace{at\_\ell_2}_{p} \;\Rightarrow\; \diamondsuit \underbrace{at\_\ell_3}_{q} \;.$$

We already model-checked this program for the safety property of mutual exclusion in Section 2.6 of the SAFETY book. During this check we constructed the state-transition graph for MUX-SEM as presented in Fig. 2.27 of the SAFETY book.

Following algorithm STATE-RESP, we construct Q_P, the graph of q-pending states for p: at_ℓ_2 and q: at_ℓ_3. This graph is presented in Fig. 3.57.

Analyzing the strongly connected graph Q_P, we find that it is just but not compassionate, since the compassionate transition ℓ_2 is enabled on states s_0, s_1, and s_2. Removing these states, we are left with two SCS's $\{m_3\}$ and $\{m_4\}$. Neither of these subgraphs is just. Transition m_3 is enabled but not taken in $\{m_3\}$ and transition m_4 is enabled but not taken in $\{m_4\}$.

Thus Q_P contains no fair SCS. We conclude that $at\_\ell_2 \Rightarrow \diamondsuit\, at\_\ell_3$ is valid over program MUX-SEM. ∎

Example (checking accessibility for MUX-PET1)

Consider program MUX-PET1 of Fig. 3.58 (see also Fig. 1.13). In Section 2.6 of the SAFETY book we analyzed its safety property of mutual exclusion and constructed in Fig. 2.28 of the SAFETY book its state-transition graph.

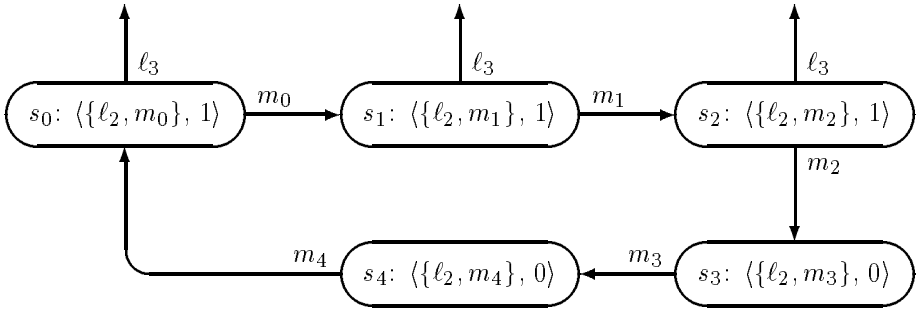


Fig. 3.57. Graph Q_P of q-pending states of program MUX-SEM.

local $y_1,\ y_2$: **boolean where** $y_1 = y_2 = \text{F}$
s : **integer where** $s = 1$

$$P_1 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } (y_1,\ s) := (\text{T},\ 1) \\ \ell_3\text{: } \textbf{await } \neg y_2 \vee s \neq 1 \\ \ell_4\text{: } \textbf{critical} \\ \ell_5\text{: } y_1 := \text{F} \end{array}\right] \end{array}\right]$$

$$\|$$

$$P_2 :: \left[\begin{array}{l} m_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} m_1\text{: } \textbf{noncritical} \\ m_2\text{: } (y_2,\ s) := (\text{T},\ 2) \\ m_3\text{: } \textbf{await } \neg y_1 \vee s \neq 2 \\ m_4\text{: } \textbf{critical} \\ m_5\text{: } y_2 := \text{F} \end{array}\right] \end{array}\right]$$

Fig. 3.58. Program MUX-PET1 (Peterson's algorithm) — version 1.

Here we wish to check the response property of accessibility

$$\underbrace{at\_\ell_2}_{p} \ \Rightarrow \ \diamondsuit \underbrace{at\_\ell_4}_{q}$$

over program MUX-PET1.

In Fig. 3.59 we present graph Q_P of q-pending states for p: at_ℓ_2 and q: at_ℓ_4 as extracted from the graph of Fig. 2.28 of the SAFETY book. Each node in this graph is a triple (i, j, k) representing a state in which π: $\{\ell_i, m_j\}$ and s: k.

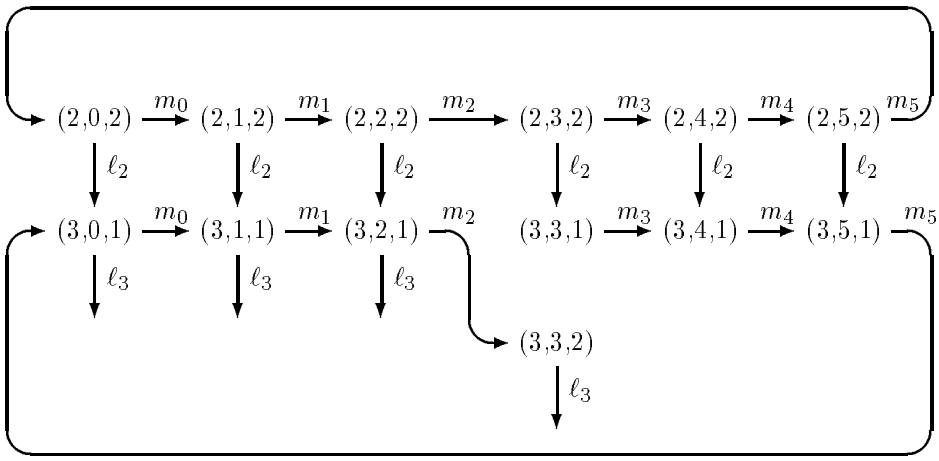


Fig. 3.59. Graph Q_P of q-pending states for MUX-PET1.

Decomposing Q_P into a sequence of MSCS's, we obtain the following list

$$S_1\colon \{(2,0,2),\ (2,1,2),\ (2,2,2),\ (2,3,2),\ (2,4,2),\ (2,5,2)\},$$
$$S_2\colon \{(3,3,1)\},\quad S_3\colon \{(3,4,1)\},\quad S_4\colon \{(3,5,1)\},\quad S_5\colon \{(3,0,1)\},$$
$$S_6\colon \{(3,1,1)\},\quad S_7\colon \{(3,2,1)\},\quad S_8\colon \{(3,3,2)\}.$$

None of these subgraphs is just. Subgraph S_1 is not just because ℓ_2 is enabled on all of its states and not taken in S_1. Subgraphs S_2, S_3, and S_4 are unjust because transitions m_3, m_4, and m_5 are enabled but not taken in them. Each of S_5, S_6, S_7, or S_8 is unjust because transition ℓ_3 is enabled but not taken in it.

We conclude that the accessibility property

$$at\_\ell_2 \ \Rightarrow\ \diamondsuit\, at\_\ell_4$$

is valid over program MUX-PET1. ∎

3.11 Finite-State Algorithmic Verification: General Response

Algorithm STATE-RESP can be applied for algorithmic verification of a formula

$$\psi: \quad p \;\Rightarrow\; \diamondsuit\, q$$

over a finite-state program P for the case that p and q are assertions. Here, we consider the more general case that p and q are past formulas. Algorithmic verification of general response formulas is based on extending algorithm STATE-RESP to analyze the reachable atoms graph $\mathcal{A}_{(P,\psi)}$ instead of the state-transition graph G_P.

Recall from Section 4.10 of the SAFETY book that $\mathcal{A}_{(P,\psi)}$ is a graph of atoms where each atom $\alpha \in \mathcal{A}$ has the form α: $\langle s; \varphi_1^{b_1}, \ldots, \varphi_n^{b_n} \rangle$ consisting of a state s and assignment of truth-values to each past formula φ_i belonging to the past-closure Φ_ψ of formula ψ.

Example Consider system INC2 presented in Fig. 3.54 and the response property

$$\psi: \quad \underbrace{x = 0}_{p} \;\Rightarrow\; \diamondsuit\Big(\underbrace{even(x) \;\wedge\; \ominus\, odd(x)}_{q}\Big)$$

claiming that any state satisfying $x = 0$ is followed by an even state (a state in which x is even) which is immediately preceded by an odd state.

The past-closure Φ_ψ for this case is

$$\Phi_\psi: \quad \{\ominus\, odd(x)\}$$

since $\ominus\, odd(x)$ is the only subformula of ψ with a principal past operator. Consequently, we construct in Fig. 3.60 the graph $\mathcal{A}_{(P,\psi)}$ of reachable atoms, using the algorithm presented in Section 4.10 of the SAFETY book.

Given an atom $\alpha \in \mathcal{A}_{(P,\psi)}$, we can evaluate formulas p and q on α since α provides an interpretation (T or F) to each subformula of p or q. Consequently, we refer to an atom α as a *p-atom* or a *q-atom* if the evaluation of p or q on α yields T. In analogy with the assertional case, we say that a path $\alpha_a, \ldots, \alpha_b$ in $\mathcal{A}_{(P,\psi)}$ is q-free if no α_i, $i \in [a, b]$ is a q-atom. An atom $\alpha \in \mathcal{A}_{(P,\psi)}$ is called *q-pending* if there exists a q-free atom path $\alpha_a, \ldots, \alpha_b = \alpha$ such that α_a is a p-atom. We define $Q_{\mathcal{A}}$ to be the subgraph of all q-pending atoms in $\mathcal{A}_{(P,\psi)}$. It can be calculated by an algorithm analogous to Q-GRAPH replacing all references to states, p-states, and q-states, by atoms, p-atoms, and q-atoms.

Example In Fig. 3.61 we present $Q_{\mathcal{A}}$ the subgraph of q-pending atoms for system INC2, p: $x = 0$, and q: $even(x) \wedge \ominus\, odd(x)$. ∎

The notions of just, compassionate, and fair subgraphs extend naturally to subgraphs of atoms.

Algorithm GENERAL-RESP generalizes algorithm STATE-RESP to the case that p and q are past formulas.

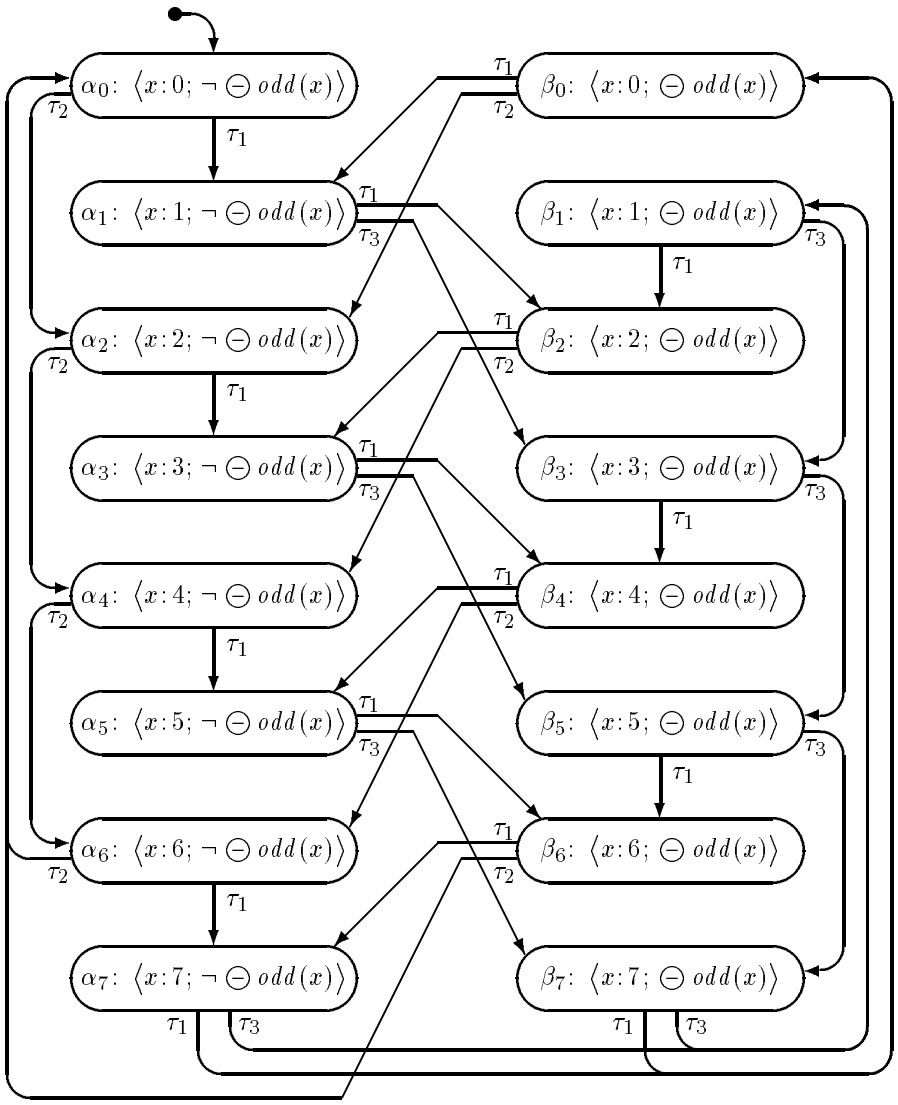


Fig. 3.60. Graph $\mathcal{A}_{(P,\psi)}$ of reachable atoms for INC2.

Algorithm GENERAL-RESP (algorithmic verification of general response formulas)

To check whether formula ψ: $p \Rightarrow \diamondsuit q$, where p and q are past formulas, is valid over a finite-state system P.

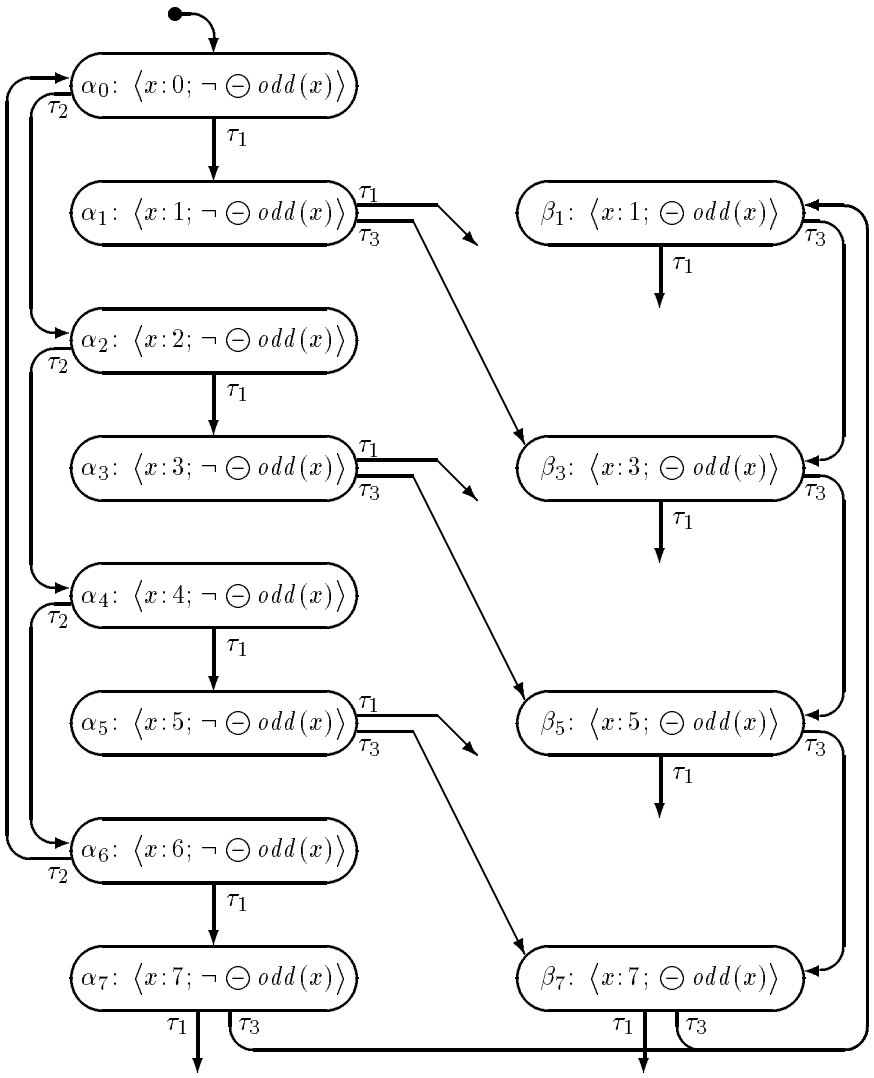


Fig. 3.61. Graph of q-pending atoms for INC2.

- Construct $\mathcal{A}_{(P,\psi)}$ the graph of reachable atoms for P and ψ.
- Calculate $Q_{\mathcal{A}}$ the graph of q-pending atoms.
- Decompose $Q_{\mathcal{A}}$ into MSCS's $A_1, \ldots, A_t$.
- For each $i = 1, \ldots, t$, apply *fair-sub* to A_i.

If any of the applications returns with a nonempty result, then formula ψ is not P-valid. A counter-example can be constructed from the returned fair subgraph.
If all applications return the empty set as a result, then formula ψ is P-valid.

Example Decomposing graph $Q_{\mathcal{A}}$ of Fig. 3.61 into MSCS's, we obtain the following list of subgraphs

$$A_1\colon \{\alpha_0, \alpha_2, \alpha_4, \alpha_6\}, \quad A_2\colon \{\alpha_1\}, \quad A_3\colon \{\alpha_3\}, \quad A_4\colon \{\alpha_5\},$$
$$A_5\colon \{\alpha_7\}, \quad A_6\colon \{\beta_1, \beta_3, \beta_5, \beta_7\}.$$

All of these subgraphs are unjust towards τ_1 since this transition is always enabled but not taken in any of these subgraphs.

We conclude that $Q_{\mathcal{A}}$ contains no fair SCS and, therefore, $x = 0 \Rightarrow \diamondsuit\big(even(x) \wedge \ominus\, odd(x)\big)$ is valid over system INC2. ∎

The justification of Algorithm GENERAL RESP follows precisely the same lines as the sequence of claims justifying its state counterpart, Algorithm STATE-RESP.

Problems

Problem 3.1 (program PROD-CONS without compassion) page 159

Consider program PROD-CONS (Fig. 3.5) under the assumption that the transitions corresponding to the *request* statements are taken to be just, and the compassion set is empty. Show that, in spite of this weakening assumption, the program still satisfies the response property

$$at\_\ell_2 \quad \Rightarrow \quad \diamondsuit\, at\_\ell_4\ .$$

Problem 3.2 (invariant for program ALTER) page 176

Prove the invariant

$$Y[1..j-1] \;=\; X[1..j-1]$$

for program ALTER of Fig. 3.18. In your proof you may use invariants I_1–I_6 presented in the text. Please verify any of the invariants you use.

Problem 3.3 (communal accessibility for program INADEQUATE) page 200

Prove communal accessibility $N_2 > 0 \Rightarrow \diamondsuit(N_3 > 0)$ for program INADEQUATE (Fig. 3.37).

Problem 3.4 (using the completeness construction to verify MUX-SEM) page 224

Use the construction outlined in the completeness proof of Section 3.8 to verify accessibility for program MUX-SEM of Fig. 3.1.

Chapter 2

Response for Parameterized Programs

In this chapter we study methods for proving response properties of parameterized programs.

Parameterized programs allow the variable size (dynamic) statements

$$\|_{j=1}^{M} S[j] \qquad \text{and} \qquad \mathbf{OR}_{j=1}^{M} S[j].$$

The elaboration of such a statement is deferred until execution reaches it. At that point, the variable (or expression) M is evaluated and the program prepares to execute the statement

$$S[1] \,\|\, \ldots \,\|\, S[m] \qquad \text{or} \qquad S[1] \textbf{ or } \ldots \textbf{ or } S[m],$$

respectively, where m is the current value of M. Each parameterized statement $S[i]$, for $i = 1, \ldots, m$, is obtained by substituting the numeral i for the formal parameter j appearing in the text of the program and using it as a subscript in the labels and locally declared variables. Note that the variable M may be an input variable or may even be computed during the execution up to this point. In both cases, it is required that, on arrival to the dynamic statement, M has the value of a positive integer.

We have already considered such parameterized programs when studying methods for verifying their safety properties (Section 2.1 of the SAFETY book). To do so, we introduced several notations: If ℓ_i is a location within the parameterized statement $S[k]$, we write $at\_\ell_i[k]$ to denote that control is currently at location ℓ_i within statement $S[k]$. For a cooperation statement, we write L_i for the set of processes whose control is currently at ℓ_i, i.e.,

$$L_i \;=\; \{k \in [1..M] \mid at\_\ell_i[k]\},$$

and

$$N_i \;=\; |L_i|$$

for the size of this set. These notations are extended to sets and ranges of locations. Thus,

$$\begin{aligned} L_{i_1,i_2,\ldots,i_k} &= L_{i_1} \cup L_{i_2} \cup \cdots \cup L_{i_k} \\ L_{i..j} &= L_i \cup L_{i+1} \cup \cdots \cup L_j \\ N_{i_1,i_2,\ldots,i_k} &= |L_{i_1,i_2,\ldots,i_k}| \\ N_{i..j} &= |L_{i..j}|. \end{aligned}$$

The study of methods for verifying safety properties of parameterized programs in that section led to the conclusion that, with the appropriate notations, the basic rules are applicable and no special extensions are required to deal with parameterized progrmas. We intend to establish a similar conclusion for response properties, while illustrating the utility of the special notations for proving response properties of parameterized programs.

In most of the chapter, we consider the simpler case of response formulas $p \Rightarrow \diamondsuit q$, where p and q are assertions. The generalization necessary to handle the more general case that p and q are arbitrary past formulas is straightforward. Instead of assertions, we use past formulas, and instead of requiring that implications such as $p \rightarrow q$ (for assertions p and q) be P-state valid, we require that the entailment $p \Rightarrow q$, for past formulas p and q be P-valid.

2.1 Parameterized Well-Founded Rule

In most of the cases, response proofs of parameterized programs must be based on a version of rule WELL-J (Fig. 1.26), rather than on the weaker rule CHAIN-J (Fig. 1.7). This is due to the potentially unbounded number of helpful steps, which may depend on the value of the parameter M.

Rule WELL-J assumes that we can find a fixed number m of intermediate assertions $\varphi_0, \ldots, \varphi_m$, with corresponding helpful transitions $\tau_1, \ldots, \tau_m$, and ranking functions $\delta_0, \ldots, \delta_m$. To verify properties of parameterized programs, we can still use a fixed number of intermediate formulas and helpful transitions but they may refer to an additional variable k which is a process index.

To improve readability of formulas, we write $\rho_{\tau_i[k]}$ as $\rho_{\tau_i}[k]$.

Example (program TRIVIAL)

Consider the parameterized program TRIVIAL of Fig. 2.1. This program satisfies the response property

in M: **integer where** $M > 0$

$$\|_{i=1}^{M} P[i] :: \quad [\ell_0\colon \textbf{skip};\ \ell_1\colon \textbf{skip};\ \ell_2\colon]$$

Fig. 2.1. Program TRIVIAL.

$$\underbrace{N_0 > 0}_{p} \quad \Rightarrow \quad \Diamond \underbrace{N_1 > 0}_{q=\varphi_0},$$

which states that, if some process is at ℓ_0, then eventually some process will be at ℓ_1.

A natural intermediate assertion to be used for proving this property is

$$\varphi[k]\colon \quad at\_\ell_0[k].$$

This assertion refers to the process index k such that $P[k]$ is currently at ℓ_0. The transition which is helpful for states satisfying $\varphi[k]$ is

$$\tau[k]\colon \quad \ell_0[k].$$

This shows that the identity of the helpful transition may also depend on the process index parameter k. ∎

In Fig. 2.2 we present rule WELL-JP, a version of rule WELL-J, extended to accommodate parameterized intermediate formulas and corresponding parameterized helpful transitions. For each $i = 1, \ldots, m$, the parameter k in $\varphi_i[k]$ and $\tau_i[k]$ is assumed to range over some nonempty set R_i whose size may depend on the inputs to the program. For $i \neq j$, the ranges R_i and R_j may be different. For each $j = 1, \ldots, m$, we use the notation $\widehat{\varphi}_j$ as an abbreviation for

$$\widehat{\varphi}_j\colon \quad \exists u \in R_j\colon\ \varphi_j[u]$$

indicating that $\widehat{\varphi}_j$ holds at a state if $\varphi_j[u]$ holds there, for some $u \in R_j$.

We refer to $\widehat{\varphi}_j$ as the *summary* of the parameterized formula $\varphi_j[k]$.

As usual, it is sufficient to check premise JP2 for $\tau \neq \tau_i[k]$.

Example (program TRIVIAL)

We show how to prove the response formula

$$\underbrace{N_0 > 0}_{p} \quad \Rightarrow \quad \Diamond \underbrace{N_1 > 0}_{q}$$

For assertions p, and $q = \varphi_0, \varphi_1[k], \ldots, \varphi_m[k]$,
transitions $\tau_1[k], \ldots, \tau_m[k] \in \mathcal{J}$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_0, \ldots, \delta_m$: $\Sigma \mapsto \mathcal{A}$

JP1. $p \rightarrow \bigvee_{j=0}^{m} \widehat{\varphi}_j$

JP2. $\rho_\tau \wedge \varphi_i[k] \rightarrow \left[\begin{array}{l} \bigvee_{j=0}^{m} (\widehat{\varphi}'_j \wedge \delta_i \succ \delta'_j) \\ \vee (\varphi'_i[k] \wedge \delta_i = \delta'_i) \end{array}\right]$ for every $\tau \in \mathcal{T}$

JP3. $\rho_{\tau_i[k]} \wedge \varphi_i[k] \rightarrow \bigvee_{j=0}^{m} (\widehat{\varphi}'_j[r] \wedge \delta_i \succ \delta'_j)$

JP4. $\varphi_i[k] \rightarrow En(\tau_i[k])$

} for $i = 1, \ldots, m$, and $k \in R_i$

$$p \Rightarrow \Diamond q$$

Fig. 2.2. Rule WELL-JP (parameterized well-founded response under justice).

for program TRIVIAL (Fig. 2.1), using rule WELL-JP. Take $m = 1$ and the following parameterized intermediate assertions, helpful transition, and ranking functions:

$$\begin{array}{rlll} & \varphi_1[k]\colon\ at\_\ell_0[k], & \tau_1[k]\colon\ \ell_0[k], & \delta_1\colon\ 1 \\ q = \varphi_0[k]\colon\ N_1 > 0, & & & \delta_0\colon\ 0, \end{array}$$

where the parameter k ranges over R: $[1..M]$. Note that

$$\widehat{\varphi}_1 \ = \ \exists u \in [1..M]\colon\ at\_\ell_0[u] \ = \ N_0 > 0.$$

Let us check that all premises of rule WELL-JP are satisfied by this choice. Since $m = 1$ all multiple disjunctions over j collapse to a single disjunct.

- Premise JP1 requires

$$\underbrace{N_0 > 0}_{p} \ \rightarrow \ \cdots \vee \underbrace{N_0 > 0}_{\widehat{\varphi}_1},$$

which is valid.

- For premise JP2, it suffices to show

$$\rho_\tau \;\wedge\; \underbrace{at\_\ell_0[k]}_{\varphi[k]} \;\rightarrow\; \cdots \vee \cdots \vee \underbrace{at'\_\ell_0[k]}_{\varphi'[k] \,\wedge\, \delta=\delta'},$$

  for every $\tau \neq \ell_0[k]$. Clearly, any such transition preserves the truth of $at\_\ell_0[k]$.

- Premise JP3 requires

$$\rho_{\ell_0}[k] \;\wedge\; \underbrace{at\_\ell_0[k]}_{\varphi[k]} \;\rightarrow\; \underbrace{N_1' > 0}_{q'} \;\vee\; \cdots,$$

  which is obviously valid.

- Premise JP4 requires the obviously valid implication

$$\underbrace{at\_\ell_0[k]}_{\varphi[k]} \;\rightarrow\; \underbrace{at\_\ell_0[k]}_{En(\tau[k])}.$$

This established the validity of $N_0 > 0 \Rightarrow \diamondsuit(N_1 > 0)$ over program TRIVIAL. ∎

A Useful Well-Founded Domain

Progress in parameterized programs can often be measured by the number of processes that satisfy some requirements. For example, the response property

$$N_0 > 0 \;\Rightarrow\; \diamondsuit(N_0 = 0)$$

states that, starting from a state at which some processes are at location ℓ_0, eventually we will get to a state at which no process is executing at ℓ_0.

The most natural progress measure for such cases is the set of indices of the processes that currently satisfy the requirement of interest.

For example, we can respecify eventual evacuation of location ℓ_0 by the formula

$$L_0 \neq \emptyset \;\Rightarrow\; \diamondsuit(L_0 = \emptyset),$$

and use L_0 as the ranking function.

It is straightforward to show that the domain of subsets of $\{1, \ldots, M\}$ with the ordering relation $\succ$ taken as set inclusion, i.e., the domain $(2^{[1..M]}, \supset)$ is well founded.

While proving verification conditions involving a ranking function of the form L_i, we have to establish inclusions of the form $L_i \supseteq L_i'$ and $L_i \supset L_i'$.

To prove $L_i \supseteq L_i'$, we will establish

$$\forall j \in [1..M]\colon\; at'\_\ell_i[j] \;\rightarrow\; at\_\ell_i[j],$$

which guarantee that any index j belongs to L_i' only if it also belongs to L_i.

To prove $L_i \supset L_i'$, we will establish

$$\big(\forall j \in [1..M]\colon\ at'\_\ell_i[j] \ \rightarrow\ at\_\ell_i[j]\big) \wedge \big(\exists j \in [1..M]\colon\ at\_\ell_i[j] \wedge \neg at'\_\ell_i[j]\big)\,.$$

The first clause of this conjunction guarantees that $L_i \supseteq L_i'$, while the second clause requires the identification of, at least, one process which has just left L_i.

Example (evacuation of ℓ_0 in program TRIVIAL)

We prove the response formula

$$\underbrace{L_0 \neq \emptyset}_{p} \ \Rightarrow\ \diamondsuit \underbrace{L_0 = \emptyset}_{q}\,,$$

specifying eventual evacuation of location ℓ_0 for program TRIVIAL.

We take $m = 1$, and choose as follows:

$$\begin{array}{lll} \varphi_1[k]\colon\ at\_\ell_0[k], & \tau_1[k]\colon\ \ell_0[k], & \delta_1\colon\ L_0 \\ \varphi_0[k]\colon\ L_0 = \emptyset, & & \delta_0\colon\ \emptyset. \end{array}$$

Let us check that all premises of rule WELL-JP are satisfied by this choice.

- To establish JP1, it is sufficient to show

$$\underbrace{L_0 \neq \emptyset}_{p} \ \rightarrow\ \cdots \vee \underbrace{\exists j \in [1..M]\colon\ at\_\ell_0[j]}_{\widehat{\varphi}_1}$$

  which is valid.

- To establish JP2, it is sufficient to show

$$\rho_\tau \ \wedge\ \underbrace{at\_\ell_0[k]}_{\varphi_1[k]} \ \rightarrow\ \underbrace{at'\_\ell_0[k] \ \wedge\ L_0 \supseteq L_0'}_{\varphi_1'[k]\ \wedge\ \delta_1 \succcurlyeq \delta_1'}\,,$$

  for every $\tau \neq \ell_0[k]$. There are two cases to be considered: $\tau = \ell_0[i]$ for $i \neq k$, and $\tau = \ell_1[i]$. It is clear that none of these transitions can invalidate $at\_\ell_0[k]$, or cause a new process to join L_0.

- For premise JP3, it is sufficient to show

$$\rho_{\ell_0}[k] \ \wedge\ \underbrace{at\_\ell_0[k]}_{\varphi_1[k]} \ \rightarrow\ (L_0 \supset L_0') \ \wedge\ \big(\underbrace{L_0' = \emptyset}_{\varphi_0'} \ \wedge\ \underbrace{\exists j\colon\ at'\_\ell_0[j]}_{\widehat{\varphi}_1'}\big)\,.$$

  The strict inclusion $L_0 \supset L_0'$ follows from the fact that transition $\ell_0[k]$ causes $P[k]$ to move away from ℓ_0 to ℓ_1, and does not change the location of any other process.

- Premise JP4,

$$\underbrace{at\_\ell_0[k]}_{\varphi_1[k]} \ \rightarrow\ \underbrace{at\_\ell_0[k]}_{En\left(\tau_1[k]\right)}$$

is obviously valid. ∎

Example (program MAX-ARRAY)

As a more advanced example, consider program MAX-ARRAY for parallel computation of the maximum of an integer array. This program was first presented in Fig. 2.5 of the SAFETY book and is reproduced here in Fig. 2.3.

$$
\begin{array}{ll}
\textbf{in} & M: \textbf{integer where } M \geq 1 \\
 & x : \textbf{array } [1..M] \textbf{ of integer} \\
\textbf{local} & y : \textbf{array } [1..M] \textbf{ of boolean where } y = \mathrm{T} \\
\textbf{out} & z : \textbf{integer}
\end{array}
$$

$$
\ell_0 \colon \Big\|_{i,j \in [1..M]} P[i,j] :: \left[\begin{array}{l} \ell_1 \colon \textbf{if } x[i] < x[j] \textbf{ then } \ell_2 \colon y[i] := \mathrm{F} \\ \ell_3 \colon \end{array}\right]
$$

$$
\ell_4 \colon \Big\|_{i=1}^{M} Q[i] :: \left[\begin{array}{l} \ell_5 \colon \textbf{if } y[i] \textbf{ then } \ell_6 \colon z := x[i] \\ \ell_7 \colon \end{array}\right]
$$

$$
\ell_8 \colon
$$

Fig. 2.3. Program MAX-ARRAY (finding the maximum of an array).

In Section 2.2 of the SAFETY book, dealing with safety, we proved the partial correctness of this program, expressed by

$$at\_\ell_8 \quad \Rightarrow \quad maximal(z,\ x).$$

To complete the verification of this program, we have to prove its termination, expressed by response property

$$\underbrace{at\_\ell_0}_{p} \quad \Rightarrow \quad \diamondsuit \underbrace{at\_\ell_8}_{q} \ .$$

Intuitively, termination is obvious, since each process has a straight line program with no loops. To prove it formally, using rule WELL-JP, we identify a well-founded domain

$$(\mathcal{A}, \succ): \quad \big([1..8],\ >\big) \ \times \ \big(2^{[1..M]},\ \supset\big).$$

Note that $(r_1, L_1) \succ (r_2, L_2)$ for $r_1, r_2 \in \{1, \ldots, 8\}$ and $L_1, L_2 \subseteq [1..M]$ if

$$r_1 > r_2 \qquad \text{or} \qquad (r_1 = r_2) \ \wedge \ (L_1 \supset L_2).$$

The parameterized intermediate assertions and helpful transitions, and the ranking functions are given in the table of Fig. 2.4. To compare a ranking function

such as δ_6: $(6, L_2)$ with a shorter rank such as δ_4: 5, we pad the shorter tuple on the right by the empty set $\emptyset$. Thus, we are actually comparing $(6, L_2)$ with $(5, \emptyset)$.

φ_8	: at_ℓ_0	τ_8	: ℓ_0	δ_8:	8
$\varphi_7[i,j]$:	$at\_\ell_1[i,j] \wedge N_{1..3} = M^2$	$\tau_7[i,j]$:	$\ell_1[i,j]$	δ_7:	$(7, L_1)$
$\varphi_6[i,j]$:	$at\_\ell_2[i,j] \wedge N_{2,3} = M^2$	$\tau_6[i,j]$:	$\ell_2[i,j]$	δ_6:	$(6, L_2)$
φ_5	: $N_3 = M^2$	τ_5	: ℓ_3	δ_5:	5
φ_4	: at_ℓ_4	τ_4	: ℓ_4	δ_4:	4
$\varphi_3[i]$	: $at\_\ell_5[i] \wedge N_{5..7} = M$	$\tau_3[i]$	: $\ell_5[i]$	δ_3:	$(3, L_5)$
$\varphi_2[i]$	: $at\_\ell_6[i] \wedge N_{6,7} = M$	$\tau_2[i]$	: $\ell_6[i]$	δ_2:	$(2, L_6)$
φ_1	: $N_7 = M$	τ_1	: ℓ_7	δ_1:	1
$q = \varphi_0$:	at_ℓ_8			δ_0:	0

Fig. 2.4. Assertions, transitions and rankings for program MAX-ARRAY.

Note that we refer to the exit transitions from the cooperation statements ℓ_0 and ℓ_4 by the names ℓ_3 and ℓ_7, respectively. These are *single* transitions that are enabled only when all processes within the cooperation statement have reached their terminal locations ℓ_3 and ℓ_7, respectively.

The summaries of the parameterized assertions are given by

$$\widehat{\varphi}_7\colon \quad N_1 > 0 \ \wedge \ N_{1..3} = M^2 \qquad \widehat{\varphi}_6\colon \quad N_2 > 0 \ \wedge \ N_{2,3} = M^2$$
$$\widehat{\varphi}_3\colon \quad N_5 > 0 \ \wedge \ N_{5..7} = M \qquad \widehat{\varphi}_2\colon \quad N_6 > 0 \ \wedge \ N_{6,7} = M.$$

As we see, the table refers to eight parameterized assertions and helpful transitions. Assertions φ_6, φ_7, and their corresponding transitions are parameterized by $i, j \in [1..M]$, while assertions φ_2, φ_3, and their corresponding transitions are parameterized by $i \in [1..M]$. Thus, assertions φ_6 and φ_7 can be viewed as parameterized by $k = \langle i, j \rangle$, ranging over $R_6 = R_7$: $[1..M] \times [1..M]$, assertions φ_2 and φ_3 are parameterized by $k = i$, ranging over $R_2 = R_3$: $[1..M]$, and assertions φ_1, φ_4, φ_5, and φ_6 can be viewed as trivially parameterized by k ranging over $R_1 = R_4 = R_5 = R_6$: $[1]$. Note that, for $i \in \{1, 4, 5, 6\}$, $\widehat{\varphi}_i = \varphi_i$.

The proof uses the following invariant

$$\psi\colon \quad \begin{cases} N_{0,4,8} = 1 & \wedge \ N_{1..3,5..7} = 0 & \vee \\ N_{1..3} = M^2 & \wedge \ N_{0,4..8} = 0 & \vee \\ N_{5..7} = M & \wedge \ N_{0..4,8} = 0. \end{cases}$$

It is not difficult to check the premises of rule WELL-JP and verify that they are all satisfied.

For example, let us check premise JP3 for assertion $\varphi_2[i]$. We show that the following implication is state valid

$$\rho_{\ell_6}[i] \ \wedge \ \underbrace{at\_\ell_6[i] \ \wedge \ N_{6,7} = M}_{\varphi_2[i]} \ \rightarrow \ \cdots \ \vee \ \underbrace{N_7' = M \ \wedge \ (2, L_6) \succ (1, 0)}_{\widehat{\varphi}_1' \ \wedge \ \delta_2 \succ \delta_1'} \ \vee$$
$$\underbrace{N_6' > 0 \ \wedge \ N_{6,7}' = M \ \wedge \ (2, L_6) \succ (2, L_6')}_{\widehat{\varphi}_2' \ \wedge \ \delta_2 \succ \delta_2'}.$$

Assertion $\varphi_2[i]$: $at\_\ell_6[i] \wedge N_{6,7} = M$ implies $N_7 \leq M - 1$. To show the validity of the implication, we consider two cases. If $N_7 = M - 1$, then transition $\ell_6[i]$ leads to a successor state in which $N_7 = M$, establishing φ_1'. If $N_7 < M - 1$, there exists another index $k \neq i$ such that $at\_\ell_6[k]$ holds at the state before the transition and therefore also at the successor state. This establishes $\widehat{\varphi}_2'$ and a rank decrease at the successor state. ▟

2.2 Representation by Diagrams

To represent a proof of parameterized program by verification diagrams using rule WELL-JP, we follow the same conventions as for RANK diagrams (see Section 1.4), but introduce a special notation for parameterized nodes and edges.

A parameterized node is identified by a parameterized assertion of the form

$$\varphi_j\colon \quad \lambda k \in R_j\colon \ f[k],$$

where R_j is the range of the parameter k, which may be different for different j's, and f is an assertion that may refer to the parameter k.

When there is no danger of ambiguity (and the range R_j is understood from the context), we may also represent the parameterized assertion as

$$\varphi_j[k]\colon \quad f[k].$$

To make the treatment uniform, we view nodes labeled by an unparameterized assertion φ_u as though they were labeled by the trivially parameterized assertion

$$\varphi_u\colon \quad \lambda k \in \{1\}\colon \ f_u[k],$$

where $f_u[k] = \varphi_u$.

Edges departing from a node parameterized by k are of the following forms:

- A double-line edge labeled by $\tau[k]$.
- A single-line edge labeled by a *set-expression* T_r which stands for the set of transition instances $\{\tau_r[i] \mid i \in R_r\}$, where R_r is the range of the parameterized transition $\tau_r[i]$.

Example (program TRIVIAL)

In Fig. 2.5, we present a P-RANK diagram for program TRIVIAL of Fig. 2.1.

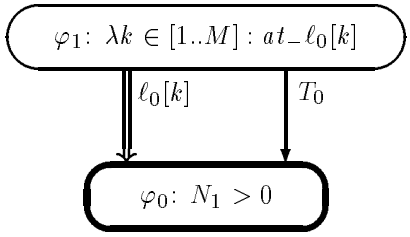


Fig. 2.5. P-RANK diagram for program TRIVIAL.

Node φ_1 in this diagram is parameterized by $k \in [1..M]$, while node φ_0 is unparameterized.

Assertion φ_0 can be trivially represented as parameterized by writing

$$\varphi_0: \quad \lambda k \in \{1\}: \; N_1 > 0. \; \blacksquare$$

Verification and Enabling Conditions for P-RANK Diagrams

Consider a non-terminal node labeled by the assertion $\varphi[k]$ and the ranking δ. Let L be a label of some edge departing from the node φ.

Let $\varphi_1, \ldots, \varphi_n$, $n > 0$, be the successors of φ by L-labeled edges and let $\delta_1, \ldots, \delta_n$ be the ranking functions of these successors.

- If $L = \tau[k]$ (labeling double edges), then the following verification conditions are implied:

$$\rho_\tau[k] \; \wedge \; \varphi[k] \;\; \rightarrow \;\; \left((\widehat{\varphi}'_1 \; \wedge \; \delta \; \succ \; \delta'_1) \; \vee \; \cdots \; \vee \; (\widehat{\varphi}'_n \; \wedge \; \delta \; \succ \; \delta'_n)\right)$$

$$\varphi[k] \;\; \rightarrow \;\; En\left(\tau[k]\right).$$

- If $L = T_r$ (labeling single edges), then the following verification condition is implied:

$$\rho_{\tau_r}[i] \wedge \varphi[k] \;\; \rightarrow \;\; \left((\varphi'[k] \wedge \delta \succcurlyeq \delta') \vee (\widehat{\varphi}'_1 \wedge \delta \; \succ \; \delta'_1) \vee \cdots \vee (\widehat{\varphi}'_n \wedge \delta \; \succ \; \delta'_n)\right).$$

In addition, Let τ_r be any transition not appearing in the label of any edge departing from φ. Then, the following verification condition is implied:

- $\varphi_{\tau_r}[i] \ \wedge \ \varphi[k] \ \rightarrow \ \varphi'[k] \ \wedge \ \delta \succcurlyeq \delta'$.

This corresponds to transitions labeling $n = 0$ single edges.

Example (program TRIVIAL)

The P-RANK diagram of Fig. 2.5 implies the following verification conditions:

$$\rho_{\ell_0}[k] \ \wedge \ at\_\ell_0[k] \ \rightarrow \ N_1' > 0 \ \wedge \ \underbrace{1}_{\delta_1} > \underbrace{0}_{\delta_0'}$$

$$at\_\ell_0[k] \ \rightarrow \ \underbrace{at\_\ell_0[k]}_{En(\ell_0[k])}$$

$$\rho_{\ell_0}[i] \ \wedge \ at\_\ell_0[k] \ \rightarrow \ \big(at\_\ell_0'[k] \ \wedge \ 1 \ \geq \ 1) \ \vee \ (N_1' \ > \ 0 \ \wedge \ 1 \ > \ 0)\big)$$

$$\rho_{\ell_1}[i] \ \wedge \ at\_\ell_0[k] \ \rightarrow \ at\_\ell_0'[k] \ \wedge \ 1 \ \geq \ 1.$$

The last verification condition corresponds to transition $\ell_1[i]$ which labels no edge departing from φ_1.

In Fig. 2.6, we present a P-RANK diagram for the evacuation property

$$L_0 \neq \emptyset \ \Rightarrow \ \Diamond(L_0 = \emptyset)$$

for program TRIVIAL.

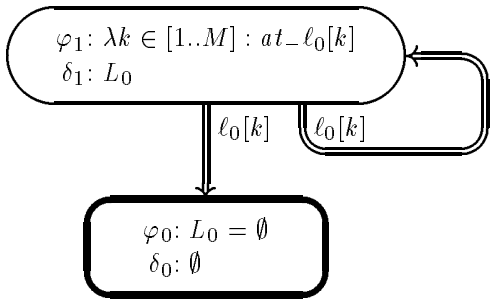


Fig. 2.6. P-RANK diagram for eventual evacuation of ℓ_0.

Note that while $P[k]$ is at ℓ_0, no other transition $\ell_0[i]$, for $i \neq k$, can cause L_0 to become empty. This is why we do not need a single edge labeled by T_0 connecting node φ_1 to node φ_0. ⌟

Valid P-RANK Diagrams

A P-RANK diagram is said to be *valid over program P* (*P-valid* for short) if all the verification conditions associated with the diagram are P-state valid.

The consequences of having a valid P-RANK diagram are stated in the following claim:

Claim 2.1 (P-RANK diagrams)

A P-valid P-RANK diagram establishes that the response formula

$$\bigvee_{j=0}^{m} \Rightarrow \Diamond \widehat{\varphi}_0$$

is P-valid.

If, in addition, we can establish the P-state validity of the implications

$$p \rightarrow \bigvee_{j=0}^{m} \widehat{\varphi}_j \qquad \text{and} \qquad \widehat{\varphi}_0 \rightarrow q$$

then, we can conclude the P-validity of

$$p \Rightarrow \Diamond q.$$

Justification The first part of the claim follows from the soundness rule WELL-JP, and the observation that the verification conditions induced by a P-RANK diagram imply premises JP2–JP4 of rule WELL-JP. Given the two additional implications, we can also infer premise JP1 and the replacement of $\widehat{\varphi}_0$ by q. ∎

Example (program MAX-ARRAY)

We return to the example of program MAX-ARRAY (Fig. 2.3), for which we wish to prove termination, specified by

$$\underbrace{at\_\ell_0}_{p} \Rightarrow \Diamond \underbrace{at\_\ell_8}_{q} .$$

In Fig. 2.7 we present a P-RANK diagram for this property.

In this diagram, we use encapsulation conventions to factor out the prefix $\lambda_{i,j} \in [1..M]$ from φ_6 and φ_7 and write it as a label of the supernode containing these nodes. A similar factoring was applied to nodes φ_2 and φ_3.

When we compute the default verification conditions corresponding to transitions that do not label departing edges, it is important to consider these transitions only for the values of the parameter which do not appear in labels of departing edges.

Consider, for example, the verification conditions for the parameterized assertion φ_6: $\lambda i, j \in [1..M]$: $at\_\ell_2[i,j] \vee N_{2,3} = M^2$. The verification condition corresponding to the double edge departing from φ_6 is:

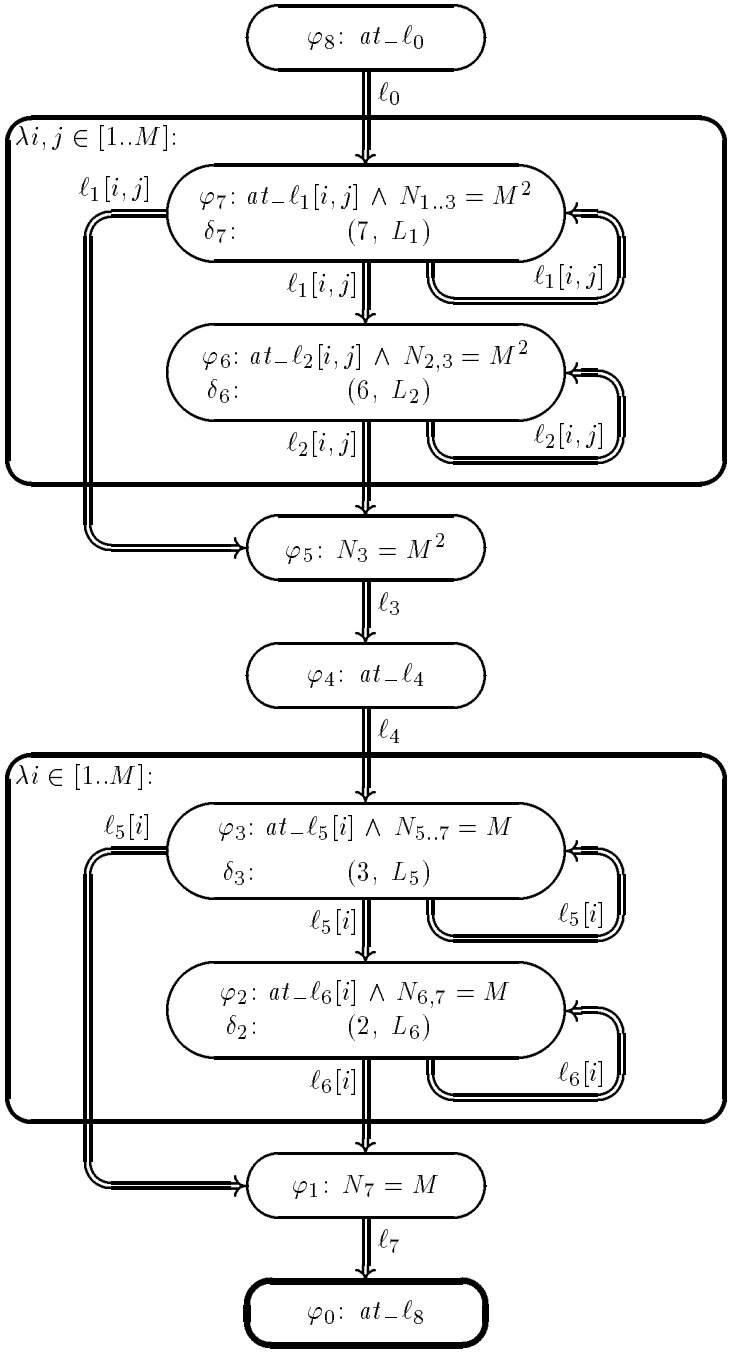


Fig. 2.7. P-RANK diagram by rule WELL-JH for program MAX-ARRAY.

$$\rho_{\ell_2}[i,j] \ \wedge \ \underbrace{at\_\ell_2[i,j] \ \wedge \ N_{2,3} = M^2}_{\varphi_6} \ \rightarrow$$

$$\big(at'\_\ell_2[i,j] \ \wedge \ N'_{2,3} = M^2 \ \wedge \ (6,L_2) \ \succ \ (6,L'_2)\big) \ \vee \ \big(N'_3 = M^2 \ \wedge \ (6,L_2) \ \succ \ 5\big).$$

Among the default verification conditions, we should include

$$(k,r) \neq (i,j) \ \wedge \ \rho_{\ell_2}[k,r] \ \wedge \ at\_\ell_2[i,j] \ \wedge \ N_{2,3} = M^2 \ \rightarrow$$
$$at'\_\ell_2[i,j] \ \wedge \ N'_{2,3} = M^2 \ \wedge \ (6,L_2) \ \succ \ (6,L'_2),$$

as asll we

$$\rho_{\ell_1}[k,r] \ \wedge \ at\_\ell_2[i,j] \ \wedge \ N_{2,3} = M^2 \ \rightarrow$$
$$at'\_\ell_2[i,j] \ \wedge \ N'_{2,3} = M^2 \ \wedge \ (6,L_2) \ \succ \ (6,L'_2).$$

Note the difference between the condition corresponding to $\ell_2[k,r]$ and the one corresponding to $\ell_1[k,r]$. In the case of ℓ_2, we have to exclude the parameter value $(k,r) = (i,j)$, since $\ell_2[i,j]$ already labels an existing edge departing from $\varphi[i,j]$. To do so, we add the conjunct $(k,r) \neq (i,j)$ to the antecedent of the implication. In the case of ℓ_1, which labels no edge departing from $\varphi_7[i,j]$, no such restriction is necessary.

Example (mutual exclusion by turn setting)

Consider parameterized program TURN of Fig. 2.8 which manages mutual exclusion between an arbitrary number of processes by turn setting. The program uses a shared variable t, which is initially 0. Each process $P[i]$, interested in entering its critical section, loops in $\ell_{2..5}$ trying to change t to the identity number of $P[i]$, namely i. The grouped statement at ℓ_3 is such that it sets t to i only if it finds $t = 0$. If $P[i]$ finds $t \neq 0$ it does not modify t but still moves to ℓ_4. At ℓ_4 the process checks whether its execution of ℓ_3 succeeded in setting t to i. Finding a value $t = i$ at ℓ_4, $P[i]$ enters its critical section ℓ_5. Otherwise it returns to ℓ_2 and then to ℓ_3 to try again.

Mutual exclusion is ensured by the invariant

$$\chi: \quad i \in L_{5,6} \ \rightarrow \ t = i.$$

We wish to show a weak-accessibility property, formulated by

$$\psi: \quad \underbrace{N_2 > 0 \ \wedge \ t = 0}_{p} \ \Rightarrow \ \Diamond \underbrace{N_5 > 0}_{q}.$$

This formula states that if some process is at ℓ_2 with $t = 0$, then eventually some process will get to ℓ_5. We refer to it as *weak accessibility* (sometimes also as *communal accessibility*), since it does not guarantee that the same process detected at ℓ_2 with $t = 0$ will be the one to enter ℓ_5. The stronger property

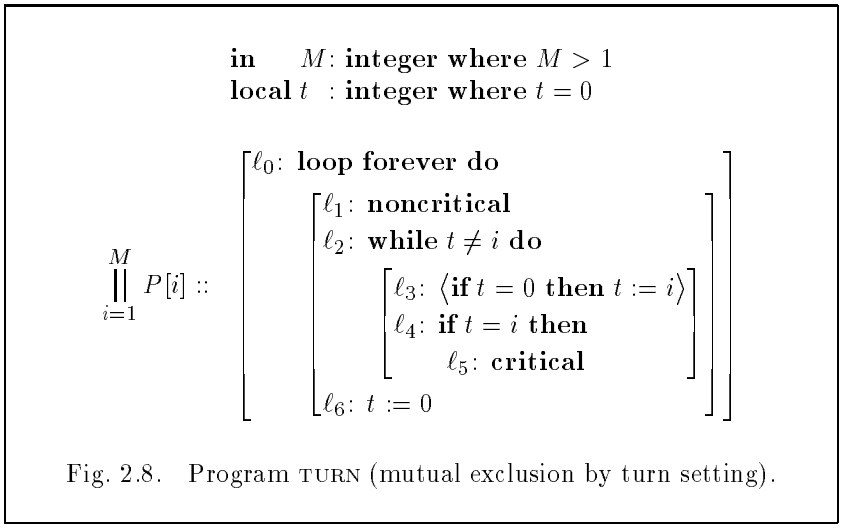


Fig. 2.8. Program TURN (mutual exclusion by turn setting).

of accessibility, claiming that any individual process visiting ℓ_2 with $t = 0$, will eventually reach ℓ_5, which can be expressed by the formula

$$at\_\ell_2[i] \ \wedge \ t = 0 \quad \Rightarrow \quad \diamondsuit \, at\_\ell_5[i],$$

is not valid for this program. In **Problem 2.1**, the reader is requested to show that accessibility is not guaranteed for program TURN.

To verify the weak-accessibility property expressed by ψ, we use the following in rule WELL-JP

$\varphi_3[k]$:	$at\_\ell_2[k] \ \wedge \ N_3 = 0 \ \wedge \ t = 0$	δ_3:	3	$\tau_3[k]$:	$\ell_2[k]$
$\varphi_2[k]$:	$at\_\ell_3[k] \ \wedge \ t = 0$	δ_2:	2	$\tau_2[k]$:	$\ell_3[k]$
φ_1:	$at\_\ell_4[t] \ \wedge \ t \neq 0$	δ_1:	1	τ_1:	$\ell_4[t]$
$q = \varphi_0$:	$N_5 > 0$	δ_0:	0.		

We proceed to check the premises of rule WELL-JP.

- Premise JP1 requires showing $p \rightarrow \cdots \vee \widehat{\varphi}_3$. Since $N_2 > 0$ implies the assertion $\widehat{\varphi}_3$: $\exists k: at\_\ell_2[k]$, the premise follows.
- Premise JP2 will be checked for each $i = 1, 2, 3$.
  - $\varphi_3[k]$: $at\_\ell_2[k] \wedge \ N_3 = 0 \ \wedge \ t = 0$

  Most of the transitions satisfy JP2 for φ_3 by preserving φ_3 and δ_3. The only transitions that may falsify φ_3 are transitions of the form $\ell_2[i]$. Transition $\ell_2[i]$ leads to a state satisfying $\widehat{\varphi}_2$: $\exists r: at\_\ell_2[r] \ \wedge \ t = 0$.
  - $\varphi_2[k]$: $at\_\ell_3[k] \ \wedge \ t = 0$

The only transitions that can falsify φ_2 are of the form $\ell_3[i]$. However, such a transition leads to a state satisfying $\varphi_1\colon\ at\_\ell_4[t] \wedge t \neq 0$.

- $\varphi_1\colon\ at\_\ell_4[t]\ \wedge\ t \neq 0$

The only transition that can falsify φ_1 is $\ell_4[t]$ which leads to a state satisfying the goal assertion $q\colon\ N_5 > 0$.

- Premise JP3 will be checked for each $i = 1, 2, 3$.
  - $\varphi_3[k]\colon\ at\_\ell_2[k]\ \wedge\ N_3 = 0\ \wedge\ t = 0$

  Here we show

  $$\rho_{\ell_2}[k]\ \wedge\ \underbrace{\cdots\ \wedge\ t = 0}_{\varphi_3}\ \rightarrow\ \underbrace{\exists r\colon\ at'\_\ell_3[r]\ \wedge\ t' = 0}_{\widehat{\varphi}'_2}\ \wedge\ 3 > 2.$$

  Since $\rho_{\ell_2}[k]$ implies $at'\_\ell_3[k]$ and $t' = t$, the implication is valid.

  - $\varphi_2[k]\colon\ at\_\ell_3[k]\ \wedge\ t = 0$

  Here we show

  $$\rho_{\ell_3}[k]\ \wedge\ \underbrace{N_3 > 0\ \wedge\ t = 0}_{\varphi_2}\ \rightarrow\ \underbrace{at'\_\ell_4[t]\ \wedge\ t' \neq 0}_{\widehat{\varphi}'_1}\ \wedge\ 2 > 1.$$

  Since $\rho_{\ell_3}[k]$ under $t = 0$ implies $at'\_\ell_4[k]$ and $t' = k \neq 0$, the right-hand side follows.

  - $\varphi_1\colon\ at\_\ell_4[t]\ \wedge\ t \neq 0$

  We will show

  $$\rho_{\ell_4}[t]\ \wedge\ \underbrace{at\_\ell_4[t]\ \wedge\ t \neq 0}_{\varphi_1}\ \rightarrow\ \underbrace{N'_5 > 0}_{q'}\ .$$

  Since $\rho_{\ell_4}[t]$ and $t \neq 0$ imply $at'\_\ell_5[i]$, the right-hand side follows.

- Premise JP4 is obviously satisfied, since $\varphi_i[k]$ implies $En\left(\tau_i[k]\right)$ for $i = 1, 2, 3$.

We can summarize this WELL-JP proof in the P-RANK diagram presented in Fig. 2.9.

■

Program RACE

Consider program RACE, presented in Fig. 2.10. This program consists of M competing processes. The first process $P[k_1]$ to perform statement ℓ_1 will set its local variable $t[k_1]$ to 1, while incrementing the shared variable y to 2. This process eventually gets back to location ℓ_0 and, on finding $t[k_1] = 1$ will proceed to ℓ_2. All processes performing statement ℓ_1 later than $P[k_1]$ will obtain values of $t[k]$ greater than 1, and cannot exit the while loop at ℓ_0.

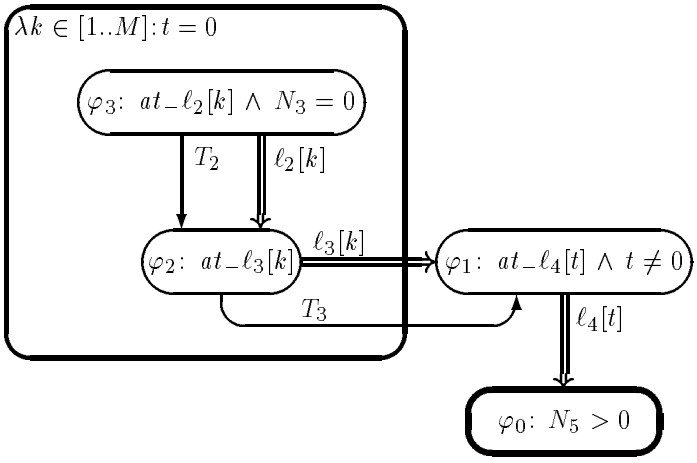


Fig. 2.9. P-RANK diagram for program TURN.

in M: **integer where** $M > 0$
local y : **integer where** $y = 1$

$$\prod_{k=1}^{M} P[k] :: \begin{bmatrix} \textbf{local}\, t\text{: integer where } t = 0 \\ \ell_0\text{: } \textbf{while}\, t \neq 1\, \textbf{do} \\ \qquad \ell_1\text{: } (t,\ y) := (y,\ y+1) \\ \ell_2\text{:} \end{bmatrix}$$

Fig. 2.10. Program RACE.

The property we wish to establish for this program is that eventually some process reaches ℓ_2. This property is stated by the following response formula.

$$\underbrace{N_0 > 0 \ \wedge \ y = 1}_{p} \ \Rightarrow \ \diamondsuit \underbrace{N_2 > 0}_{q} \ .$$

It is easy to trace the progress of the computation towards the goal $N_2 > 0$ through the stages

$$\widehat{\varphi}_3\text{:} \quad N_0 > 0 \ \wedge \ y = 1$$

(initial) and

$$\widehat{\varphi}_2\text{:} \quad N_1 > 0 \ \wedge \ y = 1,$$

up to

$$\varphi_1[k]: \quad at\_\ell_0[k] \ \wedge \ t[k] = 1.$$

The next helpful step takes transition $\ell_0[k]$ and reaches

$$q = \varphi_0: \quad N_2 > 0.$$

Consequently, we choose assertions, helpful transitions, and ranking functions as follows:

$\varphi_3[k]$:	$at\_\ell_0[k] \ \wedge \ y = 1$	$\tau_3[k]$: $\ell_0[k]$	δ_2: 3
$\varphi_2[k]$:	$at\_\ell_1[k] \ \wedge \ y = 1$	$\tau_2[k]$: $\ell_1[k]$	δ_2: 2
$\varphi_1[k]$:	$at\_\ell_0[k] \ \wedge \ t[k] = 1$	$\tau_1[k]$: $\ell_0[k]$	δ_1: 1
φ_0:	$N_0 > 0$.		

In Fig. 2.11, we present a P-RANK diagram representing this proof.

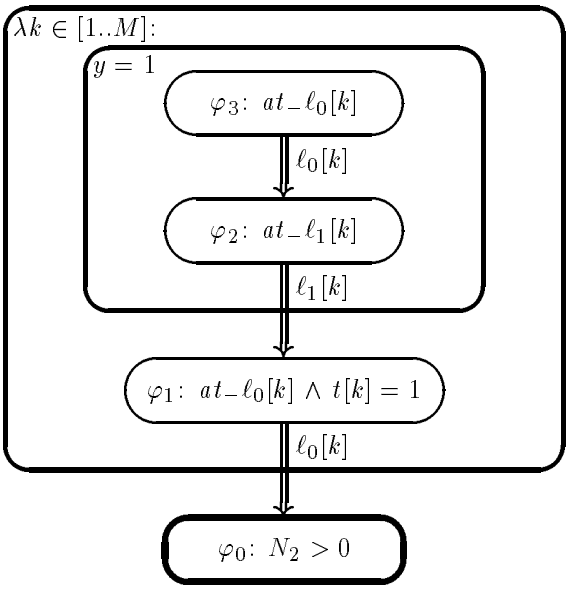


Fig. 2.11. P-RANK diagram for program RACE.

Advanced Example: Resource-Allocator Program

In some cases, response properties of a parameterized program can be proven without the need for explicit parameterization of the assertions and helpful transitions. Thus, rules CHAIN-J (Fig. 1.7) and WELL-J (Fig. 1.26) can be used directly.

We illustrate this on the example of a resource allocator program.

We reconsider program RES-SV for the allocation of a resource between several customers (the shared-variables version). This program was first presented in Section 2.2 of the SAFETY book (Fig. 2.9), and is reproduced here in Fig. 2.12. We denote $t \oplus_M 1 = (t \ mod \ M) + 1$.

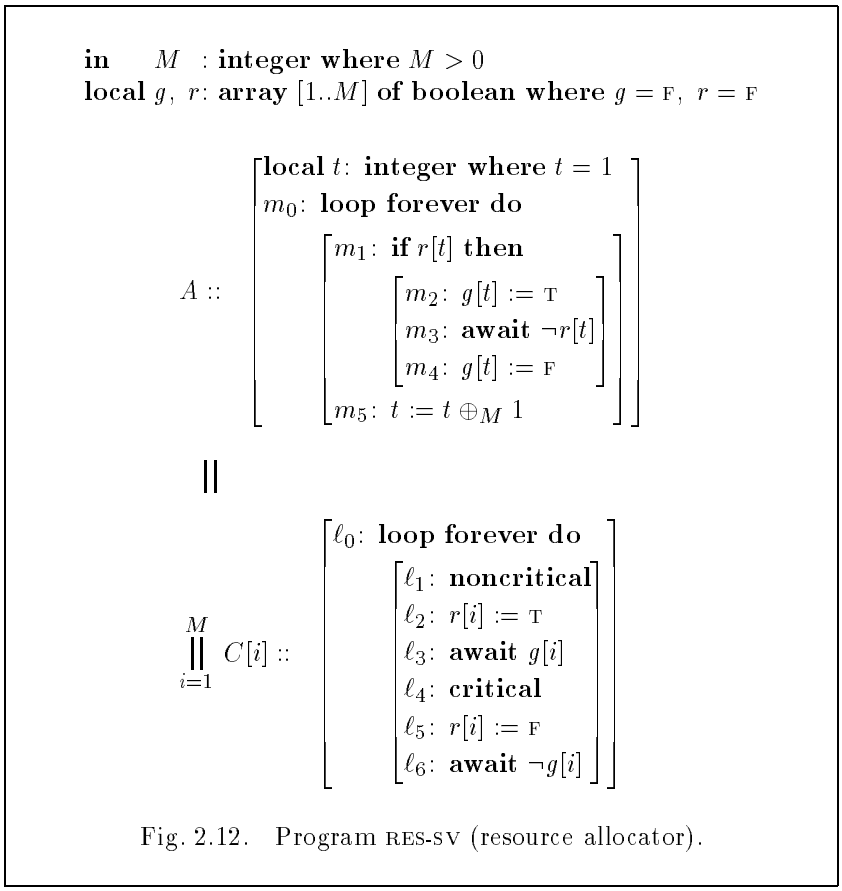


Fig. 2.12. Program RES-SV (resource allocator).

We are interested in proving the response properties of this program. The main response property is that of accessibility, stated by the formula

$$\psi: \quad at\_\ell_2[k] \quad \Rightarrow \quad \diamondsuit \, at\_\ell_4[k].$$

This formula states that any process that is interested in entering its critical section, will eventually succeed in doing so.

We are aided in this proof by the invariant

$$\varphi[i]: \quad idle[i] \ \vee \ requesting[i] \ \vee \ granted[i] \ \vee \ releasing[i]$$

holding for every $i = 1, \ldots, M$, where

$$\begin{aligned}
idle[i]: &\quad \big(at_m_{0,1,5} \ \vee \ t \neq i\big) \ \wedge \ at_\ell_{0..2,6}[i] \ \wedge \ \neg r[i] \ \wedge \ \neg g[i] \\
requesting[i]: &\quad \big(at_m_{0..2,5} \ \vee \ t \neq i\big) \ \wedge \ at_\ell_3[i] \ \wedge \ r[i] \ \wedge \ \neg g[i] \\
granted[i]: &\quad at_m_3 \ \wedge \ t = i \ \wedge \ at_\ell_{3..5}[i] \ \wedge \ r[i] \ \wedge \ g[i] \\
releasing[i]: &\quad at_m_{3,4} \ \wedge \ t = i \ \wedge \ at_\ell_6[i] \ \wedge \ \neg r[i] \ \wedge \ g[i].
\end{aligned}$$

In Fig. 2.13, we present a diagram (duplicated from Fig. 2.10 of the SAFETY book) which represents the possible movements of the system between the four cases comprising assertion $\varphi[i]$.

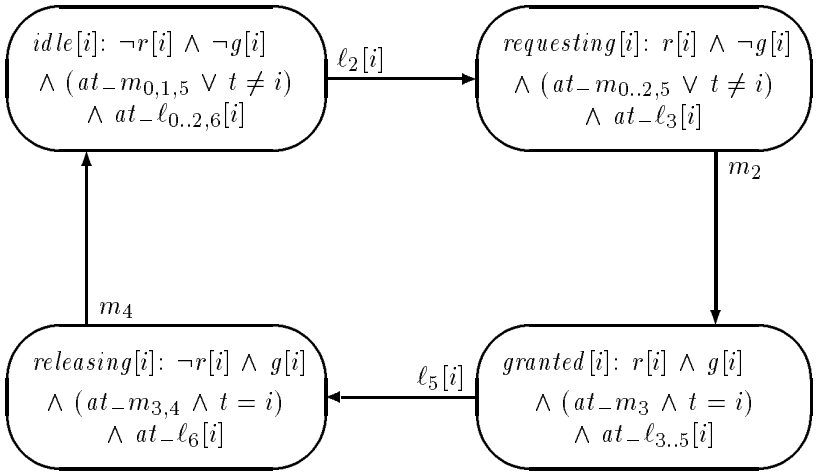


Fig. 2.13. Four phases in the life of a customer process.

The proof is established by the following lemmas:

Lemma A $\quad at\_\ell_2[k] \ \Rightarrow \ \diamondsuit\Big(at\_\ell_3[k] \ \wedge \ \big((at\_m_{0..3} \ \wedge \ t = k) \ \vee \ t \neq k\big)\Big)$

Lemma B $\quad at\_\ell_3[k] \ \wedge \ t \neq k \ \Rightarrow \ \diamondsuit\big(at\_\ell_3[k] \ \wedge \ at\_m_{0..3} \ \wedge \ t = k\big)$

Lemma C $\quad at\_\ell_3[k] \ \wedge \ at\_m_{0..3} \ \wedge \ t = k \ \Rightarrow \ \diamondsuit \, at\_\ell_4[k].$

Lemma A traces the progress of $C[k]$ from $\ell_2[k]$ to $\ell_3[k]$. Deriving at $\ell_3[k]$, we distinguish between two cases according to whether the allocator is currently paying attention to $C[k]$ ($t = k$) or to some other customer $C[i]$, $t = i \neq k$. If the allocator is currently devoted to $C[i]$, $i \neq k$, then Lemma B ensures that $C[k]$'s turn will eventually come. Lemma C shows that, once the allocator pays attention to $C[k]$, process $C[k]$ eventually advances to $\ell_4[k]$.

Clearly, by the transitivity of response (rule TRNS-R), we obtain the required accessibility property.

- *Proof of Lemma A*

  The proof of the response property

$$at\_\ell_2[k] \;\Rightarrow\; \diamondsuit\Big(at\_\ell_3[k] \;\wedge\; \big((at\_m_{0..3} \;\wedge\; t = k) \;\vee\; t \neq k\big)\Big)$$

using rule CHAIN-J, is presented in the diagram of Fig. 2.14.

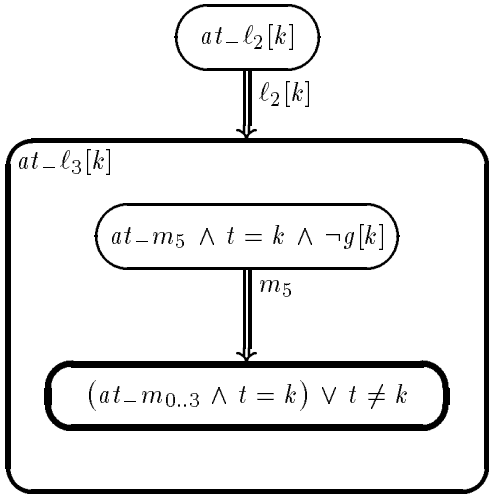


Fig. 2.14. Lemma A for program RES-SV.

The proof of this diagram relies on the invariant $\varphi[k]$, by which

$$at\_\ell_3[k] \;\rightarrow\; \begin{bmatrix} (at\_m_{0..3} \;\wedge\; t = k) \;\vee \\ (at\_m_5 \;\wedge\; t = k \;\wedge\; \neg g[k]) \;\vee \\ t \neq k \end{bmatrix}.$$

To see this observe that, by $\varphi[k]$, $at\_\ell_3[k]$ implies that either *requesting*$[k]$ or *granted*$[k]$ hold. The assertion *requesting*$[k]$ implies

$$(at\_m_{0..2,5} \ \vee \ t \neq k) \ \wedge \ \neg g[k],$$

which is equivalent to

$$\big((at\_m_{0..2,5} \ \wedge \ t = k) \ \vee \ t \neq k\big) \ \wedge \ \neg g[k].$$

This implies

$$(at\_m_{0..2} \ \wedge \ t = k) \ \vee \ \big(at\_m_5 \ \wedge \ t = k \ \wedge \ \neg g[k]\big) \ \vee \ t \neq k.$$

Taking the disjunction of this formula with $at\_m_3 \wedge t = k$, which is implied by $granted[k]$, we obtain the right-hand side of the implication above.

- *Proof of Lemma B*

  The proof of the response property

  $$at\_\ell_3[k] \ \wedge \ t \neq k \ \ \Rightarrow \ \ \diamondsuit\big(at\_\ell_3[k] \ \wedge \ at\_m_{0..3} \ \wedge \ t = k\big),$$

  using rule WELL-J, is presented in Fig. 2.15.

Let us trace progress towards the goal φ_0 in this verification diagram, starting from a state in which $C[k]$ is at ℓ_3 and $t \neq k$. The diagram analyze the various locations in which A can currently be. If A is at m_0 (φ_9) it eventually moves to m_1 (φ_8). At m_1, A tests the value of $r[t]$. If $r[t] = \text{T}$, A moves to m_2 (φ_7). Otherwise, A moves to m_5 (φ_1). According to invariant $\varphi[t]$ (i.e., $\varphi[i]$ when $i = t$), when A is at m_2, $g[t] = \text{F}$ and since, on entering m_2, A has just tested $r[t]$ to be T, we obtain the conjunction $at\_m_2 \wedge r[t] \wedge \neg g[t]$ in φ_7. By $\varphi[t]$, $r[t] \wedge \neg g[t]$ implies $at\_\ell_3[t]$ which complete φ_7. From m_2, A eventually sets $g[t]$ to T and moves to m_3. At this point, we analyze the various possible locations of $C[t]$, i.e., the customer process $C[i]$, $i = 1, \ldots, M$, such that currently $t = i$.

Since at φ_7, $C[t]$ is at $at\_\ell_3[t]$ and $r[t] = \text{T}$, transition m_2 leads to the state described by φ_6. From ℓ_3, $C[t]$ moves to ℓ_4 (φ_5) and from there to ℓ_5 (φ_4).

Taking ℓ_5, $r[t]$ is reset to F and $C[t]$ moves to ℓ_6 (φ_3) waiting for $g[t]$ to become F. Up until this point, A cannot move because it waits for $r[t]$ to become F. Once $r[t] = \text{F}$, A moves to m_4 (φ_2) and from there to m_5 (φ_1).

At m_5, A sets t to $t \oplus_M 1$ advancing to the next process index in cyclic order. There are two cases. If $t = (k-1) \ mod \ M$ then $t \oplus_M 1 = k$ and transition m_5 leads to a state in which A is at m_0, $t = k$, and $C[k]$ is still waiting at ℓ_3.

If $t \neq (k-1) \ mod \ M$ then m_5 leads again to a state covered by φ_9. Here we must show that this transition causes a rank decrease. This means that

$$\big(\delta(t,k), \ 1\big) \ > \ \big(\delta(t \oplus_M 1, \ k), \ 9\big),$$

for $t \neq k$. To show this, we will prove $\delta(t,k) > \delta(t \oplus_M 1, k)$ for $t \neq k$.

Recall that the cyclic distance function was defined in Section 3.4 of the SAFETY book as

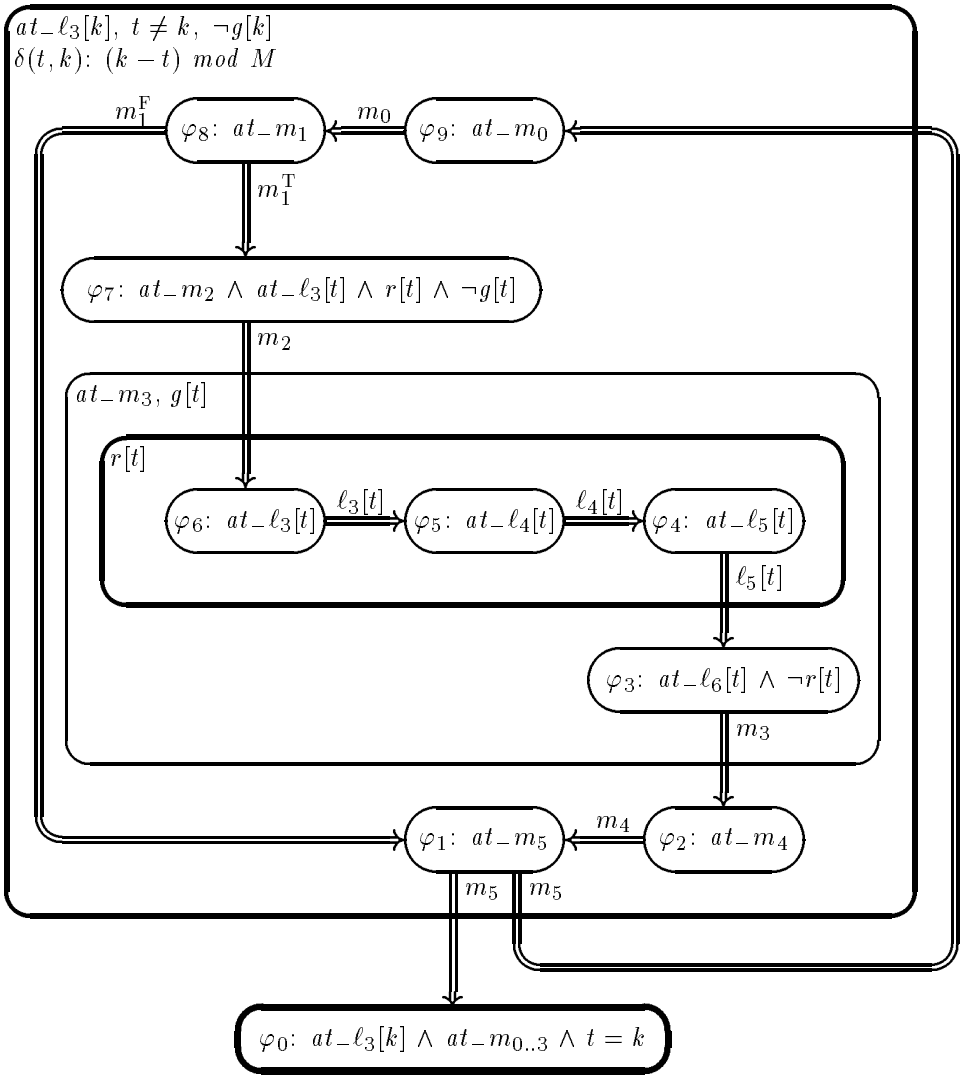


Fig. 2.15. Lemma B for program RES-SV.

$$\delta(t,k): \quad (k-t)\ mod\ M.$$

This function measures the number of times t has to be increased by 1 modulo M until it equals k.

It is obvious that if $t \neq k$ then $t \oplus_M 1$ is closer to k in a cyclic distance than t. For example, consider a situation where $M = 8$, $t = 8$ and $k = 3$.

Observe that

$$t \oplus_8 1 \;=\; 8 \oplus_8 1 \;=\; (8 \;mod\; 8) + 1 \;=\; 1.$$

The two cyclic distances are

$$\delta(t,k) \;=\; \delta(8,3) \;=\; (3-8) \;mod\; 8 \;=\; (-5) \;mod\; 8 \;=\; 3$$
$$\delta(t \oplus_8 1, k) \;=\; \delta(1,3) \;=\; (3-1) \;mod\; 8 \;=\; 2 \;mod\; 8 \;=\; 2.$$

Note that the restriction to $t \neq k$ is essential, because if $t = k$ then

$$\delta(t,k) \;=\; \delta(k,k) \;=\; 0 \;\not>$$
$$\delta(t \oplus_M 1,\; k) \;=\; \delta(k \oplus_M 1,\; k) \;=\; (-1) \;mod\; M \;=\; M-1.$$

Let us prove the rank decrease for arbitrary $M > 1$ and arbitrary $t \neq k$, using the mathematical definitions of $\oplus_M$ and $\delta(t,k)$.

Let n denote $\delta(t,k) = \big((k-t) \;mod\; M\big)$. Obviously $0 \leq n < M$. From the facts that $0 < t \leq M$, $0 < k \leq M$, and $t \neq k$, it follows that $n \neq 0$. Using $0 < n \leq M$, we compute

$$\begin{aligned}
\delta(t \oplus_M 1,\; k) \;&=\; \big(k - (t \oplus_M 1)\big) \;mod\; M \;=\; \Big(k - \big((t \;mod\; M) + 1\big)\Big) \;mod\; M \\
&=\; \big((k-1) - (t \;mod\; M)\big) \;mod\; M \\
&=\; (k-1-t) \;mod\; M \\
&=\; \big(-1 + (k-t)\big) \;mod\; M \\
&=\; \big(-1 + ((k-t) \;mod\; M)\big) \;mod\; M \\
&=\; (-1+n) \;mod\; M \;=\; (n-1) \;mod\; M \\
&=\; n - 1 <\; n \;=\; \delta(t,k).
\end{aligned}$$

This computation uses the fact that, for C satisfying $0 \leq C < M$, $C \;mod\; M = C$, and the following property of the *mod* operator:

$$\big(a \pm (b \;mod\; M)\big) \;=\; (a \pm b) \;mod\; M,$$

which allows the introduction and elimination of $mod\; M$ within the scope of another $mod\; M$.

- *Proof of Lemma C*

  The response property

  $$at\_\ell_3[k] \;\wedge\; at\_m_{0..3} \;\wedge\; t = k \;\;\Rightarrow\;\; \diamondsuit\, at\_\ell_4[k],$$

  is proven by rule CHAIN-J. The proof is presented in the diagram of Fig. 2.16. ▟

The Bakery Algorithm

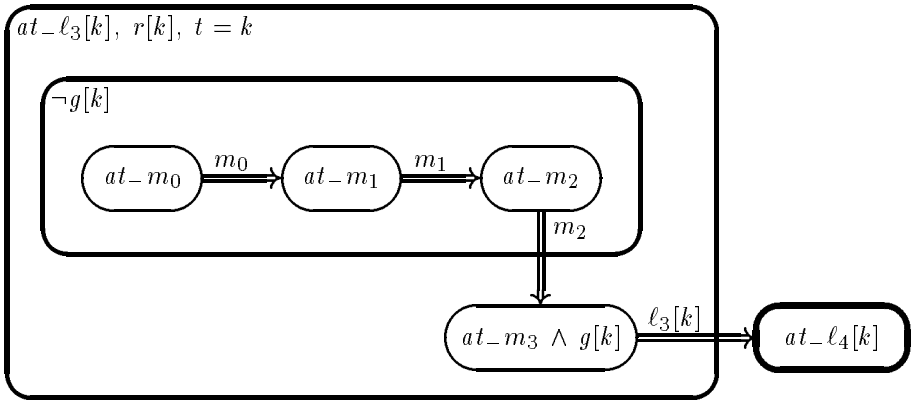


Fig. 2.16. Lemma C for program RES-SV.

In Fig. 2.17, we present an atomic version of the bakery algorithm for mutual exclusion among $M > 0$ processes.

The algorithm is called the *bakery* algorithm, since it is based on the idea that customers, as they enter, pick numbers which form an ascending sequence. Then, a customer with a lower number has higher priority in accessing its critical section. Statement ℓ_2 ensures that the number assigned to $y[i]$ is greater than the current value of $ticket[j]$, for all $j = 1, \ldots, M$. Then statement ℓ_3 admits process $P[i]$ into the critical section only if, for all $j \neq i$, either $y[j] = 0$, implying that process $P[j]$ is currently not interested in entering its cricial section, or $ticket[j] > ticket[i]$ which implise that $P[j]$ has a higher turn number and, therefore, a lower priority than $P[i]$.

We refer to this version of the program as *atomic* because it assumes that taking the maximum of $ticket[1], \ldots, ticket[M]$ can be done in one step and, similarly, testing that all $P[j]$ for $j \neq i$ have lower priority than $P[i]$ can be done in one step.

Proving Mutual Exclusion

To prove the property of mutual exclusion, we first establish some local invariants, relating the value of $ticket[i]$ to the location of process $P[i]$.

$$\chi_0\colon \quad ticket[i] \geq 0$$

$$\chi_1\colon \quad ticket[i] > 0 \quad \leftrightarrow \quad at\_\ell_{3..5}[i].$$

Consider two distinct processes, $P[i]$ and $P[k]$, for $i \neq k$. Their mutual exclusion can be specified by the invariance of the assertion

$$\psi\colon \quad \neg\big(at\_\ell_4[i] \ \wedge \ at\_\ell_4[k]\big).$$

$$\begin{array}{l}\textbf{in} \quad M \quad : \textbf{integer where } M > 0 \\ \textbf{local } \mathit{ticket}\textbf{: array}[1..M] \textbf{ of integer } \textbf{ where } \mathit{ticket} = 0\end{array}$$

$$\Big\|_{i=1}^{M} P[i] :: \left[\begin{array}{l}\ell_0\textbf{: loop forever do} \\ \quad \left[\begin{array}{l}\ell_1\textbf{: noncritical} \\ \ell_2\textbf{: } \mathit{ticket}[i] := 1 + \mathit{max}(\mathit{ticket}[1], \ldots, \mathit{ticket}[M]) \\ \ell_3\textbf{: await } \forall j \neq i\textbf{: } \mathit{ticket}[j] = 0 \ \vee \ \mathit{ticket}[i] < \mathit{ticket}[j] \\ \ell_4\textbf{: critical} \\ \ell_5\textbf{: } \mathit{ticket}[i] := 0\end{array}\right]\end{array}\right]$$

Fig. 2.17. Program BAKERY-AT (atomic bakery algorithm).

Applying the technique of assertion propagation (described in Chapter 1 of the SAFETY book), we obtain the following necessary condition for ψ being invariant (taking into account invariant χ_0):

$$\kappa\text{:} \quad at\_\ell_3[i] \ \wedge \ at\_\ell_4[k] \ \rightarrow \ \exists j \neq i\text{: } 0 < \mathit{ticket}[j] \leq \mathit{ticket}[i].$$

A good heuristic is to try to establish mutual exclusion of $P[i]$ and $P[k]$, using assertions that only refer to the variables of $P[i]$ and $P[k]$. Therefore we may look for the weakest (i,k)-assertion which implies κ. This is given by

$$at\_\ell_3[i] \ \wedge \ at\_\ell_4[k] \ \rightarrow \ 0 < \mathit{ticket}[k] \leq \mathit{ticket}[i].$$

Since $at\_\ell_4[k]$ implies $\mathit{ticket}[k] > 0$ by χ_1, we can simplify the candidate assertion to

$$\chi_3\text{:} \quad at\_\ell_3[i] \ \wedge \ at\_\ell_4[k] \ \rightarrow \ \mathit{ticket}[k] \leq \mathit{ticket}[i].$$

It is not difficult to ascertain that χ_3 is an invariant (using χ_0 and χ_1).

Informally, we may consider the two relevant transitions.

- Taking $\ell_3[k]$ while $at\_\ell_3[i]$. Transitions $\ell_3[k]$ can be taken only if $\mathit{ticket}[i] = 0$ or $\mathit{ticket}[k] < \mathit{ticket}[i]$. By χ_1, $at\_\ell_3[i]$ implies $\mathit{ticket}[i] > 0$. Therefore, taking the transition implies $\mathit{ticket}[k] < \mathit{ticket}[i]$ which guarantees $\mathit{ticket}[k] \leq \mathit{ticket}[i]$.
- Taking $\ell_2[i]$. This transitions sets $\mathit{ticket}[i]$ to a value which is not smaller than $1 + \mathit{ticket}[k]$. Hence $\mathit{ticket}[k] \leq \mathit{ticket}[i]$.

In a symmetric way, we propose and verify the invariance of

$$\chi_4\text{:} \quad at\_\ell_3[k] \ \wedge \ at\_\ell_4[i] \ \rightarrow \ \mathit{ticket}[i] \leq \mathit{ticket}[k].$$

It is now straightforward to verify the invariance ψ, relying on χ_3 and χ_4 as

previously established invariances.

Proving Accessibility

The property of accessibility for process $P[i]$ can be specified by the formula

$$\alpha: \quad at\_\ell_2[i] \quad \Rightarrow \quad \diamondsuit\, at\_\ell_4[i].$$

It is trivial to follow the progress of $P[i]$ from ℓ_2 to ℓ_3. Consequently, the main verification task should establish the property

$$\beta: \quad at\_\ell_3[i] \quad \Rightarrow \quad \diamondsuit\, at\_\ell_4[i].$$

The only reason process $P[i]$ may be held at ℓ_3, not being able to proceed immediately to ℓ_4 is that there are some processes *superior* to $P[i]$. A process $P[j]$ is said to be superior to $P[i]$ if $0 < ticket[j] < ticket[i]$. Consequently, we define

$$superior(i) \;=\; \{j \in [1..M] \mid 0 < ticket[j] < ticket[i]\}$$

to be the set of all (indices of) processes superior to $P[i]$.

Consider what may happen to the set $superior(i)$ while $P[i]$ is waiting at ℓ_3. Processes may leave the set $superior(i)$ by performing ℓ_5, setting their *ticket* variable to 0. Once a process has left $superior(i)$ it cannot reenter this set because the only way for a process to attain a positive ticket value is by performing ℓ_2. However, execution of ℓ_2 by $P[j]$ while $P[i]$ is at ℓ_3, necessary leads to a value $ticket[j] > ticket[i]$, which shows that $P[j]$ is not in $superior(i)$ even after execution of ℓ_2.

We conclude that while $P[i]$ is waiting at ℓ_3, the set $superior(i)$ can never increase. We will now argue that if the set is non-empty, it will eventually decrease.

Consider process $P[j]$ which has the minimal positive value of $ticket[j]$. If $superior(i) \neq \emptyset$ then, obviously $j \in superior(i)$. Having a positive ticket value implies that $P[j]$ is somewhere in the range $\{\ell_3, \ell_4, \ell_5\}$. We claim that wherever it is, its progress is guaranteed. In particular, even if $P[j]$ is at ℓ_3 it cannot be blocked by any other process, since $ticket[j]$ is (positively) minimal.

Thus, $P[j]$ is guaranteed to progress until it reaches ℓ_5, whose execution decreases the size of $superior(i)$.

Consequently, while $P[i]$ is waiting at ℓ_3, the set $superior(i)$ repeatedly decreases, until it becomes empty. Once $superior(i)$ is empty, $P[i]$ has the minimal positive ticket and its progress to ℓ_4 is guaranteed.

In Fig. 2.18, we present a P-RANK diagram for the proof of property β based on the above arguments. The proof uses the assertion

$$minimal(k): \quad ticket(k) > 0 \wedge \forall j\colon ticket(j) > 0 \;\rightarrow\; ticket(k) \leq ticket(j).$$

In **Problem 2.3**, the reader is requested to consider several non-atomic versions of the bakery algorithm and verify their properties.

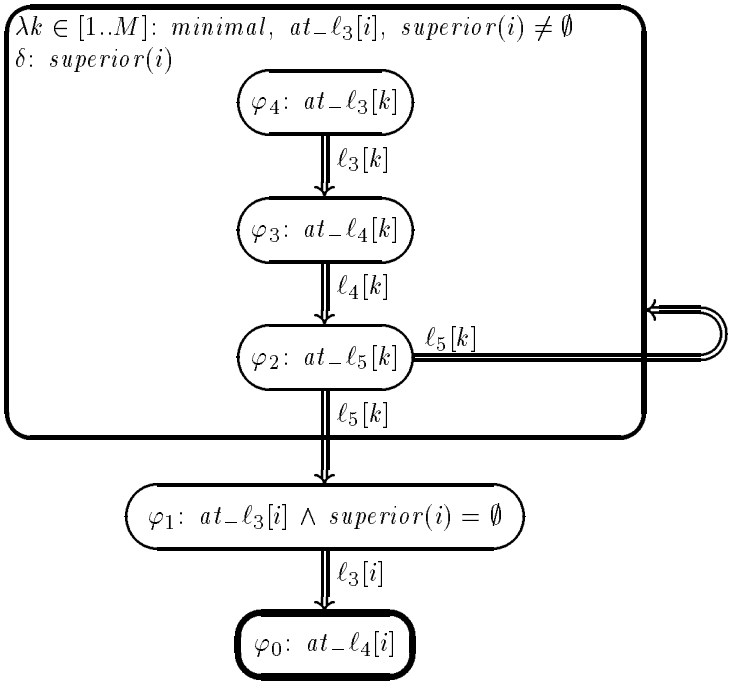


Fig. 2.18. A P-RANK diagram proving accessibility for program BAKERY-AT.

2.3 Coordination by Add-And-Store

As our next set of examples, illustrating the techniques for proving response properties of parameterized programs, we present a family of algorithms, based on a special synchronization instruction.

Some computers have special "add-and-store" instructions that fetch a value from a location in memory, add to it a number and store the result in the same location, all within one execution cycle. If several processors that have this instruction are connected to a common shared memory then, while one of them performs such an add-and-store instruction, no other processor can interfere with the value being fetched and stored at this location. This provides a natural implementation of the atomic instruction $y := y + e$ for a shared variable y and a *local expression* e. A local expression is an expression that depends only on variables local to the process. Usually, the value which is stored is also retained in an internal register. This enables a single transition implementation of the multiple assignment statement

$$(t,\, y) \;:=\; (y + e,\, y + e),$$

where y is a shared variable, e is a local expression, and t is a variable local to the process that is executing this instruction.

To emphasize the special role of this statement, it is written in the form

$$t := y := y + e,$$

in all the programs in this section.

Add-and-store statements of the described form have often been suggested as means for implementing semaphores. As we will see below, they are not as strong as semaphores, since they lack the compassion property associated with semaphores. That is, the transitions associated with add-and-store statements are not compassionate while semaphore transitions are.

We present two programs that coordinate entry to critical sections by means of add-and-store statements and investigate their response properties. Since these statements are not compassionate, they cannot directly ensure individual accessibility of each process to its critical section. Instead, they can only ensure the weaker property of cummunal accessibility by which, if some process wishes to enter the critical section, some process (not necessarily the same) will eventually enter its critical section. However, as will be shown, communal accessibility can be used to program more involved algorithms that ensure individual accessibility.

Notations and Inferences

To facilitate proofs of the programs in this section, we introduce some special notations that rely on the special structure of the considered programs. All programs considered here consist of a parallel composition of M processes $P[1], \ldots, P[M]$, where M is an input to the program. Each process has a local variable t. Let ℓ_a and ℓ_b, $a \leq b$ be two locations in the program. We define the following notations:

$$L^{t\sim c}_{a..b}: \quad \left\{ i \in [1..M] \;\mid\; at\_\ell_{a..b}[i] \;\wedge\; t[i] \sim c \right\},$$

$$N^{t\sim c}_{a..b}: \quad \left| L^{t\sim c}_{a..b} \right|$$

where $\sim$ is one of the six binary relations $\{<, \leq, =, \neq, \geq, >\}$.

Thus, $L^{t>c}_{3..4}$ denotes the set of all process indices $i \in [1..M]$, such that $P[i]$ is currently at ℓ_3 or at ℓ_4 and $t[i] > 0$, and $L^{t>c}_{3..4}$ denotes the total number of such processes.

For the case that $b = a$, we write $L^{t\sim c}_{a..b}$ and $N^{t\sim c}_{a..b}$ simply as $L^{t\sim c}_{a}$ and $N^{t\sim c}_{a}$. For the case that $b = a + 1$, we write $L^{t\sim c}_{a..b}$ and $N^{t\sim c}_{a..b}$ simply as $L^{t\sim c}_{a,b}$ and $N^{t\sim c}_{a,b}$.

There is an obvious connection between assertions referring to expressions such as $N^{t\sim c}_{a..b}$ and assertions that use existential quantification over the index i. For example, the following equivalence is a direct consequence of the definitions.

$$N_{a..b}^{t>0} > 0 \quad \leftrightarrow \quad \exists i\colon \big(at\_\ell_{a..b}[i] \ \wedge \ t[i] > 0\big).$$

All quantified variables in this section are rigid variables denoting process indices which range over $[1..M]$. We therefore omit their range specifications from the formulas.

Some of the following proofs use rule EE-INTR, which can be derived in the proof system presented in Volume I.

Rule EE-INTR (introduction of existential quantifiers)

$$\frac{p \ \Rightarrow \ \diamondsuit q}{(\exists i\colon p) \ \Rightarrow \ \diamondsuit(\exists i\colon q)}$$

In the case that formula q has no free occurrences of i, we can use this rule to infer $(\exists i\colon p) \Rightarrow \diamondsuit q$ from $p \Rightarrow \diamondsuit q$.

Mutual Exclusion by Add-and-Store

We consider first program MUX-AST presented in Fig. 2.19, which implements mutual exclusion by add-and-store instructions.

in M: **integer where** $M > 0$
local y : **integer where** $y = 1$

$$\Big\|_{i=1}^{M} P[i] :: \left[\begin{array}{l} \textbf{local } t\colon \textbf{integer} \\ \ell_0\colon \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\colon \textbf{noncritical} \\ \ell_2\colon t := -1 \\ \ell_3\colon \textbf{while } t < 0 \textbf{ do} \\ \quad \left[\begin{array}{l} \ell_4\colon \textbf{await } y > 0 \\ \ell_5\colon t := y := y - 1 \\ \ell_6\colon \textbf{if } t = 0 \textbf{ then} \\ \quad \ell_7\colon \textbf{critical} \\ \ell_8\colon y := y + 1 \end{array}\right] \end{array}\right] \end{array}\right]$$

Fig. 2.19. Program MUX-AST (mutual exclusion by add-and-store).

The program consists of $M > 0$ processes which coordinate their entry to their critical sections via the shared "semaphore" variable y. In addition, each process $P[i]$ has a local variable t which is used in the program. The program for each process is a cycle, in which the process alternates between the execution of its noncritical section at ℓ_1 and the attempt to enter its critical section at ℓ_7.

The process may elect to stay forever in its noncritical section.

If the process exits its noncritical section, the protocol for trying to enter the critical section starts at ℓ_2, where the variable t is set to -1. Then, the process enters a loop at ℓ_3 which is terminated only when t becomes nonnegative. A nonnegative value of t (actually 0), signals that the process managed to enter its critical section, and may therefore return to ℓ_1.

Within the loop, the process waits first at ℓ_4 for y to become positive. When the process detects a positive y it performs an add-and-store statement at ℓ_5 with y and the constant -1, retaining the value stored in y also in t.

One may wonder why do we need further checks after having detected a positive y at ℓ_4. The answer is that two processes $P[i]$ and $P[j]$ may detect a positive y at ℓ_4 and move, one after the other, to ℓ_5. Assume $P[i]$ to be the first to perform ℓ_5. It will lower y to 0 and obtain a $t[i] = 0$. The second process $P[j]$, performing ℓ_5 one step later, will get to $y = t[j] = -1$. Therefore, it is necessary for a process to check its own t after the addition to see whether it was the first to perform ℓ_5.

Indeed, at ℓ_6 process $P[i]$ checks whether it was the lucky one. On finding $t[i] = 0$, $P[i]$ concludes that it was the first to perform ℓ_5 and proceeds to enter the critical section at ℓ_7. On exit it increments y, offsetting the subtraction at ℓ_5. Since $t[i] = 0$ in this case, the loop of ℓ_3 terminates, and lucky process $P[i]$ returns to ℓ_1.

If the process finds a negative t, it concludes that it is not its turn to enter, and proceeds to ℓ_8. Here it increments y to offset the subtraction at ℓ_5. Since $t < 0$, the loop of ℓ_3 does not terminate, and the process returns to ℓ_4 to try its luck once more.

Proving Safety Properties: Mutual Exclusion

As a first step in the verification of program MUX-AST, let us prove the safety property of mutual exclusion. This property states that at most one process may be at ℓ_7, and is expressible by:

$$N_7 \ \leq \ 1.$$

We prove several invariants that lead to the desired conclusion.

The first invariant is given by

$$I_1\colon \quad at\_\ell_7[i] \quad \rightarrow \quad t[i] = 0.$$

Observe that the assertions refer to the instance of variable t that belongs to process $P[i]$ as $t[i]$. This invariant can be verified by checking the relevant transitions. They are $\ell_6[i]$ which proceeds to ℓ_7 only if $t[i] = 0$, and $\ell_2[i]$, $\ell_5[i]$ which modify $t[i]$, but do not end up at $\ell_7[i]$.

A second invariant is given by

$$I_2: \quad N_{6..8} + y \;=\; 1.$$

This is easily verifiable by observing that initially $N_{6..8} = 0$ and $y = 1$. Then we check the relevant transitions and find that $\ell_5[i]$ decrements y by 1 but increments $N_{6..8}$ by 1. The latter is because process $P[i]$ has just joined the set $L_{6..8}$ by entering ℓ_6. Similarly, transition ℓ_8 increments y by 1 but decrements $N_{6..8}$ by 1. Thus, both preserve the sum $N_{6..8} + y$.

From this invariant it follows that $y \leq 1$ is also invariant.

The last invariant we consider is:

$$I_3: \quad i \neq j \;\rightarrow\; \neg\Big(\big(at\_\ell_{6..8}[i] \wedge t[i] = 0\big) \;\wedge\; \big(at\_\ell_{6..8}[j] \wedge t[j] = 0\big)\Big)$$

which states that at most one process $P[i]$ can be at $\ell_{6..8}$ with a zero t variable.

The only transition which may potentially falsify this assertion is taking $\ell_5[i]$ from a state satisfying $at\_\ell_{6..8}[j] \wedge t[j] = 0$ (symmetrically, taking $\ell_5[j]$ from a state satisfying $at\_\ell_{6..8}[i] \wedge t[i] = 0$). However, in this case $N_{6..8} > 0$ which, in view of I_2, implies $y \leq 0$. Since transition $\ell_5[i]$ leads to $t'[i] = y - 1$, it follows that $t'[i] < 0$. Therefore, i does not satisfy $at'\_\ell_{6..8}[i] \wedge t'[i] = 0$ and the assertion I_3 is not falsified.

This shows that all transitions preserve assertion I_3.

From I_1 and I_3 we conclude

$$N_7 \;\leq\; 1.$$

Proving Response Property: Communal Accessibility

As we have already commented, the add-and-store instructions are not strong enough to guarantee individual accessibility, i.e., that *each* process wishing to enter its critical section will eventually do so. Instead, we prove the weaker property of *communal accessibility* (also called *weak accessibility*). This property claims that, if some process wishes to enter its critical section, then *some* process (not necessarily the same) will eventually enter its critical section. This can be expressed by:

$$N_2 > 0 \quad \Rightarrow \quad \diamondsuit(N_7 > 0).$$

Let us prove this property.

Consider the progress of a process from ℓ_2 to ℓ_7. It can certainly advance unhindered until it reaches ℓ_4. At ℓ_4 it has to wait until y becomes positive. Due

to I_2, y becomes positive only when the range $\ell_{6..8}$ is evacuated by all processes currently residing there. When y becomes positive, one or more processes will reach ℓ_5. The first process to execute ℓ_5, while y is positive, will get a zero value of t and will eventually get to ℓ_7.

Thus, we can partition the description of progress into phases, identified as follows:

1. Some process gets to ℓ_4.
2. While some processes are at ℓ_4 and $y \leq 0$, the range $\ell_{6..8}$ is evacuated until y becomes 1.
3. Some process executes ℓ_5 while y is still 1, gets a zero t, and proceeds to ℓ_7.

Consequently, we prove these stages in the progress towards entry to the critical section as separate lemmas.

Lemma A $\quad N_2 > 0 \;\Rightarrow\; \diamondsuit(N_4 > 0)$

In Fig. 2.20 we present a CHAIN-J proof of the response formula

$$at\_\ell_2[i] \;\Rightarrow\; \diamondsuit(N_4 > 0).$$

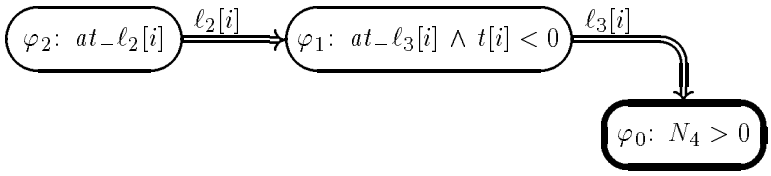


Fig. 2.20. CHAIN diagram for Lemma A.

Applying rule EE-INTR and observing that $N_4 > 0$ does not refer to the index i, we obtain

$$(\exists i\colon\ at\_\ell_2[i]) \;\Rightarrow\; \diamondsuit(N_4 > 0),$$

from which, by monotonicity, we conclude

$$N_2 > 0 \;\Rightarrow\; \diamondsuit(N_4 > 0).$$

Once we know that some process is at ℓ_4, there are two cases to consider, depending on whether y is positive. Consider first the more difficult one, in which $y \leq 0$. The following lemma claims that from the more difficult case, we are guaranteed to reach the easier case in which $N_4 > 0 \wedge y > 0$.

Lemma B $\quad N_4 > 0 \;\wedge\; y \leq 0 \;\Rightarrow\; \diamondsuit(N_4 > 0 \;\wedge\; y > 0)$

We prove this lemma by rule WELL-JP. The proof is presented by the P-RANK diagram of Fig. 2.21.

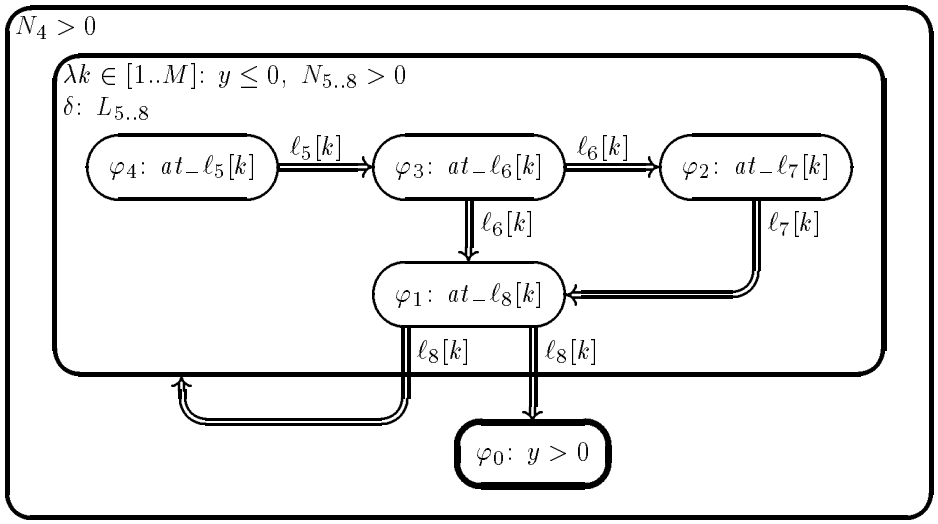


Fig. 2.21. P-RANK diagram for Lemma B.

Note that any transition that causes $N_{6..8}$ to become 0 leads, according to I_2, to a positive y.

The reason we use the ranking $N_{5..8}$ instead of $N_{6..8}$ which is mentioned in I_2, is that $N_{6..8}$ can increase by execution of ℓ_5. The rank $N_{5..8}$ cannot increase, while $y \leq 0$, because no process can pass ℓ_4 if $y \leq 0$.

Lemma C $\quad N_4 > 0 \ \wedge \ y > 0 \ \Rightarrow \ \diamondsuit(N_7 > 0)$

This lemma is proven again by rule WELL-JP, tracing the progress from ℓ_4 to ℓ_7. The proof is presented in the P-RANK diagram of Fig. 2.22.

We use the invariant I_2 to infer $y > 0 \rightarrow y = 1$.

The three lemmas can be combined into a single P-RANK diagram, presented in Fig. 2.23.

It is possible to omit most of the internal details of this diagram, and retain just the top-level structure, identifying the lemmas which lead from one phase to the next. This leads to the diagram of Fig. 2.24.

This schematic diagram identifies the partition of the proof into three lemmas. Each lemma is represented as a box, with exits that lead to subsequent

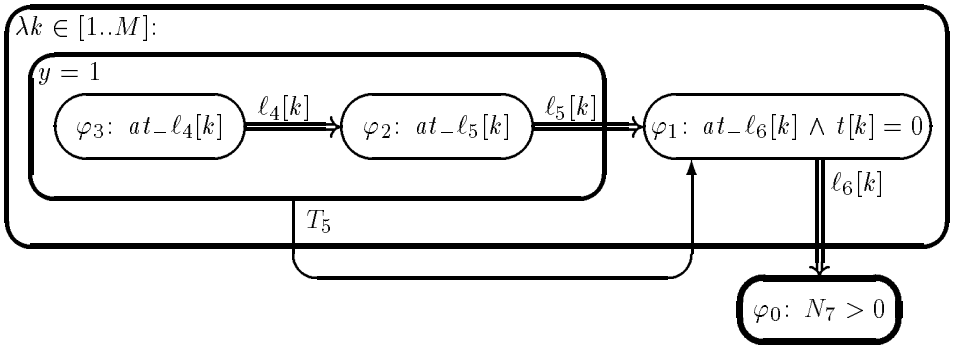


Fig. 2.22. P-RANK diagram for Lemma C.

boxes. The box represents the response property, guaranteeing that eventually the computation will exit to one of the successors of this box.

From Communal to Individual Accessibility

Program MUX-AST for mutual exclusion by add-and-store instructions does not fully satisfy our expectations from a satisfactory solution to the mutual exclusion problem. It falls short in guaranteeing communal accessibility rather than individual accessibility. However, once some process is admitted to the critical section we can appoint it as an arbitrator for the next round. This is because when being in the critical section it can perform activities such as determining which process will be the next to be admitted, without having to worry about possible interference from other processes.

In Fig. 2.25 we present program IMUX-AST, which implements individual accessibility. This program extends program MUX-AST in several ways. It contains an additional variable *next* ranging over $1..M$, and whose value represents the index of a process, that has been given higher priority in the last arbitration round. The local variable t of MUX-AST is transformed in IMUX-AST into an array $t[1..M]$ that can be inspected by each of the processes. Initially $next = 0$.

The protocol proceeds very much as before, except that at ℓ_4 we introduced a new gate. The function of the gate at ℓ_4 is to hold there all processes except the one that has been given higher priority. Note that, if no process has been given priority, $next = 0$ and all processes can pass the gate at ℓ_4. From there on, the protocol proceeds as before, except that there is an additional code performing an arbitration round at $\ell_{9..15}$. Since this code follows immediately after the critical section it is guaranteed to be performed in exclusion, by a single process $P[i]$.

The first question asked at ℓ_9 is whether a new round of arbitration is needed.

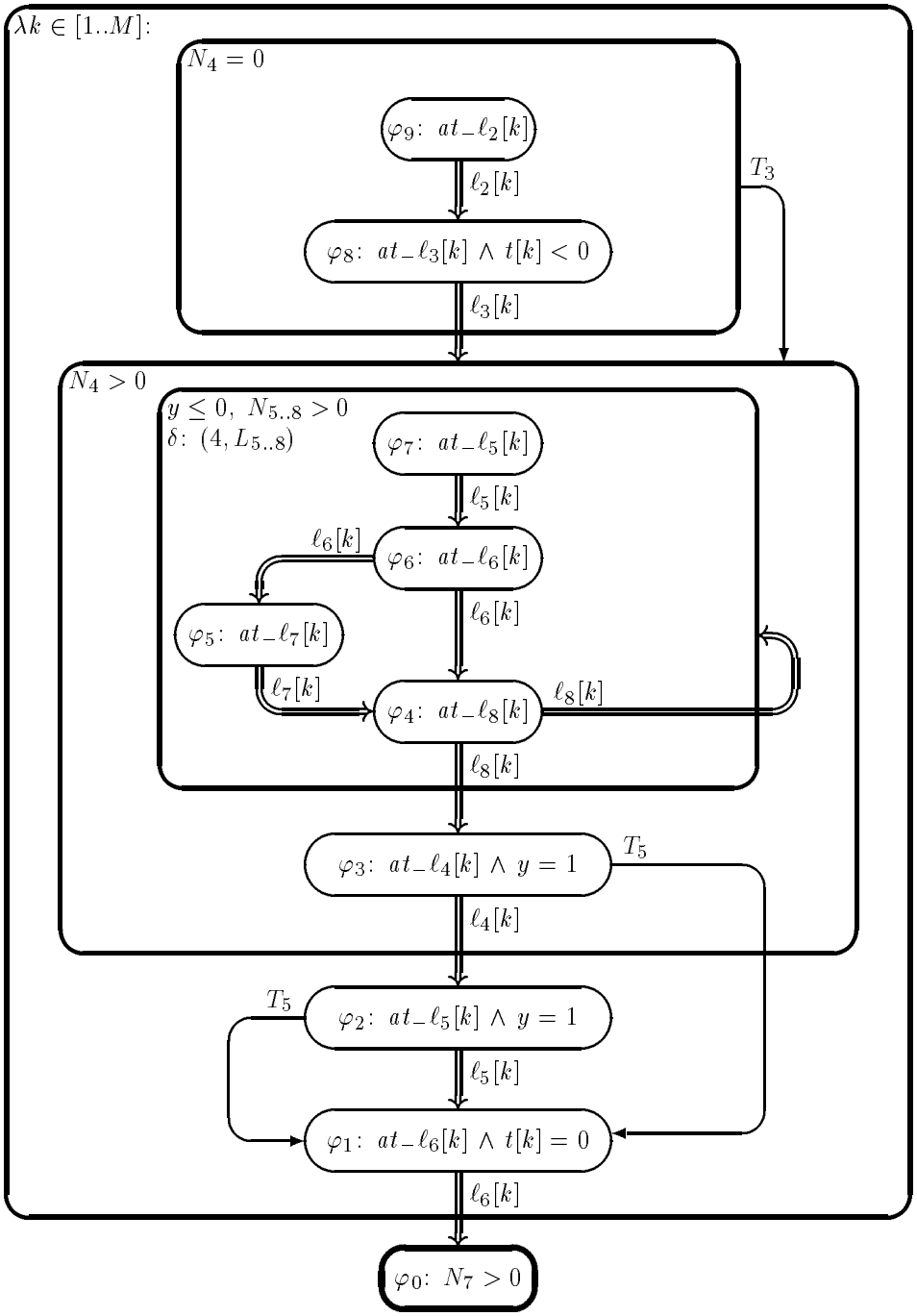


Fig. 2.23. P-RANK diagram for $N_2 > 0 \Rightarrow \diamondsuit(N_7 > 0)$.

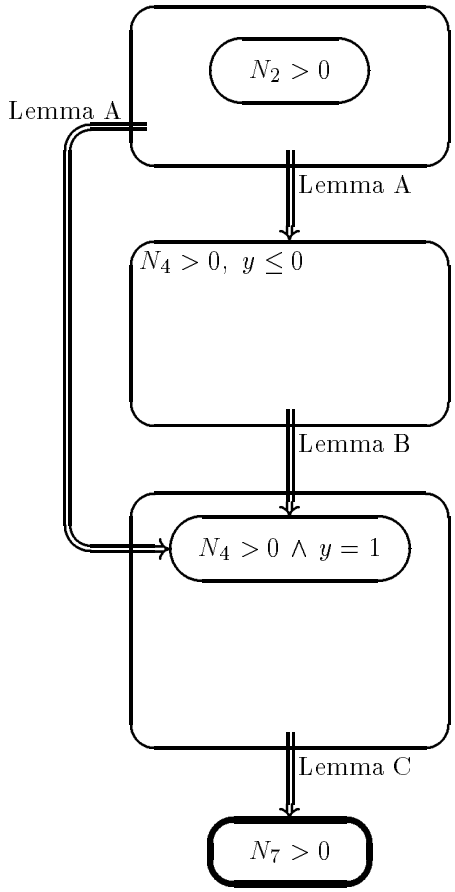


Fig. 2.24. Top-level structure of the proof.

A new round is needed if either no process has been assigned a high priority, i.e., $next = 0$, or if the process with the higher priority is $P[i]$ itself.

The other case is that some other process, $P[k]$, for $k \neq i$ has been assigned a high priority, but $P[i]$ entered the critical section ahead of $P[k]$. The algorithm is such that some overtaking is possible. In this case, $P[k]$ is still attempting entry to its critical section and we should not modify its priority. Hence in such a case, $P[i]$ proceeds directly to ℓ_{16} and no arbitration takes place.

On entering the arbitration section, variable $next$ is reset to 0 for the case that its previous value was i. Process $P[i]$ then sets variable j to the next index following i in cyclic order, and enters the while loop at ℓ_{12}. The loop searches

$$
\begin{array}{lll}
\textbf{in} & M & : \textbf{integer where } M = 0 \\
\textbf{local} & j,\ y,\ next & : \textbf{integer where } y = 1,\ next = 0 \\
 & t & : \textbf{array } [1..M] \textbf{ of integer where } t = 0
\end{array}
$$

$$
\begin{array}{l}
\Big\|_{i=1}^{M} P[i] :: \\
\quad \left[\begin{array}{l}
\ell_0\colon \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\colon \textbf{noncritical} \\
\ell_2\colon t[i] := -1 \\
\ell_3\colon \textbf{while } t[i] < 0 \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_4\colon \textbf{await } next = 0 \vee next = i \\
\ell_5\colon \textbf{await } y > 0 \\
\ell_6\colon t[i] := y := y - 1 \\
\ell_7\colon \textbf{if } t[i] = 0 \textbf{ then} \\
\quad \left[\begin{array}{l}
\ell_8\colon \textbf{critical} \\
\ell_9\colon \textbf{if } next = 0 \ \vee \ next = i \textbf{ then} \\
\quad \left[\begin{array}{l}
\ell_{10}\colon next := 0 \\
\ell_{11}\colon j := (i \bmod M) + 1 \\
\ell_{12}\colon \textbf{while } next = 0 \ \wedge \ j \neq i \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_{13}\colon \textbf{if } t[j] < 0 \textbf{ then} \\
\qquad \ell_{14}\colon next := j \\
\ell_{15}\colon j := (j \bmod M) + 1
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\ell_{16}\colon y := y + 1
\end{array}\right]
\end{array}\right]
\end{array}
$$

Fig. 2.25. Program IMUX-AST — guaranteeing individual accessibility.

for the first j in cyclic order such that $t[j] < 0$. Note that $t[j] < 0$ is a reliable indicator that $P[j]$ is interested in entering the critical section but has not done so yet. If such a j is found, $P[i]$ sets $next$ to j, thus declaring j to have a high priority, and then exits the loop. Another possibility is that no such j has been found, and then j will close a full cycle and return to the value i. The loop terminates in both of these cases.

The reader is invited to verify that this program guarantees individual accessibility, i.e., that it satisfies the requirement

$$at\_\ell_2[k] \quad \Rightarrow \quad \diamondsuit\, at\_\ell_7[k].$$

Readers-Writers with Add-And-Store Instructions

An interesting extension of the mutual exclusion problem is the readers-writers problem. In this problem we distinguish two types of critical sections, called a *reading section* and a *writing section*. The required exclusion property is that

> while some process resides in a writing section, no other process may reside in either a reading or a writing section.

Note that this allows several processes to cohabit a reading section.

The Program

Program READ-WRITE for solving the readers-writers problem, using add-and-store instructions, is presented in Fig. 2.26. (Compare with program READ-WRITE of Fig. 2.11 of the SAFETY book, which uses generalized semaphores.) After the statement *noncritical* at ℓ_1, which represents the noncritical activity of the processes, each process branches nondeterministically to either the *read protocol* R, at location $\ell_{2..7}$, or to the *write protocol* W, at $\ell_{8..14}$.

In the *read protocol*, each reader tries to decrement y and achieve a nonnegative value in t (and in y). Since the initial value of y is M, up to M readers may decrement y by 1, and still obtain a nonnegative value. If a reader succeeds to obtain a nonnegative value for t, it proceeds to the *read section* at ℓ_6. If it does not, it corrects y at ℓ_7 and continues to loop.

The *writer's protocol* is similar, except that a writer subtracts M From y, attempting to obtain a nonnegative value. This is possible only if that particular writer is the only one currently subtracting from y. Even if one reader has subtracted and lowered y to $M-1$, the writer, subtracting a further M will already obtain -1, and be barred from entering the writing section.

Another special feature of the write protocol is the wait at ℓ_{10} for y to become M. This is similar to the **await** $y > 0$ we had in the mutual exclusion program MUX-AST (Fig. 2.25). This is necessary, because otherwise we can construct a computation in which two writers chase one another around the ℓ_9 loop, keeping y always negative, without any of them or a reader being able to enter any of the critical sections. With the *await* gate at ℓ_{10}, this cannot happen as we will prove below.

Proving Mutual Exclusion

The first property we prove is that of exclusion, as required by the problem. This is established by several invariants.

- *Invariant* I_1

  $$N_{5..7} \; + \; M \cdot N_{12..14} \; + \; y \;\; = \;\; M.$$

  This equality holds initially, since $y = M$ and $N_{5..7} = N_{12..14} = 0$. It is

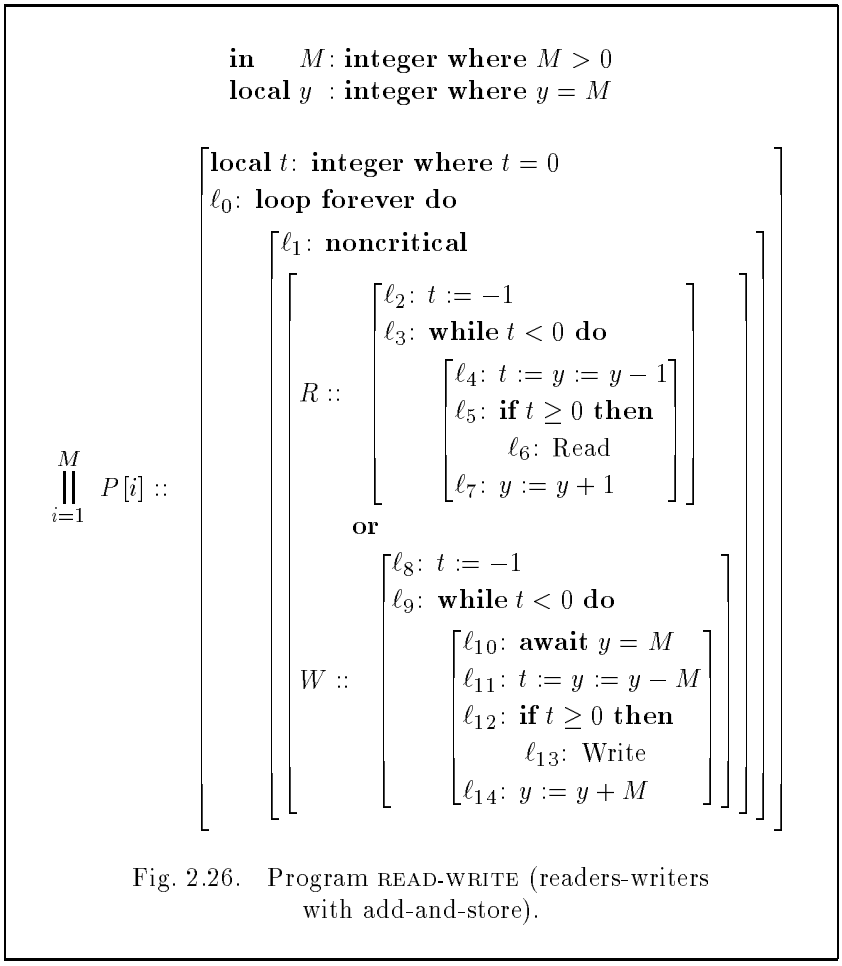


Fig. 2.26. Program READ-WRITE (readers-writers with add-and-store).

preserved by transitions, such as ℓ_4 which increments $N_{5..7}$ by 1 and decrements y by 1, or ℓ_{11} which increments $N_{12..14}$ by 1 and decrements y by M. Similarly, it is preserved by ℓ_7 and ℓ_{14}.

- *Invariant* I_2

$$N_{5..7}^{t\geq 0} \;+\; M \cdot N_{12..14}^{t\geq 0} \;\leq\; M.$$

There are two transitions that may endanger the invariance of this assertion.

- $\ell_4[i]$ while $N_{5..7}^{t\geq 0} + M \cdot N_{12..14}^{t\geq 0} = M$

However, due to I_1,

$$y \;=\; M \;-\; N_{5..7} \;-\; M \cdot N_{12..14}$$

$$\leq \; M \; - \; N_{5..7}^{t\geq 0} \; - \; M \cdot N_{12..14}^{t\geq 0} \; = \; 0.$$

So the execution of $\ell_4[i]$ will produce $t'[i] < 0$, and hence $N_{5..7}^{t\geq 0}$ does not increase.

- $\ell_{11}[i]$ while $N_{5..7}^{t\geq 0} + M \cdot N_{12..14}^{t\geq 0} > 0$

This is a dangerous situation since by adding one more element to $N_{12..14}^{t\geq 0}$ the sum will increase beyond M.

However due to I_1, and a calculation identical to the one before we obtain

$$y \; \leq \; M \; - \; N_{5..7}^{t\geq 0} \; - \; M \cdot N_{12..14}^{t\geq 0} \; < \; M.$$

Consequently, the execution of $\ell_{11}[i]$ produces $t'[i] < 0$, and does not increase $N_{12..14}^{t\geq 0}$.

- *Invariant* I_3

$$L_6 \; \subseteq \; L_{5..7}^{t\geq 0}.$$

This is equivalent to the implication

$$at\_\ell_6[i] \; \rightarrow \; t[i] \geq 0,$$

claimed for every $i = 1, \ldots, M$. The invariant can be verified by considering the transitions that enter ℓ_6. Since a process $P[i]$ can enter ℓ_6 only if $t[i] \geq 0$, it follows that $i \in L_{5..7}^{t\geq 0}$, after the transition.

- *Invariant* I_4

Similarly to the above invariant, we can also prove

$$L_{13} \; \subseteq \; L_{12..14}^{t\geq 0},$$

which is equivalent to

$$at\_\ell_{13}[i] \; \rightarrow \; t[i] \geq 0.$$

From I_3 and I_4 we can conclude

$$N_6 \; + \; M \cdot N_{13} \; \leq \; N_{5..7}^{t\geq 0} \; + \; M \cdot N_{12..14}^{t\geq 0}.$$

Using I_2 we conclude

$$N_6 \; + \; M \cdot N_{13} \; \leq \; M.$$

From this it is easy to infer

$$N_{13} > 0 \; \rightarrow \; N_6 = 0 \; \wedge \; N_{13} = 1,$$

which is precisely the exclusion property required. It states that if some process is at ℓ_{13}, then it is the only one there, and no process is at ℓ_6.

Proving Accessibility

The response property we prove for this program is again that of *communal accessibility*. Since this program has two types of critical sections, a *read* and a *write* section, communal accessibility has an even broader interpretation. It states that if some process is interested in entering *one* of the critical sections, then some process will eventually enter *one* of the critical sections. We cannot even guarantee, for example, that if some process wants to read (write) then some process will eventually read (write). This property is expressible by

$$N_2 + N_8 > 0 \quad \Rightarrow \quad \diamondsuit(N_6 + N_{13} > 0).$$

We prove this property by two lemmas, concentrating first on the *writers*.

Lemma A $\quad N_8 > 0 \quad \Rightarrow \quad \diamondsuit(N_6 + N_{13} > 0)$

This lemma states that if a process is interested in writing, then eventually some process will either read (visit ℓ_6) or write (visit ℓ_{13}).

The proof of the lemma is established by a sequence of simpler lemmas, identifying important intermediate stages in getting from $N_8 > 0$ to $N_6 + N_{13} > 0$. We refer the reader to Fig. 2.27 which presents a diagram showing the structure of the proof.

Lemma A1 $\quad N_8 > 0 \quad \Rightarrow \quad \diamondsuit(N_{10} > 0)$

This lemma ensures that if some process is currently at ℓ_8 then eventually some process will arrive at ℓ_{10}.

This simple property is a direct consequence of the property

$$at\_\ell_8[i] \quad \Rightarrow \quad \diamondsuit\, at\_\ell_{10}[i],$$

which is established by the RANK diagram of Fig. 2.28.

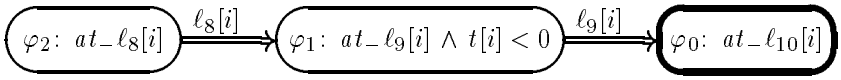


Fig. 2.28. P-RANK diagram for Lemma A1.

Arriving at a state satisfying $N_{10} > 0$, there are two cases to be considered. The easier one is that $y = M$, and Lemma A4 below shows how to get from this situation to the goal $N_6 + N_{13} > 0$. The more complicated situation is when $y < M$ (in view of the invariant I1, always $y \leq M$). The two lemmas, A2 and A3, show that being at $N_{10} > 0 \wedge y < M$ we eventually get to the easier case of $N_{10} > 0 \wedge y = M$. They split the case $N_{10} > 0 \wedge y < M$ into two subcases, according to whether $N_{11..14}$ is positive or zero, and show that from both we eventually arrive to $N_{10} > 0 \wedge y = M$.

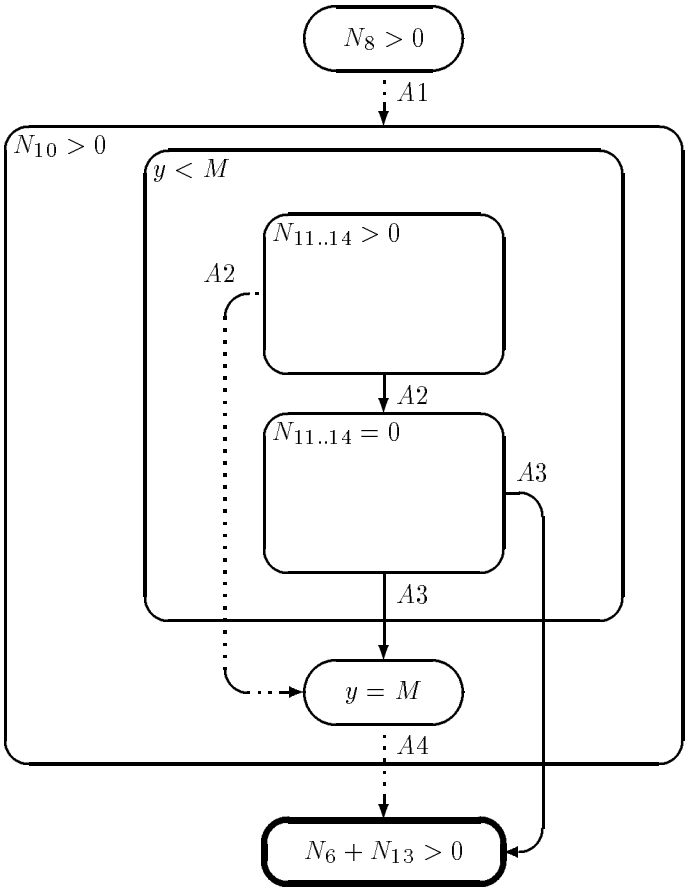


Fig. 2.27. Structure of the proof of Lemma A.

Lemma A2 $N_{10} > 0 \ \wedge \ y < M \ \wedge \ N_{11..14} > 0 \ \Rightarrow$

$$\diamondsuit\big(N_{10} > 0 \ \wedge \ (N_{11..14} = 0 \ \vee \ y = M)\big)$$

The proof of this lemma can be based on rule WELL-JP, and is presented in the P-RANK diagram of Fig. 2.29.

Note that transitions from ℓ_{14} may either retain $N_{11..14} > 0$ and $y < M$, or evacuate $L_{11..14}$ completely, or increase y to become M. It is possible for y to equal M, while some processes are still at $\ell_{10,11}$.

Note that we have to consider also the possibility that a transition ℓ_7 increases y to become M.

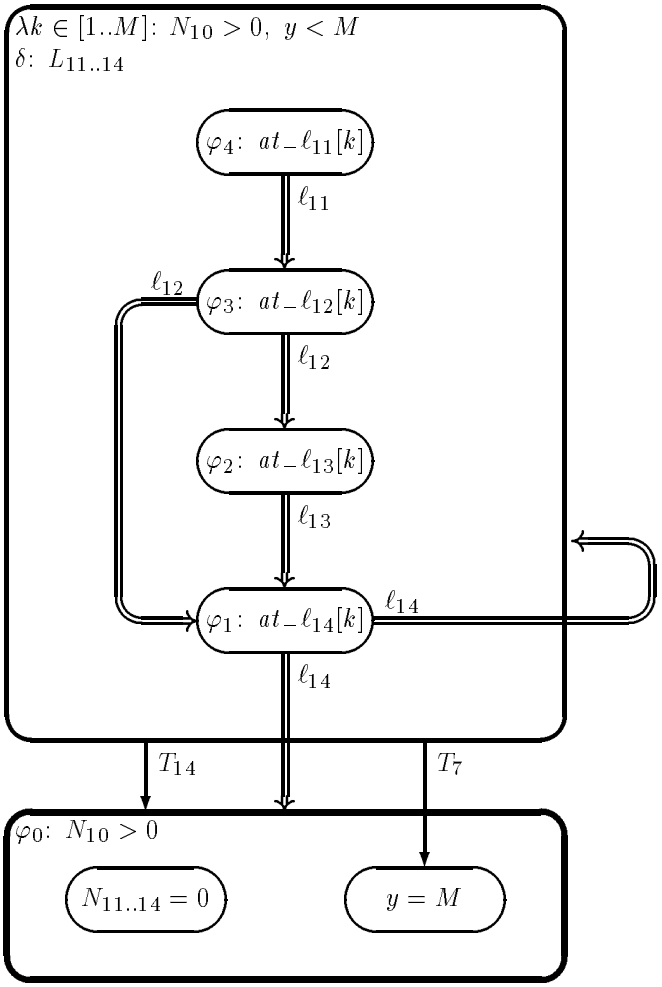


Fig. 2.29. P-RANK diagram for Lemma A2.

In the proof we rely on the fact that while $y < M$, all transitions of the form $\ell_{10}[i]$ are disabled.

Lemma A3 $N_{10} > 0 \ \wedge \ y < M \ \wedge \ N_{11..14} = 0 \ \Rightarrow$

$$\diamondsuit\big((N_{10} > 0 \ \wedge \ y = M) \ \vee \ N_6 > 0\big)$$

The proof of the lemma is presented in the P-RANK diagram of Fig. 2.30.

We observe that, due to I_1, the antecedent implies that $0 < N_{4..7} < M$ and

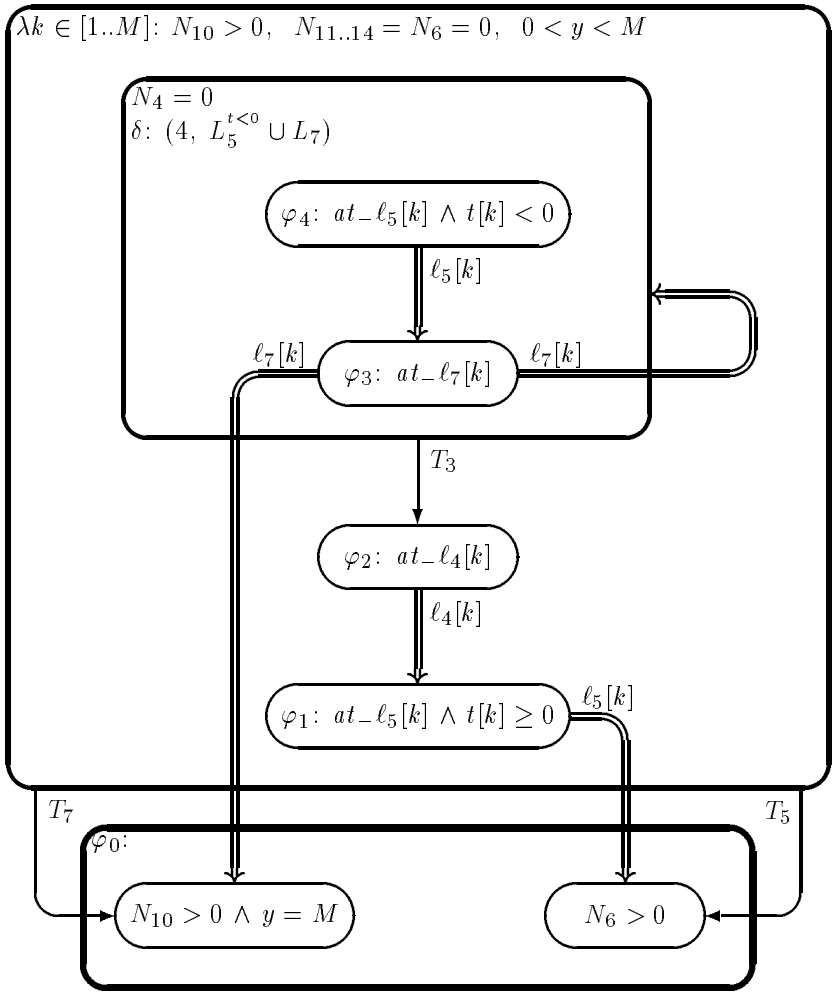


Fig. 2.30. P-RANK diagram for Lemma A3.

$0 < y < M$. This situation is split in the diagram into several subcases:

- $N_4 = N_6 = 0 \ \wedge \ N_5^{t<0} + N_7 > 0$ is covered by φ_3 and φ_4
- $N_4 > 0$ is covered by φ_2
- $N_5^{t\geq 0} > 0$ is covered by φ_1
- $N_6 > 0$ is the goal φ_0.

The sum $N_5^{t<0} + N_7$ decreases on each ℓ_7 transition. It must eventually drop

to zero, ensuring $y = M$, unless N_4 becomes positive before.

Combining Lemmas A2 and A3 together, we obtain

$$N_{10} > 0 \ \wedge \ y < M \quad \Rightarrow \quad \diamondsuit\big((N_{10} > 0 \ \wedge \ y = M) \ \vee \ N_6 > 0\big).$$

It remains to show that from $(N_{10} > 0) \wedge (y = M)$ we are also guaranteed to reach $N_{13} + N_6 > 0$. This is claimed by the next lemma.

Lemma A4 $\quad N_{10} > 0 \ \wedge \ y = M \quad \Rightarrow \quad \diamondsuit(N_6 + N_{13} > 0)$

The proof of this lemma is presented in the P-RANK diagram of Fig. 2.31.

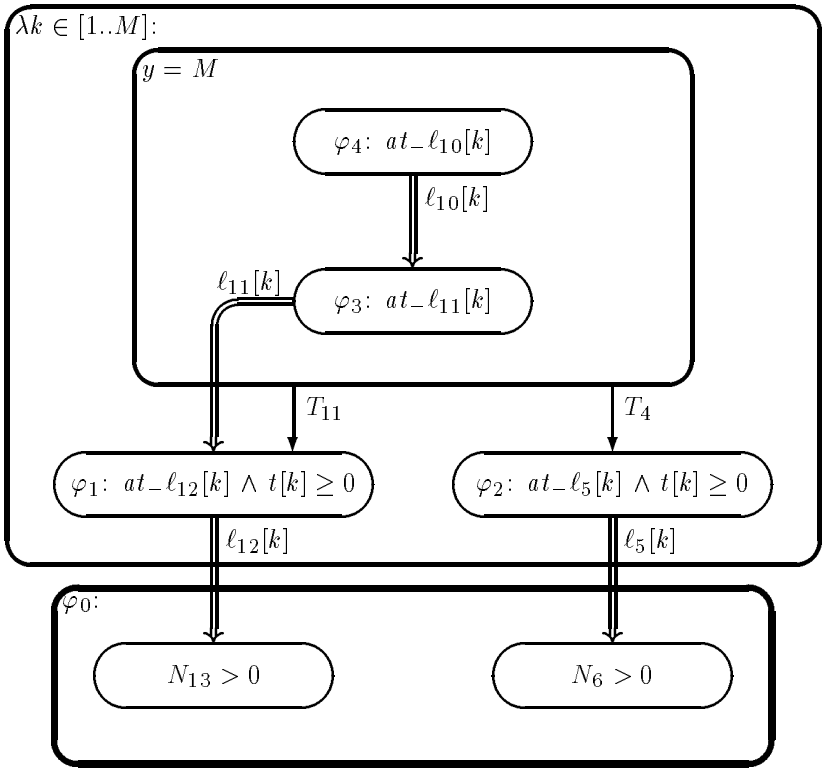


Fig. 2.31. P-RANK diagram for Lemma A4.

This concludes the proof of Lemma A, which states that if a process wishes to write then eventually some process will read or some process will write.

By combining Lemmas A1, A2, and A3, we can also obtain Corollary A

Corollary A $\quad N_{10} > 0 \;\Rightarrow\; \Diamond(N_6 + N_{13} > 0)$

The second case we have to handle is the possibility that a process wishes to read. This is covered by

Lemma B $\quad N_2 > 0 \;\Rightarrow\; \Diamond(N_6 + N_{13} > 0)$

The proof of Lemma B is also split into several intermediate stages. In Fig. 2.32 we present these stages and the lemmas that lead from one stage to the next.

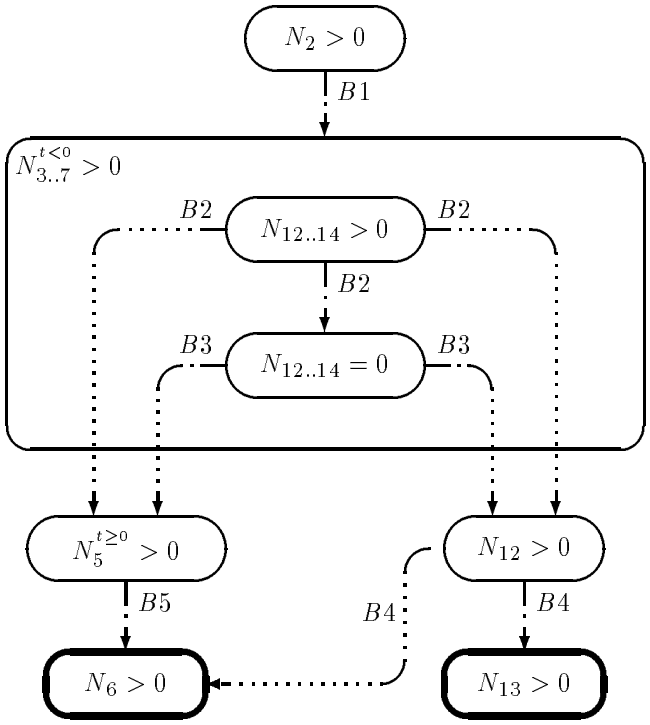


Fig. 2.32. Structure of the proof of Lemma B.

Lemma B1 $\quad N_2 > 0 \;\Rightarrow\; \Diamond\left(N_{3..7}^{t<0} > 0\right)$

This simple lemma follows a process that wishes to read, from location ℓ_2 to location ℓ_3, where it arrives with $t[i] < 0$. The lemma is so simple that we skip its proof.

Arriving at ℓ_3, it is important to distinguish between the case that $y > 0$ and the case $y \leq 0$. Due to the invariant I_1, if $y \leq 0$ then either $N_{12..14} > 0$ or $N_{5..7} = M$. Since $N_3 > 0$, it is impossible for all processes to be at $\ell_{5..7}$, and therefore we conclude that $N_{12..14} > 0$. The next lemma shows that this case must develop to the simpler case in which $N_{12..14} = 0$. However, we cannot hope that while this happens, the process which is currently at ℓ_3 will stand still. What can it do? It can loop around $\ell_{3..7}$ and perhaps even get to ℓ_5 with a nonnegative $t[i]$. If this happens then we are close to our goal $N_6 > 0$. If it does not happen, we should at least make sure that this process does not escape the loop $\ell_{3..7}$ without visiting ℓ_6 first. To contain some processes in the loop we use the set $L_{3..7}^{t<0}$, observing that the only way a process can leave this set, is by moving to $L_5^{t\geq 0}$.

Consequently, the next lemma establishes

Lemma B2

$$N_{3..7}^{t<0} > 0 \ \wedge \ N_{12..14} > 0 \quad \Rightarrow \quad \diamondsuit \left[\begin{array}{l} \left(N_{3..7}^{t<0} > 0 \ \wedge \ N_{12..14} = 0\right) \vee \\ N_5^{t\geq 0} > 0 \ \vee \ N_{12} > 0 \end{array} \right]$$

The proof of this lemma is presented in the P-RANK diagram of Fig. 2.33. It uses the ranking function $N_{12..14}$ which decreases on each ℓ_{14} transition. By I1, as long as $N_{12..14} > 0$, y is smaller than M.

In a more general situation we would have to worry about the possibility of $N_{12..14}$ increasing due to an ℓ_{11} transition. Here, however, any ℓ_{11} transition makes L_{12} nonempty, which is one of the goals of this lemma.

Similarly, the only transition that can violate $N_{3..7}^{t<0} > 0$ is L_4 from a positive y. This, however, makes $L_5^{t\geq 0}$ nonempty, which is another goal.

The next lemma considers a situation in which there is still some unsatisfied process (i.e., $t[i] < 0$) in the $\ell_{3..7}$ loop, while $N_{12..14} = 0$. It shows that eventually one of the sets $L_5^{t\geq 0}$ or L_{12} must become nonempty.

Lemma B3 $\quad N_{3..7}^{t<0} > 0 \ \wedge \ N_{12..14} = 0 \quad \Rightarrow \quad \diamondsuit \left(N_5^{t\geq 0} > 0 \ \vee \ N_{12} > 0 \right)$

The lemma is proved in P-RANK diagram of Fig. 2.34. Note that while being at ℓ_6, $t[i] \geq 0$, so that $L_6^{t<0}$ is always empty. Therefore, having $N_{3..7}^{t<0} > 0$ implies that we must have a nonempty $L_{3..5,7}$. If L_4 is nonempty, then due to $N_{12..14} = 0$ and to I1, we are guaranteed to have a positive y. Therefore, any ℓ_4 transition from such a state, leads to a nonempty $L_5^{t\geq 0}$.

Lemma B4 $\quad N_{12} > 0 \quad \Rightarrow \quad \diamondsuit(N_6 + N_{13} > 0)$

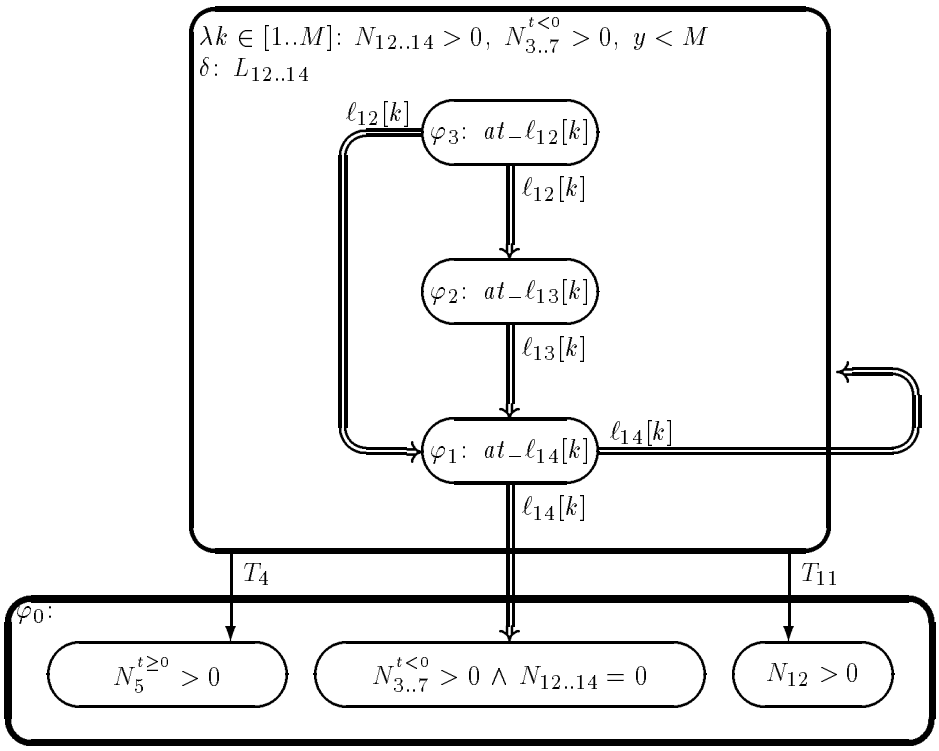


Fig. 2.33. P-RANK diagram for Lemma B2.

This lemma starts from a situation in which we identify a writer at ℓ_{12}. If $t[i] \geq 0$ for this process, then we can easily prove that eventually $N_{13} > 0$. Otherwise, $t[i] < 0$, and we can easily trace the progress of this process from ℓ_{12}, through ℓ_{14}, ℓ_9 until it reaches ℓ_{10}. We now use Lemma A to show that if L_{10} is nonempty, then eventually some process will reach ℓ_6 or ℓ_{13}.

Lemma B5 $\quad N_5^{t\geq 0} > 0 \quad \Rightarrow \quad \diamondsuit(N_6 > 0)$

This lemma requires one helpful step, performed by any of the processes in $L_5^{t\geq 0}$, to achieve a nonempty L_6.

Problems

Problem 2.1 (mutual exclusion by turn setting) page 105

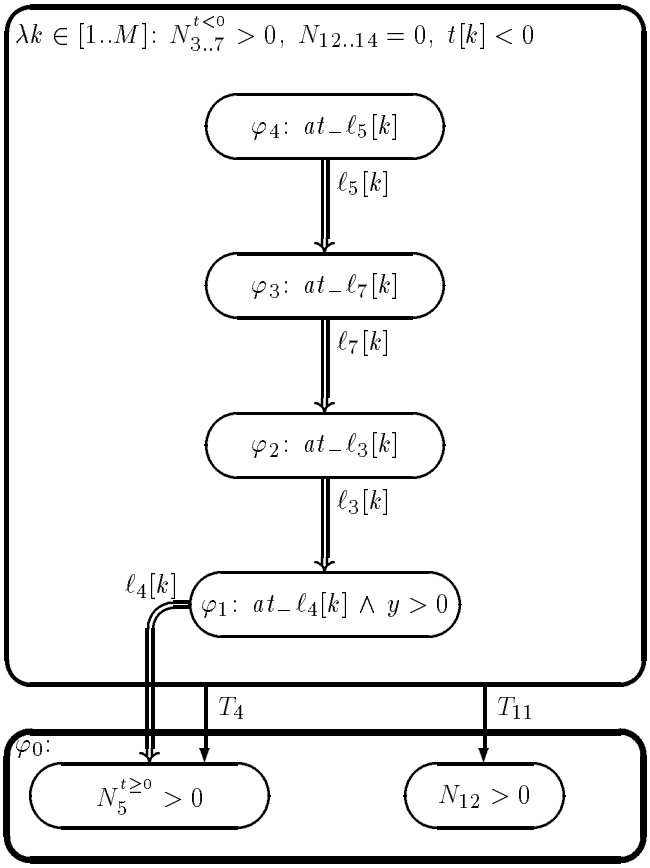


Fig. 2.34. P-RANK diagram for Lemma B3.

Show that the property of (individual) accessibility, specified by

$$at\_\ell_2[i] \ \wedge \ t = 0 \quad \Rightarrow \quad \diamondsuit\, at\_\ell_5[i]$$

is not guaranteed for program TURN (Fig. 2.8).

Problem 2.2 (mutual exclusion with central manager) page 105

Program PMUX-MAN of Fig. 2.35 implements mutual exclusion, using a central allocator process A which receives requests in variable y and responds in variable x. Using the parameterized rules presented in this chapter, prove accessibility for this program, which can be specified by the response formula

$$N_2 > 0 \ \Rightarrow \ \diamondsuit(N_4 > 0),$$

where N_2 counts the number of processes $P[j]$ currently executing at location m_2, while N_4 counts the number of processes $P[j]$ currently executing at location m_4

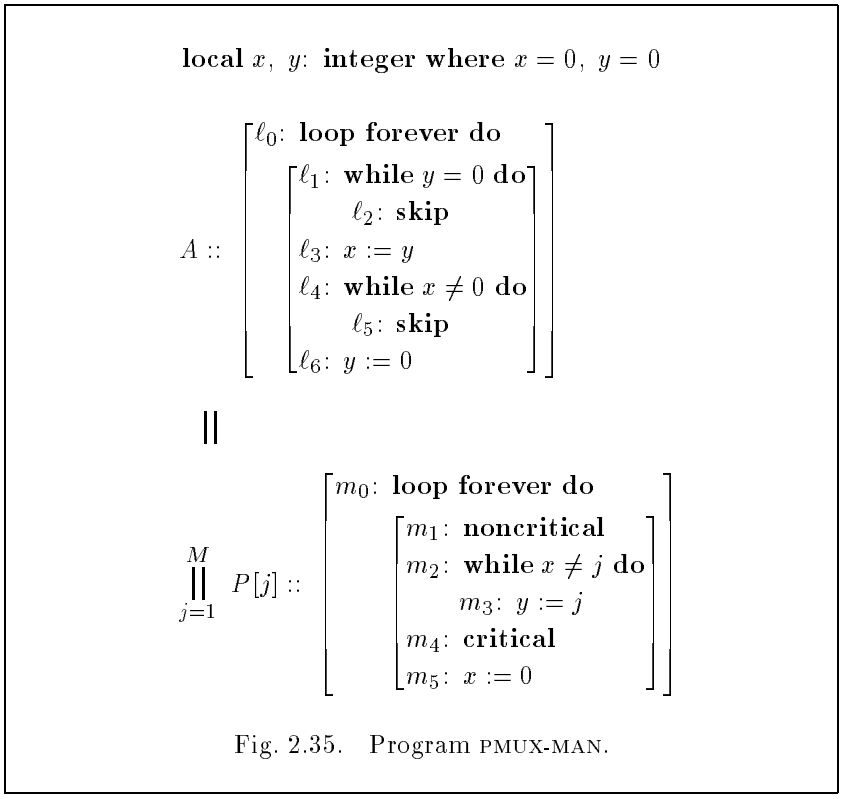


Fig. 2.35. Program PMUX-MAN.

Note that this program cannot guarantee individual accessibility but only communal accessibility (see page 104).

Problem 2.3 (molecular bakery algorithms) page 117

In Fig. 2.17, we presented the atomic version of the bakery algorithm and verified its properties.

This atomic version is obviously not an LCR-program and represents a very simplified version which cannot be directly implemented on existing hardware.

A first attempt to transform program BAKERY-AT into an LCR program breaks each of the non-LCR statements ℓ_2 and ℓ_3 into a *for*-loop. This leads to program BAKERY-1, presented in Fig. 2.36.

While program BAKERY-1 obeys the LCR restriction, it is no longer a correct solution to the mutual exclusion problem.

$$
\begin{array}{l}
\textbf{in } M \quad : \textbf{integer where } M > 1 \\
\quad \mathit{ticket}\text{: } \textbf{array } [1..M] \textbf{ of integer where } \mathit{ticket} = 0
\end{array}
$$

$$
\prod_{i=1}^{M} P[i] :: \left[\begin{array}{l}
j,\ k,\ t\text{: } \textbf{integer where } j = k = t = 0 \\
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } t := 0 \\
\ell_3\text{: } \textbf{for } k = 1 \textbf{ to } M \textbf{ do} \\
\quad \ell_4\text{: } t := \max(t,\ \mathit{ticket}[k]) \\
\ell_5\text{: } \mathit{ticket}[i] := t + 1 \\
\ell_6\text{: } \textbf{for } j = 1 \textbf{ to } M \textbf{ do} \\
\quad \left[\ell_7\text{: } \textbf{await } \left(\begin{array}{l} j = i \\ \vee\ \mathit{ticket}[j] = 0 \\ \vee\ \mathit{ticket}[i] < \mathit{ticket}[j] \end{array}\right)\right] \\
\ell_8\text{: } \textbf{critical} \\
\ell_9\text{: } \mathit{ticket}[i] := 0
\end{array}\right]
\end{array}\right]
$$

Fig. 2.36. Program BAKERY-1: an LCR version.

(a) Present a segment of a computation of program BAKERY-1 which violates mutual exclusion.

As can be seen from your counter-example, the fault can be traced to the fact that while process $P[i]$ is computing its next *ticket* value at locations ℓ_3–ℓ_5, the value of $\mathit{ticket}[i]$ is unreliable and should not be checked by other processes at statement ℓ_7.

To correct this problem, we introduce a protection mechanism, by which $\mathit{ticket}[i]$ cannot be accessed while $P[i]$ is at the location range ℓ_3–ℓ_5.

This leads to program BAKERY-2 of Fig. 2.37.

The protection mechanism implemented in BAKERY-2, indeed solves the problem of mutual-exclusion violation.

(b) Prove that program BAKERY-2 ensures mutual exclusion.

While program BAKERY-2 guarantees mutual exclusion, it fails to maintain the complementary required property of accessibility.

(c) Show a computation of program BAKERY-2 which fails to satisfy the property

$$
\begin{array}{l}
\textbf{in } M \quad\quad : \textbf{integer where } M > 1 \\
\quad \mathit{choosing}\colon \textbf{array } [1..M] \textbf{ of boolean where } \mathit{choosing} = \text{F} \\
\quad \mathit{ticket} \quad : \textbf{array } [1..M] \textbf{ of integer where } \mathit{ticket} = 0
\end{array}
$$

$$
\Big\|_{i=1}^{M} P[i] :: \left[\begin{array}{l}
j,\ k,\ t\colon \textbf{integer where } j = k = t = 0 \\
\ell_0\colon \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\colon \textbf{noncritical} \\
\ell_2\colon\ \mathit{choosing}[i] := \text{T} \\
\ell_3\colon\ t := 0 \\
\ell_4\colon \textbf{for } k = 1 \textbf{ to } M \textbf{ do} \\
\quad \ell_5\colon\ t := \max\big(t,\ \mathit{ticket}[k]\big) \\
\ell_6\colon\ \mathit{ticket}[i] := t + 1 \\
\ell_7\colon\ \mathit{choosing}[i] := \text{F} \\
\ell_8\colon \textbf{for } j = 1 \textbf{ to } M \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_9\colon \textbf{await } \neg \mathit{choosing}[j] \\
\ell_{10}\colon \textbf{await } \left(\begin{array}{l} j = i \\ \vee\ \mathit{ticket}[j] = 0 \\ \vee\ \mathit{ticket}[i] < \mathit{ticket}[j] \end{array}\right)
\end{array}\right] \\
\ell_{11}\colon \textbf{critical} \\
\ell_{12}\colon\ \mathit{ticket}[i] := 0
\end{array}\right]
\end{array}\right]
$$

Fig. 2.37. Program BAKERY-2: ticket computation is protected.

of accessibility for one of the processes.

To overcome the difficulty illustrated by your counter-example, we augment the priority comparison mechanism of statement ℓ_7 by a secondary component, which is the process index. Thus, if both $\mathit{ticket}[i]$ and $\mathit{ticket}[j]$ are different from zero, then $P[i]$ has a higher priority if

$$\big(\mathit{ticket}[i],\ i\big) \prec \big(\mathit{ticket}[j],\ j\big),$$

where the comparison is lexicographic.

This leads to program BAKERY of Fig. 2.38.

This version is fianlly satisfactory in that it satisfies the properties of mutual exclusion and accessibility.

(d) Prove that program BAKERY satisfies the property of mutual exclusion.

$$
\begin{array}{l}
\textbf{in } M \quad : \textbf{integer where } M > 1 \\
\quad \mathit{choosing}\colon \textbf{array } [1..M] \textbf{ of boolean where } \mathit{choosing} = \text{F} \\
\quad \mathit{ticket} \quad : \textbf{array } [1..M] \textbf{ of integer where } \mathit{ticket} = 0
\end{array}
$$

$$
\begin{array}{l}
\Big\|_{i=1}^{M} P[i] :: \\
\quad \left[\begin{array}{l}
j,\ k,\ t\colon \textbf{integer where } j = k = t = 0 \\
\ell_0\colon \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\colon \textbf{noncritical} \\
\ell_2\colon \mathit{choosing}[i] := \text{T} \\
\ell_3\colon t := 0 \\
\ell_4\colon \textbf{for } k = 1 \textbf{ to } M \textbf{ do} \\
\quad \ell_5\colon t := \max(t,\ \mathit{ticket}[k]) \\
\ell_6\colon \mathit{ticket}[i] := t + 1 \\
\ell_7\colon \mathit{choosing}[i] := \text{F} \\
\ell_8\colon \textbf{for } j = 1 \textbf{ to } M \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_9\colon \textbf{await } \neg \mathit{choosing}[j] \\
\ell_{10}\colon \textbf{await } \left(\begin{array}{l} j = i \\ \vee\ \mathit{ticket}[j] = 0 \\ \vee\ \mathit{ticket}[i] \prec \mathit{ticket}[j] \end{array}\right)
\end{array}\right] \\
\ell_{11}\colon \textbf{critical} \\
\ell_{12}\colon \mathit{ticket}[i] := 0
\end{array}\right]
\end{array}\right]
\end{array}
$$

Fig. 2.38. Program BAKERY: Final version.

(e) Prove that program BAKERY satisfies the property of accessibility.

∗ **Problem 2.4** (Peterson's algorithm for n processes) page 117

Program MUX-PET-N of Fig. 2.39 presents a solution to the mutual-exclusion problem for an arbitrary number of processes.

The *while* statement at ℓ_3 gradually increases the priority of process $P[i]$ from 1 to n. When the priority, expressed by j ($j[i]$ for process $P[i]$), grows beyond n, $P[i]$ is admitted to its critical section. On entering priority level j, process $P[i]$ records (at ℓ_4) its current priority in $y[i]$ and writes its identity number (i) in $s[j]$ the signature logbook for this level. It then remains at level j until it detects one of two occurrences. Either the number of processes of priority not smaller than j becomes 1, which means that $P[i]$ is the only one with such priority, or the logbook $s[j]$ assumes a value different than i, implying that some of the processes

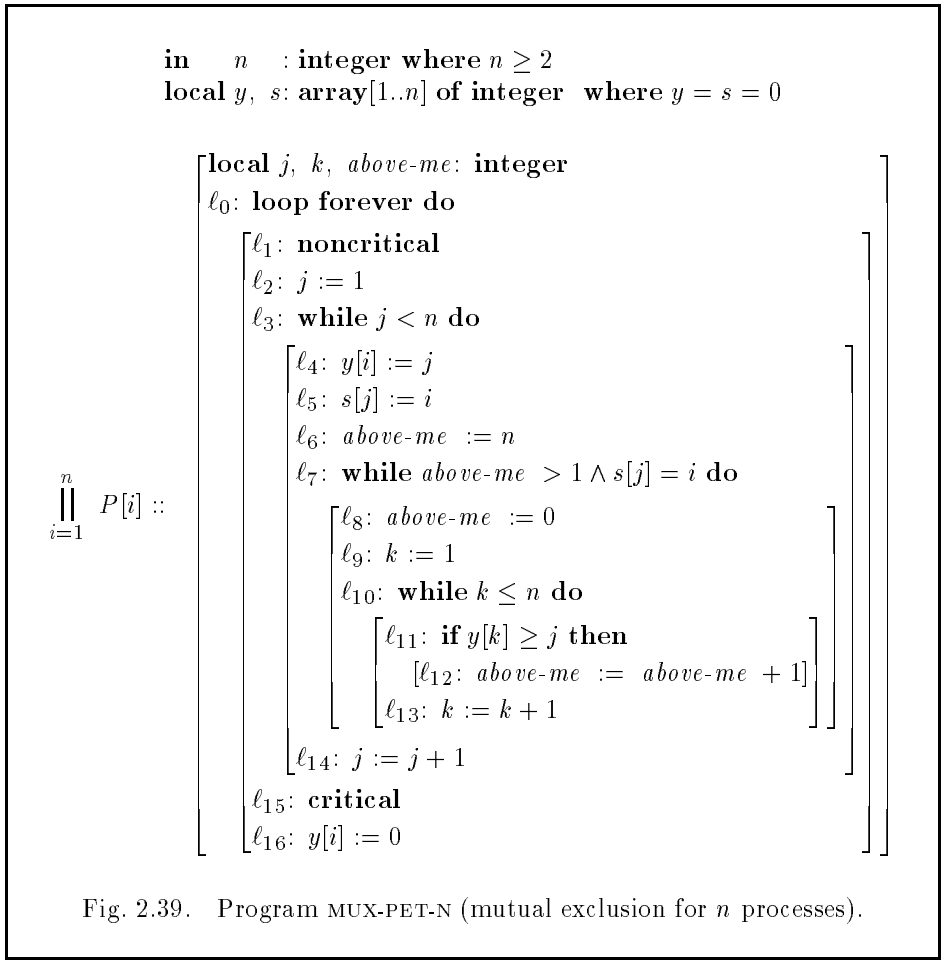


Fig. 2.39. Program MUX-PET-N (mutual exclusion for n processes).

entered level j *after* $P[i]$.

To check for the first occurrence, statements ℓ_8 to ℓ_{13} repeatedly count the number of processes with priority not smaller than j. Note that one cannot assume that on termination of the *while* statement ℓ_{10} the value of *above-me* necessarily equals the number of processes with priority not smaller than j. While $P[i]$ counts, some processes with high priority may go through the critical section and lower their priority, and some processes with low priority may raise their priority to the tested level after being considered.

Prove that program MUX-PET-N guarantees accessibility to each of its processes, by showing that the response formula

$$at\_\ell_2[i] \;\Rightarrow\; \diamondsuit\, at\_\ell_{15}[i]$$

is P-valid over MUX-PET-N.