A Temporal Proof Methodology for Reactive Systems *†

Zohar Manna
Stanford University‡

Amir Pnueli
Weizmann Institute of Science§

September 8, 1995

Abstract. The paper presents a minimal proof theory which is adequate for proving the main important temporal properties of reactive programs. The properties we consider consist of the classes of *invariance*, *response*, and *precedence* properties. For each of these classes we present a small set of rules that is complete for verifying properties belonging to this class. We illustrate the application of these rules on several examples. We discuss concise presentations of complex proofs using the devices of *transition tables* and *proof diagrams*.

Contents

---

*This research was supported in part by the National Science Foundation under grant CCR-89-11512; by the Defense Advanced Research Projects Agency under contract NAG2-703, by the United States Air Force Office of Scientific Research under contracts AFOSR-90-0057, and by the European Community ESPRIT Basic Research Action project 6021 (React).

†A preliminary and abbreviated version of this paper appeared in [MP90].

‡Department of Computer Science, Stanford University, Stanford, CA 94305.
E-mail: `manna@cs.stanford.edu`

§Department of Computer Science, Weizmann Institute, Rehovot, Israel.
E-mail: `amir@wisdom.weizmann.ac.il`

1 Introduction

In this paper we present a minimal proof theory that is adequate for proving interesting properties of reactive systems. *Reactive systems* are systems (and programs) whose main role is to maintain an ongoing interaction with their environment, rather than to produce some final result on termination. Such systems should be specified and analyzed in terms of their behaviors, i.e., the sequences of states or events they generate during their operation. The class of reactive systems includes programs such as operating systems, programs controlling industrial plants, embedded systems, and many others. It is clear that it also includes the classes of concurrent and distributed programs since, independent of the goal and purpose of the complete system, each component of the system has to be studied in terms of the interaction it maintains with the other components.

A reactive program may be viewed as a generator of *computations* which, for simplicity, we may assume to be infinite sequences of states or events. In the case that the program does terminate, we may always extend the finite computation it has generated by an infinite sequence of duplicate states or dummy events to obtain an infinite computation.

An important approach to the specification and verification of reactive systems is based on specifying a program by listing several properties, representing *requirements* that the program ought to satisfy. This approach enjoys the advantages of abstraction and modularity.

By *abstraction* we mean that, since the specifier lists separate properties and is not required to show how they can be integrated or to worry about how they may interact with one another, he is not tempted to overspecify or actually design the system. Consequently, this approach leads to specifications which are free of implementation bias.

By *modularity* we mean that a property-list based specification is very easy to modify by dropping, adding or modifying a single property. Also, the process of verifying that a proposed implementation satisfies its specification can be done in a modular fashion, by verifying each property separately.

Several formal approaches have been proposed over the years for expressing and verifying properties of programs, including the language of temporal logic [Pnu77, Lam83] and the formalism of predicate automata [AS89, MP87]. The theoretical investigations into the questions of the expressibility of the specification language and the completeness of the proof theory associated with these formal approaches grew into a large body of knowledge. This may create the false impression that all that body of knowledge is essential for the application of the methodology, and that a heavy investment in learning all this theoretical material is necessary.

One of the points we would like to demonstrate in this paper is that a very little general (temporal) theory is required to handle the most important properties of concurrent programs that occur in practice. The types of properties on which a *practicing verifier* typically spends the most time usually fall into a few simple classes. By presenting a simple but complete set of rules for verifying properties belonging to each of these classes, we provide the practicing verifier with precisely the tools that are needed. Consequently, the approach we take in this paper is to circumvent the general theory of temporal logic and proceed as directly as possible to the introduction of the classes of properties that are most frequently verified and to the proof rules that are appropriate for their verification. We consider three classes of properties, which we believe to cover most of the properties one would ever wish to verify for a reactive program.

To express the properties of programs, we use a specification language, whose building blocks are *state formulas* (also called *assertions*). These are first-order formulas which describe program states that can arise in a computation.

The three classes we consider are:

- *Invariance* – An invariance property refers to an assertion p, and requires that p is an invariant over all the computations of a program P, i.e., all the states arising in a computation of P satisfy p. In temporal logic notation, such properties are expressed by $\Box p$, for a state formula p.

- *Response* – A response property refers to two assertions p and q, and requires that every p-state (a state satisfying p) arising in a computation is eventually followed by a q-state. In temporal logic notation this is written as $p \Rightarrow \Diamond q$.

- *Precedence* – A simple precedence property refers to three assertions p, q, and r. It requires that any p-state initiates a q-interval (i.e., an interval all of whose states satisfy q) which, either extends to the end of the computation, or is terminated by an r-state. Such a property is useful for expressing the requirement that, following a certain condition p, event r will precede event q.

  In temporal logic, this property is expressed by $p \Rightarrow (\neg q)\,\mathcal{W} r$, using the *waiting-for* operator (weak *until*) $\mathcal{W}$. More complex precedence properties refer to a sequence of assertions $q_0, \ldots, q_{m-1}$, and replace the requirement of a single q-interval by requiring a q_0-interval, followed by a q_1-interval, $\ldots$, followed by a q_{m-1}-interval.

We refer the reader to [MP91a] for a top-down approach, which presents the most general proof rules possible. Here, however, we take the opposite approach of presenting rules that are closely tailored for these restricted classes.

2 Programs and Computations

The basic computational model we use to represent programs is that of a *fair transition system*. In this model, a program P consists of the following components.

- $V = \{u_1, ..., u_n\}$ – A finite set of *state variables*. Some of these variables represent *data* variables, which are explicitly manipulated by the program text. Other variables are *control* variables, which represent, for example, the location of control in each of the processes in a concurrent program. We assume each variable to be associated with a nonempty domain over which it ranges.

  We define a *state* s to be a type consistent interpretation of V, assigning to each variable $u \in V$ a value $s[u]$ over its domain. We denote by Σ the set of all states.

- Θ – The *initial condition*. This is a satisfiable assertion characterizing all the initial states, i.e., states at which the computation of the program can start. A state is defined to be *initial* if it satisfies Θ.

- $\mathcal{T}$ – A set of *transitions*. Each transition $\tau \in \mathcal{T}$ is associated with an assertion $\rho_\tau(V, V')$, called the *transition relation*, which may refer to both unprimed and primed versions of the state variables. The purpose of the transition relation ρ_τ is to express a relation between a state s and its successor s'. We use the unprimed version to refer to values in s, and the primed version to refer to values in s'. For example, the assertion $x' = x + 1$ states that the value of x in s' is greater by 1 than its value in s.

- $\mathcal{J} \subseteq \mathcal{T}$: A set of *just* transitions (also called *weakly fair* transitions). Intuitively, the requirement of justice for $\tau \in \mathcal{J}$ disallows a computation in which τ is continually enabled but not taken beyond a certain point.

We define the state s' to be a τ*-successor* of the state s if the assertion $\rho_\tau(V, V')$ is satisfied by $\langle s, s' \rangle$, the joint interpretation which interprets $x \in V$ as $s[x]$, and interprets x' as $s'[x]$. Following this definition, we can view the transition τ as a function $\tau : \Sigma \mapsto 2^\Sigma$, defined by:

$$\tau(s) = \{s' \mid s' \text{ is a } \tau\text{-successor of } s\}.$$

We say that the transition τ is *enabled* on the state s if $\tau(s) \neq \phi$. Otherwise, we say that τ is *disabled* on s. The enabledness of a transition τ can be expressed by the formula

$$En(\tau) : \ (\exists V')\rho_\tau(V, V'),$$

which is true in s iff s has some τ-successor.

We require that every state $s \in \Sigma$ has at least one transition enabled on it. This is often ensured by including in $\mathcal{T}$ the *idling* transition τ_I (also called the *stuttering* transition), whose transition relation is $\rho_I : (V = V')$. Thus, s' is a τ_I-successor of s iff $s' = s$.

Assume a program P for which the above components have been specified. Consider

$$\sigma : s_0, s_1, s_2, \ldots,$$

an infinite sequence of states of P. We say that transition $\tau \in \mathcal{T}$ is *enabled at position* k of σ if τ is enabled on s_k. We say that the transition τ is *taken at position* k if s_{k+1} is a τ-successor of s_k. Note that several different transitions can be considered as taken at the same position.

The sequence σ is defined to be a *computation* of P if it satisfies the following requirements:

- *Initiality*: s_0 is initial.

- *Consecution*: For each $j = 0, 1, \ldots$, the state s_{j+1} is a τ-successor of the state s_j, i.e., $s_{j+1} \in \tau(s_j)$, for some $\tau \in \mathcal{T}$.

- *Justice*: For each transition $\tau \in \mathcal{J}$, it is not the case that τ is continually enabled beyond some position j in σ, i.e., τ is enabled at every position $k \geq j$, while τ is not taken beyond j.

We say that a state s is *P-accessible* if it appears in some computation of P. Clearly, any τ-successor of a P-accessible state is also P-accessible.

We refer the reader to [MP91b] for a more comprehensive notion of a fair transition system that specifies also a set of *compassionate* (strongly fair) transitions. The requirement of compassion is relevant only for programs that use special synchronization constructs such as semaphores or message passing statements. In the examples presented here concurrent processes communicate by shared variables, so there is no need for the compassion component.

We assume an underlying assertional language, which contains the predicate calculus and interpreted symbols for expressing the standard operations and relations over some concrete domains. We refer to a formula in the assertional language as an *assertion*.

For an assertion p and a state s such that p holds on s, we say that s is a *p-state*. For a computation $\sigma : s_0, s_1, \ldots$, such that s_j is a p-state, we call j a *p-position*.

3 The Main Examples: Mutual Exclusion

For our main examples we use two programs that have been proposed as solutions to the mutual exclusion problem.

The simple version of the *mutual exclusion* problem considers two processes that need to coordinate access to a shared resource. This shared resource may represent a shared variable or a device, such as a disk or printer, that needs to be accessed exclusively, i.e., protected from interference.

Solutions to the mutual exclusion problem are presented by programs that contain two concurrent processes. Each process contains two schematic statements: statement *Non-Critical* and statement *Critical*. Statement *Non-Critical* represents the independent activity of the process. It stands for an arbitrary complex segment of the program that represents all the processing that requires no coordination with the other process. It is not even required that this statement terminates. Nontermination of the non-critical statement corresponds to the situation in which a certain process needs no further access to the shared resource.

Statment *Critical* (usually referred to as the *critical statement* or *critical section*) represents all the activity that has to be performed in protected mode. For this activity, we require eventual termination. Nontermination of the critical statement corresponds to one process appropriating the shared resource and never releasing it to the other process. This is, in general, an unacceptable behavior.

An important assumption about both of these schematic statements is that they do not modify any of the variables that are used in the protocol for coordination between the two processes.

We present two solutions to the mutual exclusion problem.

$$
\begin{array}{lll}
\textbf{local} & y_1, y_2 & : \textbf{boolean where } y_1 = \mathrm{F},\ y_2 = \mathrm{F} \\
 & s & : \textbf{integer where } s = 1
\end{array}
$$

$$
\left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\left[\begin{array}{ll}
\ell_1 : & \text{Non-Critical} \\
\ell_2 : & y_1 \ := \ \mathrm{T} \\
\ell_3 : & s \ := \ 1 \\
\ell_4 : & \textbf{await } (\neg y_2) \vee (s = 2) \\
\ell_5 : & \text{Critical} \\
\ell_6 : & y_1 \ := \ \mathrm{F}
\end{array}\right]
\end{array}\right]
\quad || \quad
\left[\begin{array}{l}
m_0 : \textbf{loop forever do} \\
\left[\begin{array}{ll}
m_1 : & \text{Non-Critical} \\
m_2 : & y_2 \ := \ \mathrm{T} \\
m_3 : & s \ := \ 2 \\
m_4 : & \textbf{await } (\neg y_1) \vee (s = 1) \\
m_5 : & \text{Critical} \\
m_6 : & y_2 \ := \ \mathrm{F}
\end{array}\right]
\end{array}\right]
$$

$$
- P_1 - \qquad\qquad\qquad - P_2 -
$$

Figure 1: Program PET: Peterson's algorithm for mutual exclusion.

3.1 Peterson's Program

Peterson's solution to the mutual exclusion problem [Pet83] is presented in Fig. 1. The basic mechanism protecting the critical sections is provided by the boolean variables y_1 and y_2. Each process P_i, $i = 1, 2$, that is interested in entering its critical section sets its y_i variable to T. On exiting the critical section, the corresponding y_i is reset to F.

The problem with this simple-minded approach is that the two processes may arrive at their *waiting positions*, ℓ_4 and m_4 respectively, at about the same time, with both $y_1 = y_2 = \mathrm{T}$. If the only criterion for entry to the critical section was that the y_i of the competitor be false, this situation would result in a deadlock (tie).

The variable s, ranging over $\{1, 2\}$, is intended for breaking such ties. It may be viewed as a *signature*, in the sense that each process that sets its y_i variable to T also writes its identity number in s at the next statement. Then, if both processes are at the waiting position, the first to enter will be P_i such that $s \neq i$. For $i = 1, 2$, let j denote the index of the other process. The fact that $s = j$ means that the competitor, P_j, was the *last* to reach the waiting position and therefore P_i should have priority.

3.2 Dekker's Program

Another program we study is Dekker's algorithm for mutual exclusion [Dij65]. This was one of the earliest correct solutions (possibly the first) to this problem.

Similar to Peterson's algorithm, each of the processes in Dekker's solution, also uses a boolean variable $y_i, i = 1, 2$, that expresses the interest of the process to enter its critical section. Process P_i starts by setting its y_i variable to T. It then tests the y_i value of its competitor. If the competing y_i is found to equal F, P_i enters its critical section immediately. In case of a tie, i.e., both processes have $y_i = \mathrm{T}$, we use a tie-breaker, the variable t (short for *turn*). This variable ranges over $\{1, 2\}$, and the process whose number is t has the higher priority. To ensure fair accessibility, process P_i sets variable t to the value corresponding to its rival on exit from the critical section.

Dekker's algorithm is presented in Fig. 2.

Let us follow P_1 on exit from the non-critical section. This is where the protocol of

$$
\begin{array}{lll}
\textbf{local} & y_1, y_2 & : \textbf{boolean where } y_1 = \textsc{f},\ y_2 = \textsc{f} \\
 & t & : \textbf{integer where } t = 1
\end{array}
$$

$$
\left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\left[\begin{array}{ll}
\ell_1 : & \text{Non-Critical} \\
\ell_2 : & y_1 \ := \ \textsc{t} \\
\ell_3 : & \textbf{while } y_2 \textbf{ do} \\
 & \quad \ell_4 : \ \textbf{if } (t = 2) \textbf{ then} \\
 & \quad \left[\begin{array}{ll}
 \ell_5 : & y_1 \ := \ \textsc{f} \\
 \ell_6 : & \textbf{await } (t = 1) \\
 \ell_7 : & y_1 \ := \ \textsc{t}
 \end{array}\right] \\
\ell_8 : & \text{Critical} \\
\ell_9 : & t \ := \ 2 \\
\ell_{10} : & y_1 \ := \ \textsc{f}
\end{array}\right]
\end{array}\right]
\quad \| \quad
\left[\begin{array}{l}
m_0 : \textbf{loop forever do} \\
\left[\begin{array}{ll}
m_1 : & \text{Non-Critical} \\
m_2 : & y_2 \ := \ \textsc{t} \\
m_3 : & \textbf{while } y_1 \textbf{ do} \\
 & \quad m_4 : \ \textbf{if } (t = 1) \textbf{ then} \\
 & \quad \left[\begin{array}{ll}
 m_5 : & y_2 \ := \ \textsc{f} \\
 m_6 : & \textbf{await } (t = 2) \\
 m_7 : & y_2 \ := \ \textsc{t}
 \end{array}\right] \\
m_8 : & \text{Critical} \\
m_9 : & t \ := \ 1 \\
m_{10} : & y_2 \ := \ \textsc{f}
\end{array}\right]
\end{array}\right]
$$

$$
- P_1 - \qquad\qquad\qquad\qquad - P_2 -
$$

Figure 2: Program DEKKER: Dekker's algorithm for mutual exclusion.

coordination between the two processes starts. Process P_1 sets at ℓ_2 its y_1 variable to $\textsc{t}$. It then enters a while loop that continues as long as P_1 detects $y_2 = \textsc{t}$. Process P_1 identifies this situation as a tie. Tie breaking is accomplished by one of the processes recognizing it has a lower priority, resetting its y_i variable to $\textsc{f}$, and then waiting for its priority to rise. This happens for P_1 in ℓ_5–ℓ_7. On the other hand, if P_1 recognizes it has a higher priority, it waits for y_2 to become false. This happens for P_1 in the tight loop consisting of ℓ_3 and ℓ_4 where, due to $t = 1$, it never enters the region ℓ_5–ℓ_7. Process P_1 enters its critical section at ℓ_8 only when it detects $y_2 = \textsc{f}$. After termination of the critical section, P_1 first sets t to 2 and then resets y_1 to $\textsc{f}$.

4 A Program as a Fair Transition System

Let us consider how program DEKKER (presented in Fig. 2) can be viewed as a fair transition system.

Below, we identify the four components of a fair transition system, namely, state variables, transitions, initial condition, and justice set, for program DEKKER. This enables us to view the program as a fair transition system, and to apply to it the verification methods that will later be presented for general fair transition systems.

State Variables

The state variables V are given by

$$\pi,\ y_1,\ y_2,\ t.$$

Variable π is a control variable that ranges over sets of program locations. At any state of a computation of DEKKER, $\pi = \{\ell_i, m_j\}$, for $i, j \in \{0, ..., 10\}$, whenever process P_1 is currently in front of the statement labeled ℓ_i and P_2 is currently in front of the statement labeled m_j.

Variables y_1, y_2, t naturally represent the current values of the corresponding program variables.

Initial Condition

The initial condition Θ is given by the assertion

$$\Theta \;:\; (\pi = \{\ell_0, m_0\}) \;\wedge\; \neg y_1 \;\wedge\; \neg y_2 \;\wedge\; (t = 1)$$

Thus, at the initial state of the program, the two processes reside at their initial locations ℓ_0, m_0, the two boolean variables y_1, y_2 are initialized to $\mathbf{F}$, and t is initialized to 1.

We introduce several abbreviations for referring to the location of control.

$$\begin{aligned} at\_\ell_i &: \pi \cap \{\ell_0, \ldots, \ell_{10}\} = \{\ell_i\} \\ at\_m_i &: \pi \cap \{m_0, \ldots, m_{10}\} = \{m_i\} \end{aligned}$$

Thus, at_ℓ_i implies $\ell_i \in \pi$ and also that $\ell_j \notin \pi$ for any $j \neq i$. Similar implications hold for at_m_i.

To express the movement of control effected by transitions, we use the following abbreviations:

$$\begin{aligned} move(\ell_i, \ell_j) &: at\_\ell_i \;\wedge\; (\pi' = \pi - \{\ell_i\} \cup \{\ell_j\}) \\ move(m_i, m_j) &: at\_m_i \;\wedge\; (\pi' = \pi - \{m_i\} \cup \{m_j\}) \end{aligned}$$

Clearly, $move(\ell_i, \ell_j)$ describes the movement of process P_1 from ℓ_i to ℓ_j.

Transitions

In order to avoid tedious repetition, we will present only some of the transitions for process P_1. We will concentrate on the transitions that correspond to the different types of statements appearing in the considered program. We refer the reader to [MP91b] for a fuller account of the transitions corresponding to the different types of statements.

In defining the transition relation ρ_τ corresponding to the transition τ, we adopt the convention by which a variable whose primed version does not appear explicitly in the formula is preserved by the transition. Thus, if y is a state variable and y' does not appear in ρ_τ, the clause $y' = y$ is considered as an implicit conjunct of the formula.

There is precisely one transition τ_{ℓ_i} corresponding to each statement ℓ_i in P_1, and one transition τ_{m_i} for each statement in P_2. We denote their transition relations by ρ_{ℓ_i} and ρ_{m_i}, respectively. Transition relations that have two possible actions as a result of testing a condition, such as a *while* or a conditional statement, are usually represented as a disjunction $\rho_\tau^{\mathrm{T}} \vee \rho_\tau^{\mathrm{F}}$, where ρ_τ^{T} represents the case that the test evaluates to $\mathbf{T}$, while ρ_τ^{F} represents the case that the test evaluates to $\mathbf{F}$.

- $\rho_{\ell_0} : move(\ell_0, \ell_1)$
  This transition corresponds to the case that P_1 is at ℓ_0 and moves inside the loop statement.

- $\rho_{\ell_1} : move(\ell_1, \ell_2)$
  This transition represents the termination of the non-critical section.

- $\rho_{\ell_2} : move(\ell_2, \ell_3) \wedge (y_1' = \mathbf{T})$
  This transition corresponds to the case that process P_1 moves from ℓ_2 to ℓ_3 while setting y_1 to $\mathbf{T}$. Similar transitions are included for the assignment statements ℓ_5 and ℓ_9.

- $\rho_{\ell_3} : \rho_{\ell_3}^{\mathrm{T}} \vee \rho_{\ell_3}^{\mathrm{F}}$, where

  $$\begin{aligned} \rho_{\ell_3}^{\mathrm{T}} &: move(\ell_3, \ell_4) \wedge y_2 \\ \rho_{\ell_3}^{\mathrm{F}} &: move(\ell_3, \ell_8) \wedge \neg y_2 \end{aligned}$$

  The first disjunct of this transition relation corresponds to the case that the test of the *while* statement holds (i.e., $y_2 = \mathbf{T}$). In this case P_1 moves to ℓ_4. The second disjunct corresponds to the case that the test evaluates to $\mathbf{F}$, as a result of which, P_1 moves to ℓ_8.

- $\rho_{\ell_4} : \rho_{\ell_4}^{\mathrm{T}} \vee \rho_{\ell_4}^{\mathrm{F}}$, where

  $$\begin{aligned} \rho_{\ell_4}^{\mathrm{T}} &: move(\ell_4, \ell_5) \wedge (t = 2) \\ \rho_{\ell_4}^{\mathrm{F}} &: move(\ell_4, \ell_3) \wedge (t \neq 2) \end{aligned}$$

  If $t = 2$ then, according to $\rho_{\ell_4}^{\mathrm{T}}$, P_1 moves from ℓ_4 to ℓ_5. Otherwise it skips the body of the conditional statement and returns to the *while* statement at ℓ_3.

- $\rho_{\ell_6} : move(\ell_6, \ell_7) \wedge (t = 1)$
  The transition corresponding to the *await* statement at ℓ_6 is enabled only if its condition $t = 1$ is true. When taken, it moves from ℓ_6 to ℓ_7.

- $\rho_{\ell_7} : move(\ell_7, \ell_3) \wedge (y_1' = \mathbf{T})$
  This transition sets y_1 to $\mathbf{T}$ and moves from ℓ_7 to the beginning of the *while* statement at ℓ_3.

- $\rho_{\ell_8} : move(\ell_8, \ell_9)$
  This transition represents the termination of the critical section.

- $\rho_{\ell_{10}} : move(\ell_{10}, \ell_0) \wedge (y_1' = \mathbf{F})$
  This transition sets y_1 to $\mathbf{F}$ and moves to ℓ_0 to repeat the body of the loop.

A similar set of transitions corresponds to the statements of P_2.

In addition to the transitions corresponding to statements of the program, we include the idling transition τ_I, whose transition relation, according to our conventions, can be written as:

$$\rho_I \quad : \quad \mathbf{T}$$

Justice Set

As the set of just transitions, we take all the transitions except for τ_I, τ_{ℓ_1}, and τ_{m_1}. Transition τ_I is excluded since it is necessary only if no other transition is enabled, and there is no reason to insist that it will be used.

The exclusion of transitions τ_{ℓ_1} and τ_{m_1} from the justice set allows either of the processes to remain continuously in its non-critical section from a certain point on. Note that including τ_{ℓ_8} and τ_{m_8} in the justice set guarantees that each execution of the critical sections must terminate.

5 Invariance Properties

An *invariance property* is a property that can be specified by a formula of the form

$$\Box p,$$

for an assertion p. In this section, we present several rules for proving the validity of invariance properties over all computations of a program P.

5.1 A Basic Invariance Rule

For a transition τ and state formulas p and q, we define the *verification condition* of τ, relative to p and q, denoted $\{p\}\tau\{q\}$, to be the implication:

$$(\rho_\tau \land p) \rightarrow q',$$

where ρ_τ is the transition relation corresponding to τ, and q', the *primed version* of the assertion q, is obtained from q by replacing each variable occurring in q by its primed version. Since ρ_τ holds for two states s and s' iff s' is a τ-successor of s, and q' states that q holds on s', it is not difficult to see that

> if the verification condition $\{p\}\tau\{q\}$ is valid, then every τ-successor of a p-state is a q-state.

For a set of transitions $T \subseteq \mathcal{T}$, we denote by $\{p\}T\{q\}$ the conjunction of verification conditions, containing the conjunct $\{p\}\tau\{q\}$ for each $\tau \in T$. In particular, $\{p\}\mathcal{T}\{q\}$ denotes the conjunction of verification conditions for all $\tau \in \mathcal{T}$.

The following abbreviations are used to refer to the location of control in π', i.e., after the transition.

$$\begin{array}{rcl} at'\_\ell_i & : & \pi' \cap \{\ell_0, \ldots, \ell_{10}\} \;=\; \{\ell_i\} \\ at'\_m_i & : & \pi' \cap \{m_0, \ldots, m_{10}\} \;=\; \{m_i\} \end{array}$$

Since the transition relation ρ_τ often contains a conjunct of the form $move(\ell_i, \ell_j)$, we list below some implications of this formula. They can be used to simplify verification conditions.

$move(\ell_i, \ell_j)$ implies:
- $at\_\ell_i, \quad \neg at\_\ell_k$ for all $k \neq i$.
- $at'\_\ell_j, \quad \neg at'\_\ell_k$ for all $k \neq j$.
- $at'\_m_k \leftrightarrow at\_m_k$ for all k.

Symmetric implications follow from $move(m_i, m_j)$.

The Rule

A basic rule for proving invariance properties is rule B-INV.

$$\begin{array}{|ll|}\hline \text{B-INV} & \text{(Basic Invariance rule)} \\ & \text{B1.} \quad \Theta \to p \\ & \text{B2.} \quad \{p\}\mathcal{T}\{p\} \\ \cline{2-2} & \qquad \Box p \\ \hline \end{array}$$

Premise B1 of rule B-INV ensures that p holds in the first state of a computation since it is implied by Θ. Premise B2 ensures that any successor of a p-state (a state satisfying p) is also a p-state. It follows that p holds on all states of every computation of program P and, therefore, $\Box p$ is valid over P.

Example

Consider an abstract fair transition system S_1 with a single state variable x, an initial condition $x = 0$, and a single transition τ whose transition relation is given by $\rho_\tau : x' = x + 2$. Note that τ is always enabled and can be taken an unlimited number of times. This system has a single computation, given by

$$\langle x : 0 \rangle, \langle x : 2 \rangle, \langle x : 4 \rangle, \ldots$$

We wish to prove for this system the trivial invariance property

$$x \geq 0.$$

To prove this property, we use rule B-INV with $p : (x \geq 0)$. The rule requires showing the validity of the following two premises:

$$\begin{array}{lrcl} \text{B1.} & x = 0 & \to & x \geq 0 \\ \text{B2.} & x' = x + 2 \ \land \ x \geq 0 & \to & x' \geq 0 \end{array}$$

Clearly, these two implications are valid, which establishes the invariance of $x \geq 0$.

5.2 A Rule for Incremental Proofs

An assertion p that satisfies premises B1 and B2 of rule B-INV is called *inductive*. Rule B-INV claims that every inductive assertion is invariant. However, the other direction of this claim is not true. There may be invariant assertions that are not inductive. For example, the assertion $p : x \neq 1$ is invariant over system S_1 described above, but is not inductive. This is because premise B2 for this choice of p is not a first-order tautology.

One remedy to this situation is provided by *strengthening*. We find a stronger assertion φ, i.e., an assertion that implies p, which is inductive. Rule B-INV is used to establish that φ is invariant, and then we use the monotonicity property of invariance formulas given by rule MON-INV.

Rule MON-INV (Monotonicity of Invariances):

$$\{\varphi \to p \ , \ \Box\varphi\} \quad \vdash \quad \Box p$$

For example, to prove the invariance of $p : x \neq 1$ over system S_1, we may take the stronger assertion $\varphi : even(x)$ and show that it is inductive. Rule B-INV establishes the invariance of φ. Observing that $even(x)$ implies $x \neq 1$, the result follows by rule MON-INV.

An alternative approach to proving invariance of noninductive assertions is provided by rule INC-INV.

$$
\begin{array}{|ll|}
\hline
\text{INC-INV} & \text{(Incremental Invariance rule)} \\
 & \text{I1.} \quad \Box\varphi_1, \ldots, \Box\varphi_k \\
 & \text{I2.} \quad \Theta \rightarrow p \\
 & \text{I3.} \quad \{p \wedge \bigwedge_{i=1}^{k} \varphi_i\}\mathcal{T}\{p\} \\
\cline{2-2}
 & \qquad \Box p \\
\hline
\end{array}
$$

The rule assumes that several invariants, $\Box\varphi_1, \ldots, \Box\varphi_k$, have been proven before, possibly by previous applications of rules B-INV and INC-INV in combination with MON-INV. Then, premise I3 establishes a verification condition whose left-hand side contains the conjunction $\bigwedge \varphi_i$ in addition to the assertion p. Assume that $\sigma : s_0, s_1, \ldots$ is a computation of program P, and that premises I1–I3 are valid. Then premise I2 ensures, as before, that assertion p holds at s_0. Let s_i be a p-state. Since assertions $\varphi_1, \ldots, \varphi_k$ are invariant over P, s_i satisfies the conjunction $p \wedge \bigwedge \varphi_i$. By premise I3, s_{i+1} satisfies p. It follows that any successor of a p-state is also a p-state, and therefore, p is invariant over P.

5.3 Mutual Exclusion for Peterson's Program

We use the presented rules to establish the main invariance property of program PET. This is the property of mutual exclusion, stating that processes P_1 and P_2 cannot execute their critical sections at the same time. It it specified by the invariance formula

$$\Box\neg(at\_\ell_5 \wedge at\_m_5).$$

Thus, we have to show that assertion $q : \neg(at\_\ell_5 \wedge at\_m_5)$ is invariant over program PET.

Simple Range Invariants

First, we establish a list of invariants that restrict the range of values that variable s may assume and relate the values of y_1, y_2 to the locations of P_1, P_2, respectively.

To facilitate the expression of these invariants, we introduce the following abbreviations for $k \leq r$:

$$
\begin{array}{rcl}
at\_\ell_{k..r} & : & at\_\ell_k \vee at\_\ell_{k+1} \vee \cdots \vee at\_\ell_r \\
at\_m_{k..r} & : & at\_m_k \vee at\_m_{k+1} \vee \cdots \vee at\_m_r
\end{array}
$$

The assertions whose invariance states the described range restrictions are:

$$
\begin{array}{rcccc}
\varphi_0 & : & s = 1 & \vee & s = 2 \\
\varphi_1 & : & y_1 & \leftrightarrow & at\_\ell_{3..6} \\
\psi_1 & : & y_2 & \leftrightarrow & at\_m_{3..6}
\end{array}
$$

Assertion φ_0 states that s can only assume the values 1 or 2. Assertion φ_1 states that $y_1 = \mathbf{T}$ precisely when P_1 is executing at one of the locations ℓ_3–ℓ_6. Assertion ψ_1 states a similar property for P_2.

Let us see, for example, how the invariance of an assertion such as $\varphi_0 : s = 1 \lor s = 2$ is established. We apply rule B-INV with $p = \varphi_0$. There are two premises to verify.

Premise I1 requires showing that $\Theta : \pi = \{\ell_0, m_0\} \land \neg y_1 \land \neg y_2 \land s = 1$ implies the assertion $s = 1 \lor s = 2$. This is obvious.

Premise I2 requires consideration of the verification condition $(\rho_\tau \land \varphi_0) \rightarrow \varphi_0'$, for every transition τ in the program. There are some simple heuristics that let us discard immediately many transitions as automatically guaranteed to preserve φ_0. The simplest and most effective one is:

> All transitions that do not modify any of the variables on which φ depends are guaranteed to preserve φ.

This heuristic leads immediately to the conclusion that, for the assertion $\varphi_0 : s = 1 \lor s = 2$, we should only be concerned with the transitions that modify s. These are ℓ_3 and m_3. The verification condition for ℓ_3 can be written as

$$\underbrace{\cdots \land s' = 1}_{\rho_{\ell_3}} \land \underbrace{\cdots}_{\varphi_0} \rightarrow \underbrace{s' = 1 \lor \cdots}_{\varphi_0'}$$

which is obviously valid. The verification condition for m_3 is similarly valid.

We thus conclude that the assertion $\varphi_0 : s = 1 \lor s = 2$ is an invariant of the program.

As a slightly less trivial case, let us establish the invariance of assertion $\varphi_1 : y_1 \leftrightarrow at\_\ell_{3..6}$. Let us concentrate on proving premise I2. While the expression $at\_\ell_{3..6}$ is defined in terms of the control variable π that is modified by every non-idling transition of the program, it is not difficult to see that the only transitions that affect the value of the expression $at\_\ell_{3..6}$ as a whole are those that either enter or exit the range ℓ_3–ℓ_6. Consequently, we only have to consider the transitions ℓ_2 and ℓ_6. Their verification conditions can be written as

$$\underbrace{move(\ell_2, \ell_3) \land y_1' = \mathbf{T}}_{\rho_{\ell_2}} \land \underbrace{\cdots}_{\varphi_1} \rightarrow \underbrace{y_1' \leftrightarrow at'\_\ell_{3..6}}_{\varphi_1'}$$

$$\underbrace{move(\ell_6, \ell_0) \land y_1' = \mathbf{F}}_{\rho_{\ell_6}} \land \underbrace{\cdots}_{\varphi_1} \rightarrow \underbrace{y_1' \leftrightarrow at'\_\ell_{3..6}}_{\varphi_1'}$$

Since $move(\ell_2, \ell_3)$ implies at'_ℓ_3, from which follows $at'\_\ell_{3..6}$, and $move(\ell_6, \ell_0)$ implies at'_ℓ_0, from which follows $\neg at'\_\ell_{3..6}$, these verification conditions are valid.

It is clear that these are the only transitions that change the values of variable y_1 or the expression $at\_\ell_{3..6}$ on which φ_1 depends.

We conclude that $\varphi_1 : y_1 \leftrightarrow at\_\ell_{3..6}$ is an invariant assertion.

An Incremental Proof

Now, let us consider the main invariant assertion $q : \neg(at\_\ell_5 \land at\_m_5)$. We begin by attempting to prove it by rule INC-INV, taking ψ_1 as a previously proven invariant assertion. Premise I2 is trivially valid. For premise I3, we identify the only relevant transitions as ℓ_4 and m_4. The verification condition for ℓ_4 can be written as

$$\underbrace{move(\ell_4, \ell_5) \land (\neg y_2 \lor s = 2)}_{\rho_{\ell_4}} \land \underbrace{\neg(at\_\ell_5 \land at\_m_5)}_{p} \land \underbrace{y_2 \leftrightarrow at\_m_{3..6}}_{\psi_1} \rightarrow \underbrace{\neg(at'\_\ell_5 \land at'\_m_5)}_{p'}$$

Since $move(\ell_4, \ell_5)$ implies at_ℓ_4, $\neg at\_\ell_5$, at'_ℓ_5, and $at'\_m_5 = at\_m_5$, we can simplify this implication to

$$at\_\ell_4 \;\wedge\; s = 2 \quad\rightarrow\quad \neg at\_m_5,$$

which, in view of $\varphi_0 : s = 1 \vee s = 2$, can be written as

$$at\_\ell_4 \;\wedge\; at\_m_5 \quad\rightarrow\quad s = 1. \tag{1}$$

Obviously, implication (1) is not a first-order tautology. Consequently, if we believe that $q : \neg(at\_\ell_5 \wedge at\_m_5)$ is an invariant of program PET, we should adopt (1) as another invariant assertion. By this and a symmetric argument for P_2, we add to the list of invariant assertions the following two assertions:

$$\begin{array}{lcl} \varphi_2 & : & at\_\ell_4 \;\wedge\; at\_m_5 \quad\rightarrow\quad s = 1 \\ \psi_2 & : & at\_\ell_5 \;\wedge\; at\_m_4 \quad\rightarrow\quad s = 2 \end{array}$$

It is also clear that if these two are invariant then they complete the proof of invariance of $q : \neg(at\_\ell_5 \wedge at\_m_5)$.

Let us use rule INC-INV to prove the invariance of φ_2. We use the previously proven invariant $\varphi_1 : y_1 \leftrightarrow at\_\ell_{3..6}$. Premises I1 and I2 are obvious. For premise I3, the only crucial transitions are ℓ_3 (changing at_ℓ_4 from $\mathbf{F}$ to $\mathbf{T}$), m_4 (changing at_m_5 to $\mathbf{T}$), and m_3 (setting s to 2).

The corresponding verification conditions are

$$\begin{array}{lcl} \underbrace{\cdots \wedge s' = 1}_{\rho_{\ell_3}} \;\wedge\; \cdots & \rightarrow & \underbrace{\cdots \rightarrow s' = 1}_{\varphi_2'} \\ \underbrace{move(m_4, m_5) \wedge (\neg y_1 \vee s = 1)}_{\rho_{m_4}} \;\wedge\; \cdots \;\wedge\; \underbrace{y_1 \leftrightarrow at\_\ell_{3..6}}_{\varphi_1} & \rightarrow & \underbrace{at'\_\ell_4 \wedge \cdots \rightarrow s' = 1}_{\varphi_2'} \\ \underbrace{move(m_3, m_4) \wedge \cdots}_{\rho_{m_3}} \;\wedge\; \cdots & \rightarrow & \underbrace{\cdots \wedge at'\_m_5 \rightarrow \cdots}_{\varphi_2'} \end{array}$$

The verification condition for ℓ_3 is trivially valid. In the verification condition for m_4, $move(m_4, m_5)$ implies $at'\_\ell_4 = at\_\ell_4$, and $s' = s$. If $at\_\ell_4 = \mathbf{F}$, then the condition is true since $at'\_\ell_4 \wedge \cdots \rightarrow s' = 1$ reduces to $\mathbf{F} \wedge \cdots \rightarrow s' = 1$. If $at\_\ell_4 = \mathbf{T}$ then, by φ_1, $y_1 = \mathbf{T}$ and, therefore, the clause $\neg y_1 \vee s = 1$ implies $s = s' = 1$. The verification condition for m_3 follows from the observation that $move(m_3, m_4)$ implies $\neg at'\_m_5$.

This establishes the invariance of assertion $\varphi_2 : at\_\ell_4 \wedge at\_m_5 \rightarrow s = 1$. The invariance of $\psi_2 : at\_\ell_5 \wedge at\_m_4 \rightarrow s = 2$ is established in a symmetric way. This concludes the proof of mutual exclusion for program PET.

5.4 Mutual Exclusion for Dekker's Program

We proceed to establish several invariants for Dekker's program, which together yield the desired mutual exclusion property.

Simple Range Invariants

First, we establish a list of invariants that restrict the range of values that variable t may assume and relates the values of y_1, y_2 to the locations of P_1, P_2, respectively.

$$\begin{array}{lll}
\varphi_0 & : & (t=1) \vee (t=2) \\
\varphi_1 & : & y_1 \leftrightarrow (at_\ell_{3..5} \vee at_\ell_{8..10}) \\
\psi_1 & : & y_2 \leftrightarrow (at_m_{3..5} \vee at_m_{8..10})
\end{array}$$

Invariant φ_0 states that t can only assume the values 1 or 2. Invariant φ_1 states that $y_1 = \mathbf{T}$ precisely when P_1 is executing at one of the locations ℓ_3–ℓ_5 or at one of the locations ℓ_8–ℓ_{10}. Invariant ψ_1 states a similar property for P_2. All three of these assertions are inductive.

Proving Mutual Exclusion

After establishing the range invariants, we proceed to establish the main invariance property of Dekker's program, namely, that of mutual exclusion. Instead of only forbidding joint execution of statements ℓ_8 and m_8, we prove a stronger exclusion property, given by:

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

This assertion establishes mutual exclusion of the regions $\ell_{8..10}$ and $m_{8..10}$.

To show that assertion q is preserved under all transitions (i.e., premise I2 of rule B-INV), we use the following heuristic:

> To show that the assertion φ is preserved under all transitions, it is sufficient to consider only those transitions that may *potentially falsify* φ, i.e., change φ from $\mathbf{T}$ to $\mathbf{F}$.

Since assertion q is equivalent to the disjunction $\neg at\_\ell_{8..10} \vee \neg at\_m_{8..10}$, and each transition in the program can change at most one of the disjuncts but not both, the only potentially falsifying transitions are those that falsify one of the disjuncts while the other is already false. Consequently, we need only consider transitions ℓ_3 and m_3 in the cases that the respective while conditions are false. This leads to the following verification conditions:

$$\underbrace{move(\ell_3, \ell_8) \wedge \neg y_2}_{\rho^{\mathrm{F}}_{\ell_3}} \ \wedge \ \cdots \ \rightarrow \ \underbrace{\neg(at'\_\ell_{8..10} \wedge at'\_m_{8..10})}_{q'}$$

$$\underbrace{move(m_3, m_8) \wedge \neg y_1}_{\rho^{\mathrm{F}}_{m_3}} \ \wedge \ \cdots \ \rightarrow \ \underbrace{\neg(at'\_\ell_{8..10} \wedge at'\_m_{8..10})}_{q'}$$

Consider the case of $\rho^{\mathrm{F}}_{\ell_3}$. Since, $move(\ell_3, \ell_8)$ implies $at\_\ell_3 = at'\_\ell_8 = \mathbf{T}$ and $at'\_m_{8..10} = at\_m_{8..10}$, it is sufficient to show

$$\neg y_2 \ \rightarrow \ \neg at\_m_{8..10},$$

which follows from $\psi_1 : y_2 \leftrightarrow (at\_m_{3..5} \vee at\_m_{8..10})$. The condition for $\rho^{\mathrm{F}}_{m_3}$ is established in a symmetric way.

This proves the property of mutual exclusion for Dekker's program.

As we will see below, additional invariants are needed for the proof of the response properties of program DEKKER. We will develop them as they are needed.

6 Response Properties

Next to be considered is the class of *response* properties. The typical response property is expressed by the formula

$$p \Rightarrow\!\!\!\succ \Diamond\, q,$$

for assertions p and q. A sequence of states σ is said to satisfy the response formula $p \Rightarrow\!\!\!\succ \Diamond\, q$ if every p-position $i \geq 0$ is followed by a q-position $j \geq i$. Such a response formula is said to be *valid over the program P* (also called *P-valid*), denoted $P \models (p \Rightarrow\!\!\!\succ \Diamond\, q)$, if all the computations of P satisfy the formula. This means that every occurrence of (a state satisfying) p in the execution of P is followed by an occurrence of q. We will often omit the prefix $P \models$ when stating the validity of a response formula over P.

The temporal logic adepts will recognize $\Rightarrow\!\!\!\succ \Diamond$ as the combination of the two operators $\Rightarrow\!\!\!\succ$ and $\Diamond$ (see for example [MP89]). However, for our purpose here it suffices to view it as a single binary temporal operator, whose semantics has been defined above. It is very similar to the *leads-to* operator of *Unity* ([CM88]).

The following axioms and rules identify the basic properties of the *response* operator $\Rightarrow\!\!\!\succ \Diamond$.

Axiom RFLX (Reflexivity of Response):

$$p \Rightarrow\!\!\!\succ \Diamond\, p$$

This axiom expresses the fact that every p-position is trivially followed by a p-position, namely itself.

Rule TRNS (Transitivity of Response):

$$\{p \Rightarrow\!\!\!\succ \Diamond\, q \;,\; q \Rightarrow\!\!\!\succ \Diamond\, r\} \quad \vdash \quad p \Rightarrow\!\!\!\succ \Diamond\, r$$

This rule states the transitivity of the *response* operator. It claims that if every p-position is followed by a q-position, and every q-position is followed by an r-position, then certainly every p-position must be followed by an r-position.

Rule MON-RESP (Monotonicity of Response):

$$\{p \Rightarrow\!\!\!\succ \Diamond\, q \;,\; \tilde{p} \rightarrow p \;,\; q \rightarrow \tilde{q}\} \quad \vdash \quad \tilde{p} \Rightarrow\!\!\!\succ \Diamond\, \tilde{q}$$

This rule allows us to replace in a valid response formula the antecedent p by a stronger assertion $\tilde{p}$, and the consequent q by a weaker assertion $\tilde{q}$, and obtain another valid response formula.

Rule DISJ (Disjunction of Response):

$$\{p \Rightarrow\!\!\!\succ \Diamond\, r \;,\; q \Rightarrow\!\!\!\succ \Diamond\, r\} \quad \vdash \quad (p \vee q) \Rightarrow\!\!\!\succ \Diamond\, r$$

This rule combines the two response formulae, $p \Rightarrow\!\!\!\succ \Diamond\, r$ and $q \Rightarrow\!\!\!\succ \Diamond\, r$, into the formula $(p \vee q) \Rightarrow\!\!\!\succ \Diamond\, r$. It allows us to prove the last formula by separately considering the case that p holds and the case that q holds. In this way it supports proof by case analysis.

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

$$P_1 :: \left[\begin{array}{l} \ell_0 : \textbf{ while } x = 0 \textbf{ do} \\ \quad [\ell_1 : \ y \ := \ y+1] \\ \ell_2 : \end{array}\right] \quad || \quad P_2 :: \left[\begin{array}{l} m_0 : \ x \ := \ 1 \\ m_1 : \\ \end{array}\right]$$

Figure 3: Program TERM: A terminating program.

6.1 The Basic Response Rule

The axiom and three rules listed above are independent of the particular program analyzed, and describe basic properties of the response operator. We now present a rule that enables us to establish the validity of a response formula over a program.

The rule singles out a particular just transition $\tau_h \in \mathcal{J}$, to which we refer as the *helpful* transition. It can establish response formulas $p \Rightarrow \Diamond q$ that can be achieved by a single activation of the helpful transition τ_h. We therefore refer to this rule as the *basic* or *single step* response rule. The rule uses the auxiliary *intermediate assertion* φ which describes the situation between the occurrence of p and the occurrence of q.

RESP	(Basic Response rule)
	R1. $p \rightarrow (q \vee \varphi)$
	R2. $\{\varphi\}\mathcal{T}\{q \vee \varphi\}$
	R3. $\{\varphi\}\tau_h\{q\}$
	R4. $\varphi \rightarrow En(\tau_h)$
	$p \Rightarrow \Diamond q$

Premise R1 ensures that p implies q or φ. Premise R2 states that any transition of the program either leads from φ to q or preserves φ. Premise R3 states that the helpful transition τ_h leads from φ to q. Premise R4 ensures that τ_h is enabled as long as φ holds.

It is not difficult to see that if p happens, say at position $i \geq 0$, but is not followed by a q, then φ must hold continuously beyond this position, and the helpful transition τ_h is never taken beyond i. The latter fact follows from premise R3, which states that taking τ_h from a φ-state immediately leads to a q-state, contradicting the assumption that q never happens beyond i. However, due to R4, this means that τ_h is continuously enabled but never taken beyond position i, which violates the requirement of justice for τ_h.

Example

We illustrate the application of rule RESP on program TERM presented in Fig. 3

This program consists of two processes, P_1 and P_2. Process P_1 continuously increments y while waiting for x to become nonzero. Process P_2 consists of a single statement, assigning 1 to x.

The response property we wish to establish for this program is that of termination. It can be expressed by the formula

$$(at\_\ell_0 \wedge at\_m_0 \wedge x = 0) \quad \Rightarrow\Diamond \quad (at\_\ell_2 \wedge at\_m_1),$$

that states that the event of being at the beginning of the program $(at\_\ell_0 \wedge at\_m_0)$ is eventually followed by the event of being at the end of the program $(at\_\ell_2 \wedge at\_m_1)$.

This property is established by a sequence of lemmas, each applying one of the rules presented above.

Lemma 1 (x eventually set to 1)

$$(at\_\ell_0 \wedge at\_m_0 \wedge x = 0) \quad \Rightarrow\!\!\Diamond \quad (at\_\ell_{0,1} \wedge at\_m_1 \wedge x = 1)$$

This lemma claims that variable x is eventually set to 1 by process P_2, which then moves to m_1. When this happens, process P_1 is still executing within the loop $\ell_{0,1}$.

To prove the lemma we choose

$$\begin{array}{rcl} p & : & at\_\ell_0 \wedge at\_m_0 \wedge x = 0 \\ \varphi & : & at\_\ell_{0,1} \wedge at\_m_0 \wedge x = 0 \\ \tau_h & : & \tau_{m_0} \\ q & : & at\_\ell_{0,1} \wedge at\_m_1 \wedge x = 1 \end{array}$$

and apply rule RESP.

It is not difficult to see that p implies φ. It is also clear that taking τ_{m_0} from a φ-state leads to a state satisfying q (establishing premise R3), and taking any other transition, i.e., τ_{ℓ_0} or τ_{ℓ_1}, preserves φ (establishing premise R2). Obviously φ implies that τ_{m_0} is enabled (establishing premise R4).

Lemma 2 (from ℓ_0 to ℓ_2)

$$(at\_\ell_0 \wedge at\_m_1 \wedge x = 1) \quad \Rightarrow\!\!\Diamond \quad (at\_\ell_2 \wedge at\_m_1)$$

Follows from rule RESP, by taking $\varphi = p$ and $\tau_h = \tau_{\ell_0}$.

Lemma 3 (from ℓ_1 to ℓ_0)

$$(at\_\ell_1 \wedge at\_m_1 \wedge x = 1) \quad \Rightarrow\!\!\Diamond \quad (at\_\ell_0 \wedge at\_m_1 \wedge x = 1)$$

Follows from rule RESP, by taking $\varphi = p$ and $\tau_h = \tau_{\ell_1}$.

Lemma 4 (from ℓ_1 to ℓ_2)

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

Follows by transitivity (rule TRNS) from Lemma 3 and Lemma 2.

Lemma 5 (from $\ell_{0,1}$ to ℓ_2)

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

Follows by rule DISJ from Lemma 2 and Lemma 4, using the equivalence

$$(at\_\ell_{0,1} \wedge at\_m_1 \wedge x = 1) \quad \leftrightarrow \quad \left(\begin{array}{c} at\_\ell_0 \wedge at\_m_1 \wedge x = 1 \\ \vee \\ at\_\ell_1 \wedge at\_m_1 \wedge x = 1 \end{array}\right)$$

Lemma 6 (from $\{\ell_0, m_0\}$ to $\{\ell_2, m_1\}$)

$$(at\_\ell_0 \wedge at\_m_0 \wedge x = 0) \quad \Rightarrow\!\!\Diamond \quad (at\_\ell_2 \wedge at\_m_1)$$

This lemma, which establishes the termination property, follows by rule TRNS from Lemma 1 and Lemma 5. ▪

6.2 The Chain Rule for Response

The basic response rule supports the proof of response properties which are established by a *single* helpful step. As we have seen, even the simple example above requires several helpful steps to achieve its goal, i.e., termination. When the number of helpful steps required is small and fixed we can use a sequence of lemmas, each considering a single helpful step, and then combine their results by transitivity and case splitting. However, for the case that a large number of helpful steps is required, we introduce below a more powerful rule that allows us to combine the helpful steps.

The following rule for establishing $p \Rightarrow \Diamond q$ uses several *intermediate* assertions that hold between the position satisfying p and the position satisfying the goal q. We denote these assertions by φ_i, where $i = 1, \ldots, r$. Each intermediate assertion φ_i is associated with a just transition $\tau_i \in \mathcal{J}$, that is identified as *helpful* for φ_i. For uniformity, we define $\varphi_0 : q$.

We can interpret the index i of the intermediate assertion 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 state s, let φ_i be the intermediate assertion with the smallest i s.t. φ_i holds on s. We refer to the index i as the *rank* of state s.

Assuming that these constructs have been identified, the following rule establishes the P-validity of the formula $p \Rightarrow \Diamond q$.

CHAIN (Chain Rule for Response)

$$\begin{array}{ll}
\left.\begin{array}{ll}
\text{C1.} & p \rightarrow \bigvee\limits_{i=0,\ldots,r} \varphi_i \\
\text{C2.} & \{\varphi_i\}\mathcal{T}\{\bigvee\limits_{j\le i}\varphi_j\} \\
\text{C3.} & \{\varphi_i\}\tau_i\{\bigvee\limits_{j< i}\varphi_j\} \\
\text{C4.} & \varphi_i \rightarrow En(\tau_i)
\end{array}\right\} & \text{for } i = 1,\ldots,r \\
\hline
p \Rightarrow \Diamond q &
\end{array}$$

Premise C1 requires that p implies that one of the intermediate assertions φ_i (possibly $\varphi_0 = q$) holds. Premise C2 requires that taking any transition from a φ_i-state results in a next state which satisfies φ_j, for some $j \le i$. Premise C3 requires that taking the helpful transition τ_i from a φ_i-state s results in a next state which satisfies φ_j for $j < i$, i.e., a strictly lower rank than that of s. We can view premise C2 as stating that the rank never increases, while premise C3 states that the helpful transition guarantees that the rank decreases. Premise C4 claims that the helpful transition τ_i is always enabled on any state satisfying φ_i.

Assume that all four premises hold. Consider a computation σ and a position m that satisfies p. We wish to prove that some later position satisfies q. Assume to the contrary that all positions later than m (including m itself) do not satisfy q. By C1 and C2 each of these positions must satisfy some φ_j for $j > 0$, to which we refer as the rank of the position. By C2, the rank of the position can either decrease or remain the same. It follows that there must exist some position $k \ge m$, 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 C2) that φ_i holds continually beyond k. By C3, τ_i cannot be taken beyond k, because that would have led to a rank decrease. By C4, 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 hold then $p \Rightarrow\!\!\!\!\!> \Diamond q$ is P-valid.

6.3 Presentation of Proofs by Tables

When presenting a proof by rule CHAIN, we usually do not write down a detailed proof of each of the premises. Typically, it is enough to identify the intermediate assertions $\varphi_1, \ldots, \varphi_r$ and their corresponding helpful transitions $\tau_1, \ldots, \tau_r$. To convince the reader that the effect of each transition on each intermediate assertion has indeed been considered, we augment the list of assertions and helpful transitions by a *transition table* that indicates which transitions *may* lead from φ_i to φ_j for the various $i, j = 0, \ldots, r$. By inspecting this table, and finding out that transitions always lead from φ_i to φ_j with $i \geq j$, and that helpful transitions always lead from φ_i to φ_j with $i > j$, and that all transitions are accounted for, we gain greater confidence in the correctness of the proof.

Alternately, if we doubt the correctness of the proof, the transition table mentioned above provides us with a list of claims that can be checked one by one.

Let us consider the proof of the response property

$$(at\_\ell_0 \wedge at\_m_0 \wedge x = 0) \quad \Rightarrow\!\!\!\!\!> \Diamond \quad (at\_\ell_2 \wedge at\_m_1)$$

for program TERM presented in Fig. 3. Previously, we have proven this property by individual applications of rule RESP. Let us now present a proof of the same property by a single application of rule CHAIN.

As intermediate assertions and helpful transitions we choose:

i	φ_i	τ_i
3	$at\_\ell_{0,1} \wedge at\_m_0 \wedge x = 0$	m_0
2	$at\_\ell_1 \wedge at\_m_1 \wedge x = 1$	ℓ_1
1	$at\_\ell_0 \wedge at\_m_1 \wedge x = 1$	ℓ_0
0	$at\_\ell_2 \wedge at\_m_1$	

Next, we present a *transition table* that shows which transitions may lead from one intermediate assertion to another.

	φ_3	φ_2	φ_1	$\varphi_0 = q$
φ_3	ℓ_0, ℓ_1	$\underline{m_0}$	$\underline{m_0}$	
φ_2			$\underline{\ell_1}$	
φ_1				$\underline{\ell_0}$
φ_0				

The meaning of this table can be interpreted as follows. If transition τ appears in the row corresponding to φ_i and the column corresponding to φ_j, we say that φ_j is a τ-*successor* of φ_i, and also that τ *leads from* φ_i *to* φ_j. A transition that appears underlined in the row corresponding to φ_i is identified as the transition that is helpful for φ_i.

For an assertion φ_i, $i > 0$, and a transition τ, let $\varphi_{j_1}, \ldots, \varphi_{j_t}$ be all the τ-successors of φ_i. This implies that transition τ can lead from any φ_i-state only to states which satisfy one of $\varphi_{j_1}, \ldots, \varphi_{j_t}$. We define the verification condition implied by the table for φ_i and τ to be

$$\{\varphi_i\}\tau\{\varphi_{j_1} \vee \cdots \vee \varphi_{j_t}\}.$$

By convention, a transition τ that only leads from φ_i to itself is not represented explicitly at row φ_i. This means that any transition not appearing in row φ_i satisfies the verification condition

$$\{\varphi_i\}\tau\{\varphi_i\}.$$

Note that if τ is disabled on all φ_i states then it also leads from φ_i to itself. The corresponding verification condition $\rho_\tau \wedge \varphi_i \rightarrow \varphi_i$ holds trivially since the antecedent is false. Thus, such transitions do not appear in row φ_i.

We define a transition table to be *well-formed* if

- Each row for φ_i, $i > 0$, contains precisely one underlined transition, which may appear in more than one column.
- If τ leads from φ_i to φ_j, then $i \geq j$, and if τ is underlined, then $i > j$.

A transition table is defined to be *sound with respect to* assertion p if all the verification conditions implied by the table are valid and

- $p \rightarrow \bigvee\limits_{i=0,\ldots,r} \varphi_i$.
- If τ is the transition appearing underlined in the row for assertion φ_i, for $i > 0$, then $\varphi_i \rightarrow En(\tau_i)$.

Obviously, a well-formed transition table that is sound with respect to p establishes the P-validity of the response formula

$$p \quad \Rightarrow\!\!\!\Diamond \quad \varphi_0.$$

Thus, the table above presents a proof of the property

$$(at\_\ell_0 \wedge at\_m_0 \wedge x = 0) \quad \Rightarrow\!\!\!\Diamond \quad (at\_\ell_2 \wedge at\_m_1).$$

Note that the more detailed information provided by the transition table enables us to consider for each φ_i and τ the more refined verification condition

$$(\varphi_i \wedge \rho_\tau) \rightarrow (\varphi_{j_1} \vee \cdots \vee \varphi_{j_t})$$

instead of the condition required by premises C2, C3, which is

$$(\varphi_i \wedge \rho_\tau) \rightarrow (\varphi_k \vee \varphi_{k-1} \cdots \vee \varphi_0),$$

where $k = i - 1$ for the case that τ is helpful, and $k = i$ otherwise.

6.4 Presentation of Proofs by Diagrams

An alternative but equivalent presentation of proofs by rule CHAIN can be provided by *proof diagrams*. Proof diagrams convey essentially the same information provided by transition tables, but they do it in a more visual manner that helps to trace the progress from p to q along a sequence of intermediate assertions with decreasing indices.

Flat Diagrams

First, we consider the representation of transition tables by flat (unstructured) diagrams. These diagrams are labeled directed graphs constructed as follows:

- For each assertion $\varphi_i, i = 0, \ldots, r$, we construct a node and label it by φ_i.
- For each transition τ leading from φ_i to φ_j, we construct an edge connecting the node labeled by φ_i to the node labeled by φ_j, and label the edge by τ. If τ is helpful for φ_i the edge is drawn using a double line, and we refer to it as a *double edge*. Otherwise, the edge is drawn using a single line, and is called a *single edge*.

In the case that more than one transition leads from φ_i to φ_j, we draw only one edge and label it with the set of these transitions. This is considered to be equivalent to the graph in which there is a separate edge for each transition. If among the transitions leading from φ_i to φ_j there is one which is helpful but the others are not, it is necessary to draw two edges between the corresponding nodes; one double edge labeled by the helpful transition and one single edge labeled by all the rest.

In Fig. 4 we present a flat diagram representing the transition table presented above.

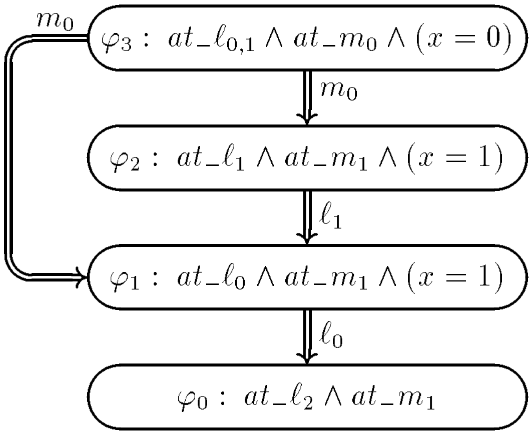

Figure 4: Flat proof diagram for program TERM.

For a transition τ that labels an edge connecting a node labeled by φ_i to a node labeled by φ_j, we say that φ_j is a *τ-successor* of φ_i in the diagram.

Verification Conditions implied by a Diagram

As in the case of transition tables, a diagram implies a set of verification conditions that are more detailed than the uniform conditions generated by premises C2 and C3 of rule CHAIN. The verification conditions implied by a diagram are defined as follows:

- Assume that τ labels at least one edge departing from φ_i, and let $\varphi_{j_1}, \ldots, \varphi_{j_t}, t > 0$, be all the τ-successors of φ_i. Then the implied verification condition is

$$\{\varphi_i\}\tau\{\varphi_{j_1} \vee \cdots \vee \varphi_{j_t}\}.$$

- For a transition τ that does not label any edge departing from φ_i, the implied condition is
$$\{\varphi_i\}\tau\{\varphi_i\}.$$

A diagram is defined to be *well-formed* if

- Each non-goal node φ_i, $i > 0$, has at least one double edge departing from it; all of then must be labeled by the same transition.
- $i \geq j$ whenever an edge connects φ_i to φ_j, and $i > j$ when this edge is double.

A diagram is said to be *sound with respect to* assertion p if all the verification conditions implied by the diagram are valid, and so are the implications

- $p \;\rightarrow\; \bigvee_{i=0,\ldots,r} \varphi_i.$
- A transition τ labeling a double edge departing from φ_i implies the condition
$$\varphi_i \;\rightarrow\; En(\tau)$$

A well-formed diagram that is sound with respect to p establishes the P-validity of the response formula
$$p \;\Rightarrow\!\Diamond\; \varphi_0.$$

Structured Diagrams

When considering large and complex programs, the flat diagrams we have introduce above tend to become cluttered and unwieldy. We therefore introduce several graphical conventions, following the style of Statecharts suggested in [Har87]. These conventions can be described as *encapsulation* conventions. They lead to more structured and hierarchical diagrams, which may considerably improve the readability and manageability of large and complex diagrams.

Composite Nodes

The basic construct of encapsulation is the introduction of a *composite node* containing one or more internal nodes. We refer to the contained nodes as the *descendants* of the composite node. Several levels of encapsulation are allowed. We refer to the nodes that are not composite, i.e., do not contain any internal nodes, as *basic nodes*.

It is possible to associate an assertion with each node in the diagram. With the basic nodes we associate the assertions labeling them. With the composite node n we associate the assertion which is the disjunction of the assertions associated with the descendants of n.

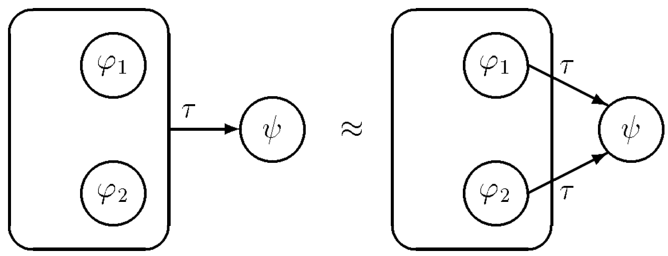

Figure 5: Distribution of departing edges.

Distribution of Common Edges

We allow edges to depart from or arrive to composite nodes. The interpretation of an edge departing from a composite node is that it is equivalent to identically labeled edges departing from each of its descendants. This interpretation is represented in the graphical equivalence presented in Fig. 5.

Note that this is fully consistent with the interpretation of a composite node as representing the disjunction of the assertions of its descendants. The diagram on the left of Fig. 5 can be interpreted as implying the verification condition

$$\{\varphi_1 \vee \varphi_2\}\ \tau\ \{\psi\},$$

since the assertion associated with the composite node is $\varphi_1 \vee \varphi_2$. The diagram on the right implies the two verification conditions

$$\{\varphi_1\}\ \tau\ \{\psi\} \qquad \text{and} \qquad \{\varphi_2\}\ \tau\ \{\psi\}.$$

Obviously, the first condition is equivalent to the conjunction of the other two.

Similarly, we interpret an edge arriving at a composite node as though it arrived at each of its descendants. This is represented by the graphical equivalence depicted in Fig. 6.

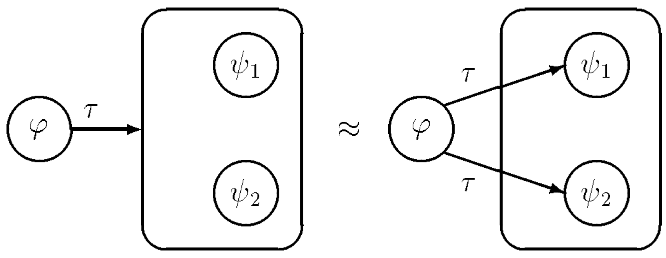

Figure 6: Distribution of arriving edges.

Again, both diagrams represent the verification condition:

$$\{\varphi\}\ \tau\ \{\psi_1 \vee \psi_2\}.$$

These conventions apply to double edge as well as to single edges.

Distribution of Common Conjuncts

A last encapsulation convention allows the removal of a conjunct that is common to the assertions of all the descendants of a composite node, and listing it at the head of the composite node. This transformation is described by the graphical equivalence presented in Fig 7.

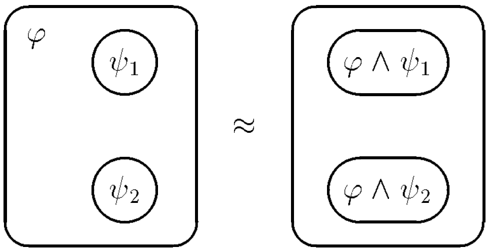

Figure 7: Common conjunct.

Thus, any structured diagram is equivalent to a flat diagram. Consequently, the notions of well-formed diagrams and the verification conditions implied by a diagram are meaningful also for structured diagrams. It is possible to check whether a given structured diagram is well-formed, or to list the verification conditions implied by such a diagram without actually constructing the equivalent flat diagram.

In Fig. 8 we present a structured version of the proof diagram previously presented in Fig. 4.

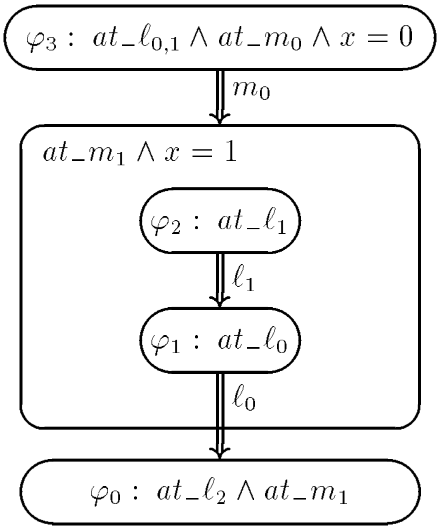

Figure 8: Structured proof diagram for program TERM.

This diagram contains a single composite node, whose descendants are (labeled by) φ_2 and φ_1. The double edge labeled by m_0, connecting φ_3 to this composite node, represents the two edges connecting φ_3 to the nodes φ_2 and φ_1 in the diagram of Fig. 4. Note also the common conjunct $at\_m_1 \wedge x = 1$, that has been factored out of the two descendants, and now labels the composite node.

6.5 Accessibility for Peteron's Program

The main response property for mutual exclusion programs is that of *accessibility*. This property states that whenever a process departs from its non-critical section it is guaranteed to eventually reach the critical section.

For program PET of Fig 1, this property can be expressed by the response formula

$$at\_\ell_2 \quad \Rightarrow\!\Diamond \quad at\_\ell_5.$$

In the response proof we make use of the assertion

$$\psi_1 \ : \ y_2 \ \leftrightarrow \ at\_m_{3..6}$$

proven to be invariant in Subsection 5.3.

In Fig. 9 we present an assertion diagram for the proof of this response property.

The diagram traces the progress of P_1 from ℓ_2 to ℓ_5. Process P_1 can progress from ℓ_2 (φ_8) to ℓ_3 (φ_7) and then to ℓ_4 with no interference from P_2. However, once location ℓ_4 is entered, it is necessary to analyze the precise location of process P_2. On entering ℓ_4, P_2 may be in any of its locations m_0–m_6 and s is set (by transition ℓ_3) to 1. This range of possibilities is covered by assertions (nodes) φ_2–φ_6. We use the invariant assertion ψ_1 to assign the appropriate value of y_2 corresponding to each of these lcations.

Within the composite node corresponding to ℓ_4, most of the progress is accomplished by P_2. Thus, transitions m_4, m_5, and m_6 are responsible for moving the computation out of states described by assertions φ_6, φ_5, and φ_4, respectively.

Assertion φ_3 is an exception, because here, it is again the responsibility of P_1 to guarantee an exit. We cannot rely on P_2 because it is allowed to remain at the non-critical section m_1 forever. However, while ℓ_4 is the helpful transision for φ_3 (enabled because $y_2 = \mathbf{F}$), it is not necessarily the transition taking the computation out of φ_3-states. It is possible that m_2 is taken first. In that case, the computation moves to φ_2, where ℓ_4 is no longer enabled, but m_3 is. Finally, if the computation reaches φ_1, then ℓ_4 is enabled again (since $s = 2$) and guarantees an eventual exit to the goal assertion $\varphi_0 : at\_\ell_5$.

6.6 Accessibility for Dekker's Program

Next, we consider program DEKKER. Accessibility for program DEKKER is expressible by the response formula

$$at\_\ell_2 \quad \Rightarrow\!\Diamond \quad at\_\ell_8.$$

That is, any state in which P_1 is observed to be at ℓ_2, implying that it is interested in entering its critical section, must eventually be followed by a state in which P_1 is observed to be at the critical section ℓ_8. A similar property is claimed for P_2.

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

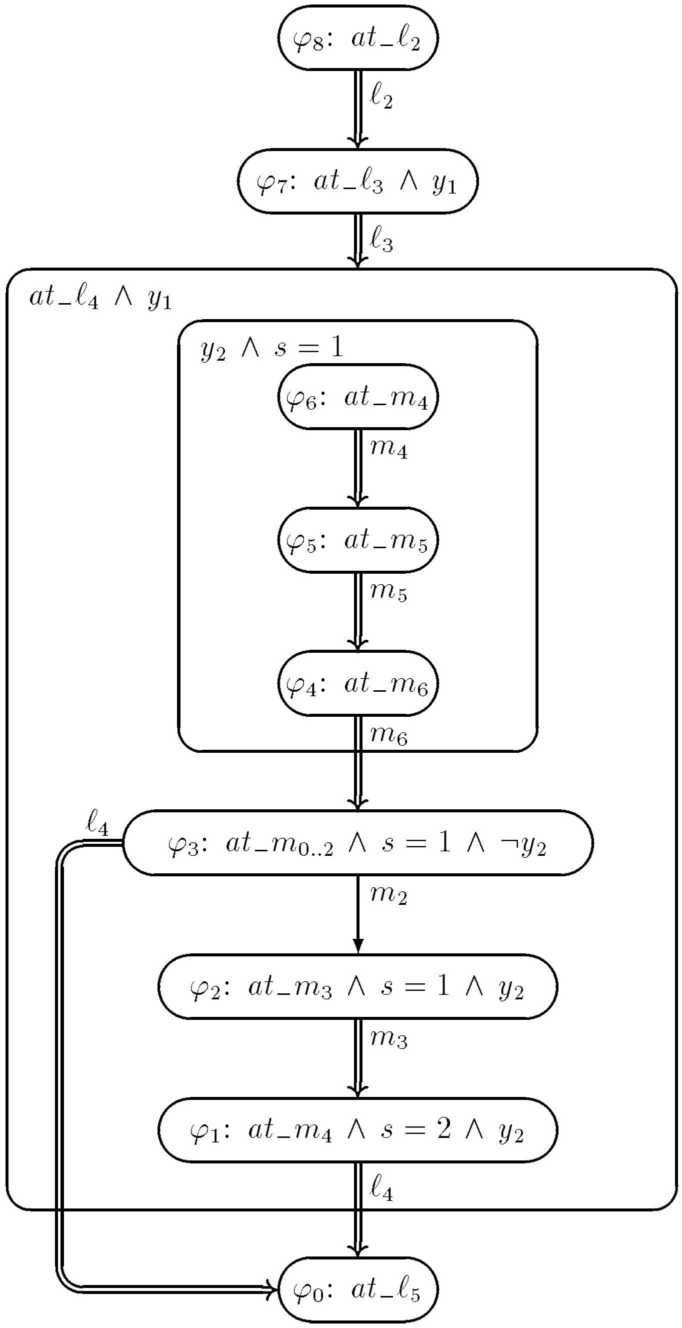

Figure 9: Proof diagram for accessibility in program PET.

Lemma A

$$at\_\ell_2 \;\; \Rightarrow\!\!\Diamond \;\; \big((at\_\ell_{3..7} \wedge t = 1) \vee at\_\ell_8\big).$$

Lemma B

$$at\_\ell_{3..7} \wedge t = 1 \;\; \Rightarrow\!\!\Diamond \;\; at\_\ell_8.$$

Obviously, the difficult part of the protocol is the loop at ℓ_3. Lemma B states that if P_1 is within this loop and has the higher priority, represented by $t = 1$, then it will get to ℓ_8. Lemma A claims that if P_1 is just starting its journey towards the critical section, then it will eventually gain a higher priority or get to ℓ_8 anyway.

Clearly, by combining these two responsiveness properties, we obtain the accessibility property.

Proof of lemma A

The proof of the response property

$$at\_\ell_2 \;\; \Rightarrow\!\!\Diamond \;\; \big((at\_\ell_{3..7} \wedge t = 1) \vee at\_\ell_8\big)$$

is presented in the diagram of Fig. 10.

It is easy to follow P_1 from ℓ_2 to ℓ_3. If on entry to ℓ_3, $t = 1$, then we are already at the goal φ_0. Otherwise we enter ℓ_3 with $t = 2$. From ℓ_3 we can either take ℓ_3^{F} and reach ℓ_8 (if $y_2 = \mathbf{T}$), which again is part of the goal, or proceed to ℓ_4 (if $y_2 = \mathbf{F}$). From ℓ_4, we proceed to ℓ_5 since $t = 2$, and then to ℓ_6 while resetting y_1 to $\mathbf{F}$. While being at ℓ_3–ℓ_5, P_2 may still set t to 1 by performing m_9, and then again we move to φ_0.

However, once we enter ℓ_6, P_1 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_3–m_9, tracing their possible flow under the relatively stable situation of $t = 2$, $y_1 = \mathbf{F}$. 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, m_1, and m_2 as possible locations while P_1 is at ℓ_6 with $y_1 = \mathbf{F}$ and $t = 2$. This assumption must hold for the program if we believe lemma A to be valid. Indeed, consider the situation that P_1 is waiting at ℓ_6 with $y_1 = \mathbf{F}$ and $t = 2$, while P_2 is at m_1. Since P_2 is allowed to stay at the non-critical section forever, this would lead to a deadlock, denying accessibility for P_1.

More Invariants

From the discussion above, it follows that if program DEKKER is correct, and guarantees accessibility to both processes, then the following assertion must be invariant:

$$\varphi_2 \;:\; at\_\ell_{4..6} \wedge t = 2 \;\rightarrow\; at\_m_{3..9}.$$

The proof of lemma A only needs $\big(at\_\ell_6 \wedge (t = 2)\big) \rightarrow at\_m_{3..9}$, but it is easy to see that if we had $at\_\ell_{4,5} \wedge t = 2 \wedge \neg at\_m_{3..9}$, we could immediately proceed to a state satisfying $at\_\ell_6 \wedge t = 2 \wedge \neg at\_m_{2..9}$ violating the assumption of the diagram.

By symmetry we should also require the invariance of

$$\psi_2 \;:\; at\_m_{4..6} \wedge t = 1 \;\rightarrow\; at\_\ell_{3..9}.$$

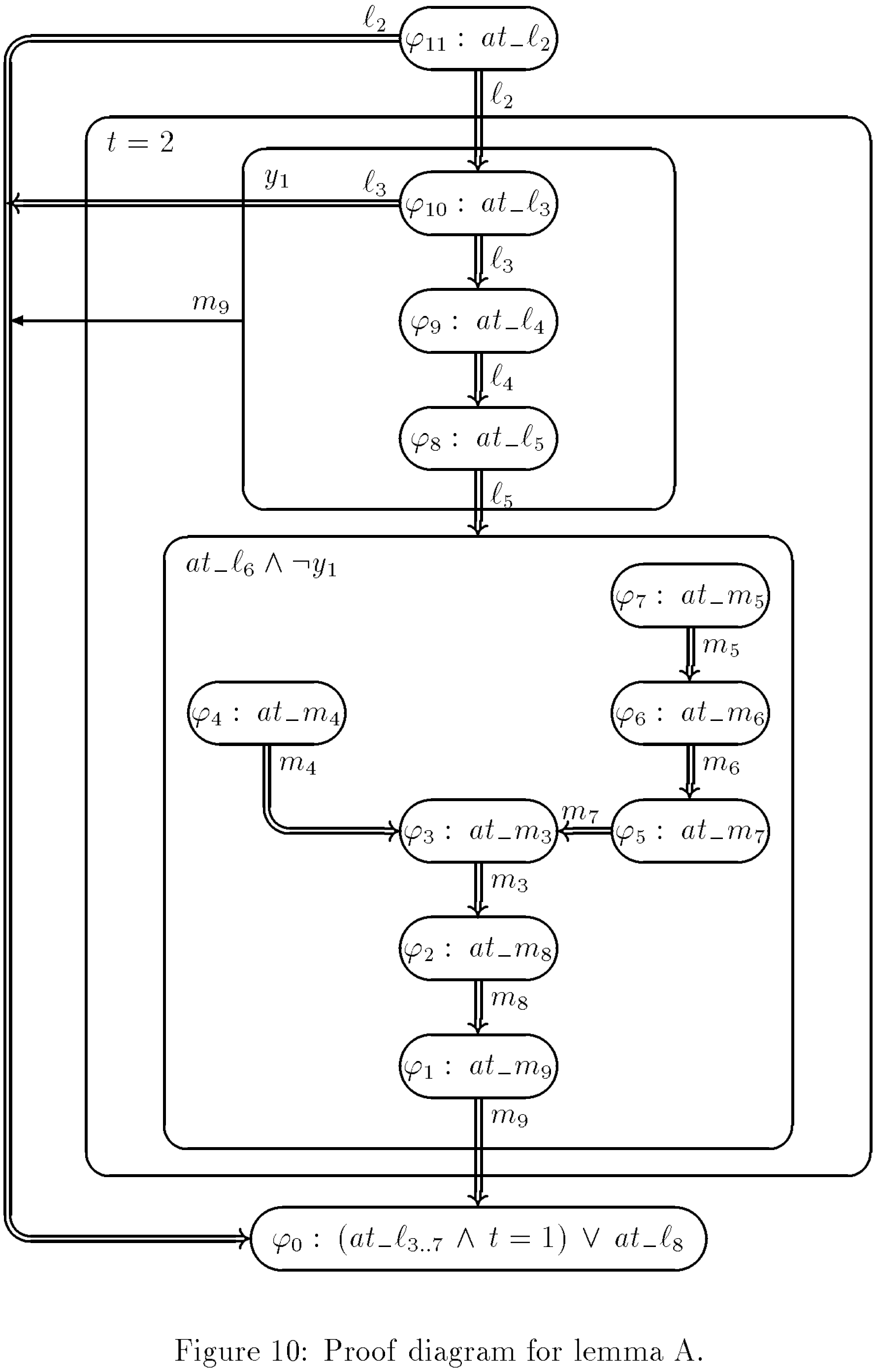

Figure 10: Proof diagram for lemma A.

Let us check the verification conditions for φ_2.

Clearly, φ_2 is implied by Θ.

There are three transitions that may potentially falsify φ_2.

- Transition m_9 — making $at\_m_{2..9}$ false.

  This transition sets t to 1 which makes $t = 2$ false and hence preserves the truth of the assertion.

- Transition ℓ_9 — making $t = 2$ true.

  This transition leads to at_ℓ_{10} which makes $at\_\ell_{4..6}$ false.

- ℓ_3^{T} while $t = 2$ — making $at\_\ell_{4..6}$ true.

  This is possible only if $y_2 = \mathbf{T}$ which, by $\psi_1 : y_2 \leftrightarrow (at\_m_{3..5} \vee at\_m_{8..10})$, implies $at\_m_{3..10}$. This almost gives us $at\_m_{3..9}$, with the exception of m_{10}. We thus need additional information that excludes the possibility of P_2 being at m_{10} while $t = 2$. This information is provided by the invariants we will develop next.

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 $q : \neg(at\_\ell_{8..10} \wedge at\_m_{8..10})$, is still a part of the critical section and therefore ℓ_9, the only statement capable of changing t to 2, cannot be enabled.

This leads us to the invariance of

$$\psi_3 \quad : \quad at\_m_{10} \;\rightarrow\; t = 1$$

and its symmetric counterpart

$$\varphi_3 \quad : \quad at\_\ell_{10} \;\rightarrow\; t = 2.$$

To prove ψ_3, we should inspect two transitions:

- Transition m_9 — making at_m_{10} true.

  This transition sets t to 1.

- ℓ_9 while at_m_{10} — making $t = 1$ false.

  Impossible due to the invariance of $q : \neg(at\_\ell_{8..10} \wedge at\_m_{8..10})$.

This establishes the invariance of ψ_3 and symmetrically φ_3. Having ψ_3 we can use it to show that the last transition considered in the proof of φ_2, namely ℓ_3^{T} while $t = 2$, implies $at\_m_{3..9}$ (excluding at_m_{10}), which establishes the invariance of φ_2.

Proof of Lemma B

Lemma B states that if P_1 is within the waiting loop ℓ_3 with higher priority, i.e., $t = 1$, then eventually it will reach ℓ_8. It is stated by $(at\_\ell_{3..7} \,\wedge\, t = 1) \Rightarrow \Diamond at\_\ell_8$. The proof is presented in the diagram of Fig. 11.

The diagram identifies several major phases in the progress of P_1 towards its critical section. First we follow P_1 through ℓ_5, ℓ_6, ℓ_7, until it reaches $\ell_{3,4}$. Its progress is not hindered at ℓ_6, since $t = 1$, and no transition of P_2 can change this fact. Once P_1 gets to $\ell_{3,4}$ with $y_1 = \mathbf{T}$, the diagram recognizes the following cases:

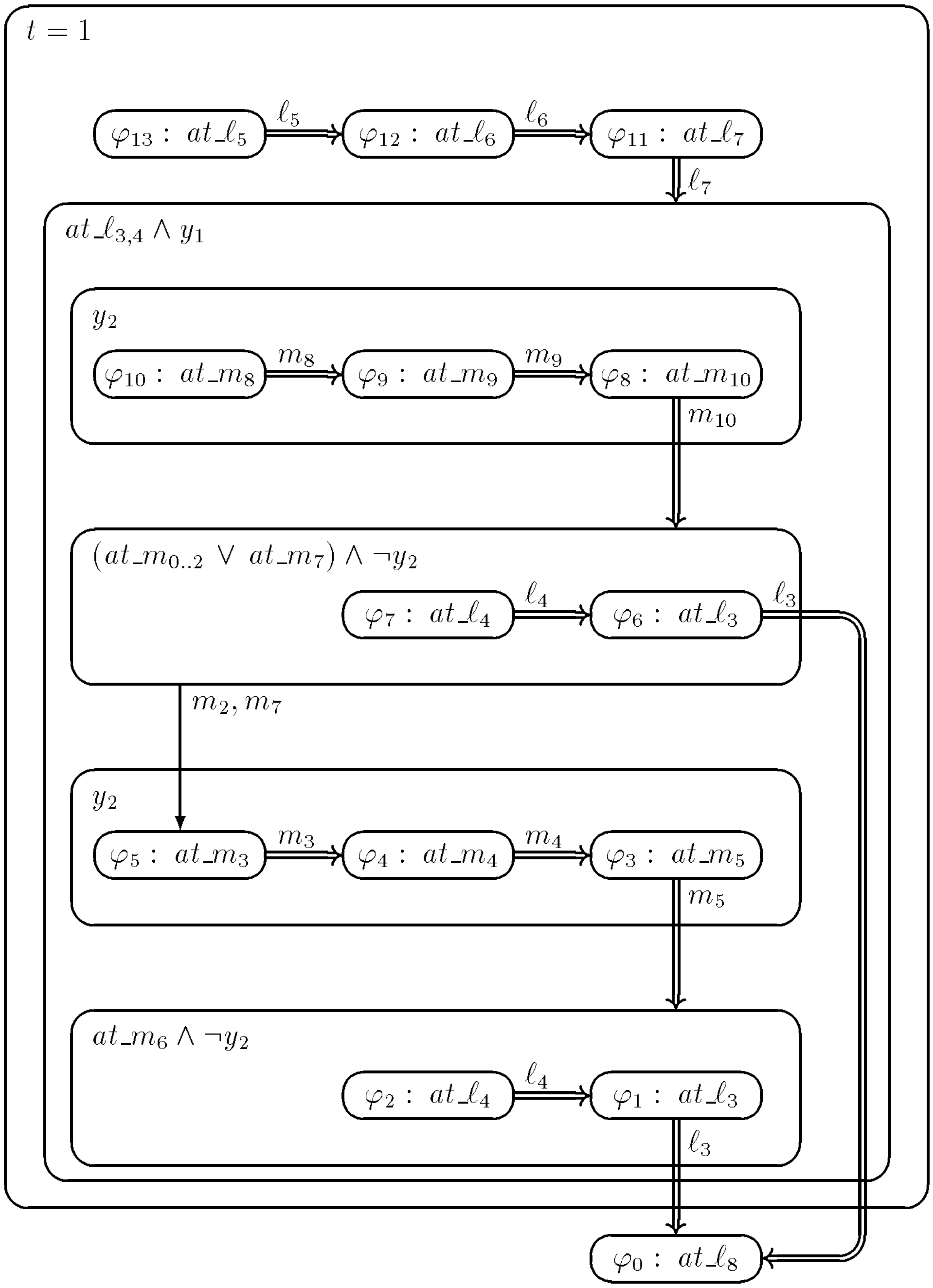

Figure 11: Proof diagram for lemma B.

- P_2 is at $m_{8..10}$ with $y_2 = \text{T}$.

  Eventually P_2 departs through m_{10} to m_0, while setting y_2 to F.

- P_2 is at $m_{0..2,7}$ with $y_2 = \text{F}$.

  There are two ways to exit out of this situation. Either P_1 will reach and perform ℓ_3^{F} first, exiting from the while loop to the critical section. Alternately, P_2 may perform first one of m_2 or m_7 and move to m_3, while setting y_2 to T.

- P_2 is at $m_{3..5}$ with $y_2 = \text{T}$.

  Eventually, P_2 performs m_5 and moves to m_6 while setting y_2 to F.

- P_2 is at m_6 with $y_2 = \text{F}$.

  Clearly, P_2 is now blocked at m_6 since $t = 1$. This allows P_1 to advance at its own pace to ℓ_3, find $y_2 = \text{F}$, and move to the critical section at ℓ_8.

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 $y_2 = \text{T}$ 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 particular situation, it is P_2 which is the helpful process and we have to trace its progress.

On the other hand, when $y_2 = \text{F}$, 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 of $y_2 = \text{T}$, but it is impossible to exit m_6 into such a situation.

This concludes the proof of the accessibility property for Dekker's program.

7 Precedence Properties

Next, we consider properties that are expressed by formulas of the form

$$p \quad \Rightarrow \quad q_r \mathcal{W} \ldots \mathcal{W} q_1 \mathcal{W} q_0,$$

for any $r > 0$. Adepts in temporal logic will recognize this formula as a nested *waiting-for* formula. For our purposes here it suffices to consider it as a temporal operator of $r+2$ arguments.

To define the semantics of this operator, we deal with *half-open intervals* of the form $[i..j)$, for $i \leq j$. Such an interval consists of all the positions k, such that $i \leq k < j$. Note that if $i = j$ the interval is empty. For the two intervals $[i..j)$ and $[j..k)$, we say that the second interval is adjacent to (or follows) the first, and observe that their union is the half open interval $[i..k)$. We also allow intervals of the form $[i..\omega)$ for an integer $i \geq 0$, and the interval $[\omega, \omega)$ which, by definition, is empty.

Given a computation $\sigma : s_0, s_1, \ldots$, we say that the interval $[i..j)$ is a *p-interval* if for every $k \in [i..j)$, s_k satisfies p. By definition, an empty interval is a p-interval for every assertion p.

A computation σ is said to satisfy the precedence formula $p \Rightarrow q_r \mathcal{W} \ldots \mathcal{W} q_1 \mathcal{W} q_0$ if for every p-position i there exists a sequence of positions $i = i_r \leq i_{r-1} \leq \ldots \leq i_0 \leq \omega$, such that $[i_r..i_{r-1})$ is a q_r-interval,...,$[i_1..i_0)$ is a q_1-interval, and finally if $i_0 < \omega$, then i_0 is a q_0-position. That is, it requires that any p-position initiates a q_r-interval, which is followed by a q_{r-1}-interval,..., which is followed by a q_1-interval, which either extends to infinity or is terminated by a q_0-position. Note that this definition allows some of the intermediate intervals to be empty, and any of them to extend to infinity, in which case, all succeeding intervals are empty and there is no terminating q_0-position.

The precedence formula $p \Rightarrow q_r \mathcal{W} \ldots \mathcal{W} q_1 \mathcal{W} q_0$ is said to be *P-valid* if it satisfied by all computations of program P.

7.1 Bounded Overtaking

Consider program PET of Fig. 1. In the previous section we proved that whenever process P_1 exits its non-critical section it eventually reaches its critical section. However, this guarantee puts no measure on how long it takes for P_1 to reach the critical section. In particular, it allows the algorithm to be grossly unfair to one of the processes, allowing P_1 one critical entry for each 10 critical entries of P_2. To specify that this does not happen and that the algorithm is reasonably fair to each of the processes, we may impose the following requirement:

> From the time P_1 is at ℓ_4, P_2 may enter its critical section ahead of P_1 (overtake P_1) at most once.

We refer to this property as 1-*bounded overtaking*.

For program PET, 1-bounded overtaking from location ℓ_4 can be specified by the precedence formula

$$at\_\ell_4 \quad \Rightarrow \quad (\neg at\_m_{5,6}) \mathcal{W} (at\_m_{5,6}) \mathcal{W} (\neg at\_m_{5,6}) \mathcal{W} at\_\ell_{5,6}$$

The formula states that, if P_1 is currently at ℓ_4, then there may be an interval in which P_2 is not in $m_{5,6}$, followed by an interval in which P_2 is in $m_{5,6}$, followed by an interval in which P_2 is not in $m_{5,6}$, followed by an entry of P_1 to $\ell_{5,6}$. Any of the intervals may be empty, in particular the interval of P_2 being in $m_{5,6}$, which also allows the entry of P_1 to $\ell_{5,6}$ without P_2 getting to $m_{5,6}$ first. Also, any of the intervals may be infinite, in which case all the following intervals, as well as the entry of P_1 to $\ell_{5,6}$, are not guaranteed. This, however, is not possible because of the previously proven accessibility property for P_1.

7.2 A Rule for Precedence

We present a single rule PREC for proving precedence formulas. Similar to rule CHAIN, rule PREC uses auxiliary assertions $\varphi_0, \ldots, \varphi_r$ that strengthen assertions $q_0, \ldots, q_r$.

PREC (Precedence Rule)

$$\begin{array}{lll}
\text{P1.} & p \;\rightarrow\; \bigvee_{i=0}^{r} \varphi_i & \\
\text{P2.} & \varphi_i \;\rightarrow\; q_i & \text{for } i = 0, \ldots, r \\
\text{P3.} & \{\varphi_i\} \mathcal{T} \{\bigvee_{j \leq i} \varphi_j\} & \text{for } i = 1, \ldots, r \\
\hline
 & p \Rightarrow q_r \mathcal{W} q_{r-1} \cdots q_1 \mathcal{W} q_0 &
\end{array}$$

To justify this rule consider a state s_k satisfying p. By premise P1, it also satisfies $\bigvee_{i=0}^{r} \varphi_j$. It follows that there exists some index j_k, $0 \leq j_k \leq r$, such that s_k satisfies φ_{j_k}. If $j_k = 0$ we are done, since φ_0 terminates the required sequence of intervals. Otherwise, $j_k > 0$ and we consider s_{k+1} the successor of s_k. Premise P3 implies that s_{k+1} satisfies $\varphi_{j_{k+1}}$ for some $j_{k+1} \leq j_k$. We now repeat the argument for s_{k+1}, and so on. Denoting the indices of the assertions established for s_k and its successors by $j_k, j_{k+1}, \ldots$, premise P3 guarantees that the sequence of indices

$$j_k \;\geq\; j_{k+1} \;\geq\; \cdots$$

is nonincreasing, and that it can either terminate at some $j_m = 0$ or extend to infinity. It is not difficult to see that this guarantees a sequence of intervals, $I_r, I_{r-1}, \ldots, I_1$ satisfying respectively

$$\varphi_r,\; \varphi_{r-1},\; \ldots,$$

which may either terminate at a state satisfying φ_0 or extend to infinity. Clearly, some of these intervals may be empty. By premise P2, any interval or state satisfying φ_i also satisfies q_i. It follows that $q_r \mathcal{W} q_{r-1} \ldots q_1 \mathcal{W} q_0$ holds at position k in the computation.

7.3 1-Bounded Overtaking for Peterson's Program

As explained above, 1-bounded overtaking for Peterson's program is specified by the formula

$$at\_\ell_4 \;\Rightarrow\; (\neg at\_m_{5,6})\mathcal{W}(at\_m_{5,6})\mathcal{W}(\neg at\_m_{5,6})\mathcal{W}(at\_\ell_{5,6})$$

We use rule PREC to prove this property for program PET.

To use rule PREC, we have to find four assertions φ_0, φ_1, φ_2, φ_3, whose disjunction is implied by at_ℓ_4 (satisfying P1), which strengthen the assertions $at\_\ell_{5,6}$, $\neg at\_m_{5,6}$, $at\_m_{5,6}$, $\neg at\_m_{5,6}$, respectively (satisfying P2), and which satisfy the verification conditions of premise P3 of the rule.

A natural candidate for φ_0 is $at\_\ell_{5,6}$ itself,

$$\varphi_0 \;:\; at\_\ell_{5,6},$$

because, obviously, it terminates the waiting period. Proceeding to φ_1, the assertion φ_1 should strengthen $\neg at\_m_{5,6}$, and we can safely add to it the conjunct at_ℓ_4, since the whole period starts with P_1 at ℓ_4 and terminates by P_1 moving to ℓ_5.

What additional information should we include in φ_1? Considering the role of φ_1 in the precedence formula and premise P3, φ_1 should be such that the only exit to a $(\neg\varphi_1)$-state would be to an $(at\_\ell_{5,6})$-state. It follows that φ_1 should characterize all the states in which the next entry to a critical section will be by P_1, i.e., all the states in which P_1 has a definite priority over P_2.

Observing that $at\_\ell_4 \wedge at\_m_4 \wedge s = 2$ is one such a state, we can add to the assertion all other states from which this state is reachable by movements of P_2 alone. This leads to the assertion

$$\varphi_1 \;:\; at\_\ell_4 \;\wedge\; \Big(at\_m_{0..3} \;\vee\; (at\_m_4 \;\wedge\; s = 2)\Big).$$

For the assertion φ_2, it seems sufficient to take

$$\varphi_2 \ : \ at\_\ell_4 \ \wedge \ at\_m_{5,6}.$$

For φ_3 we have to characterize all the states in which P_2 has priority over P_1, which waits at ℓ_4. Seeing that φ_1 and φ_2 cover almost all the configurations satisfying at_ℓ_4, the only remaining one is given by

$$\varphi_3 \ : \ at\_\ell_4 \ \wedge \ at\_m_4 \ \wedge \ s = 1.$$

From the way φ_0–φ_3 were constructed, it is obvious that at_ℓ_4 implies their disjunction (premise P1), that each of them is a strengthening of the corresponding assertion in the precedence formula (premise P2), and that φ_1–φ_3 satisfy premise P3 of rule PREC.

7.4 Tables and Diagrams for Precedence Proofs

Similar to proofs by rule CHAIN, proofs by rule PREC can be presented by both transition tables and proof diagrams. The main differences are that we no longer identify helpful transitions and that the existence of a table entry or graph edge leading from φ_i to φ_j only requires that $i \geq j$.

For example, the proof of 1-bounded overtaking from ℓ_4 can be represented by the following table:

	φ_3	φ_2	φ_1	φ_0
$\varphi_3 : at\_\ell_4 \ \wedge \ at\_m_4 \ \wedge \ s = 1$		m_4		
$\varphi_2 : at\_\ell_4 \ \wedge \ at\_m_{5,6}$			m_6	
$\varphi_1 : at\_\ell_4 \ \wedge \ \big(at\_m_{0..3} \ \vee \ (at\_m_4 \ \wedge \ s = 2)\big)$				ℓ_4
$\varphi_0 : at\_\ell_{5,6}$				

It can also be presented in the proof diagram of Fig. 12

Acknowledgement

We gratefully acknowledge the help rendered by Eddie Chang who critically read various versions of this manuscript and proposed significant improvements.

References

[AS89] B. Alpern and F.B. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. Prog. Lang. Sys.*, 11:147–167, 1989.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[Dij65] E. W. Dijkstra. Co-operating sequential processes. In *Programming Languages* (F. Genuys, editor), pages 43–112. Academic Press, 1965.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274, 1987.

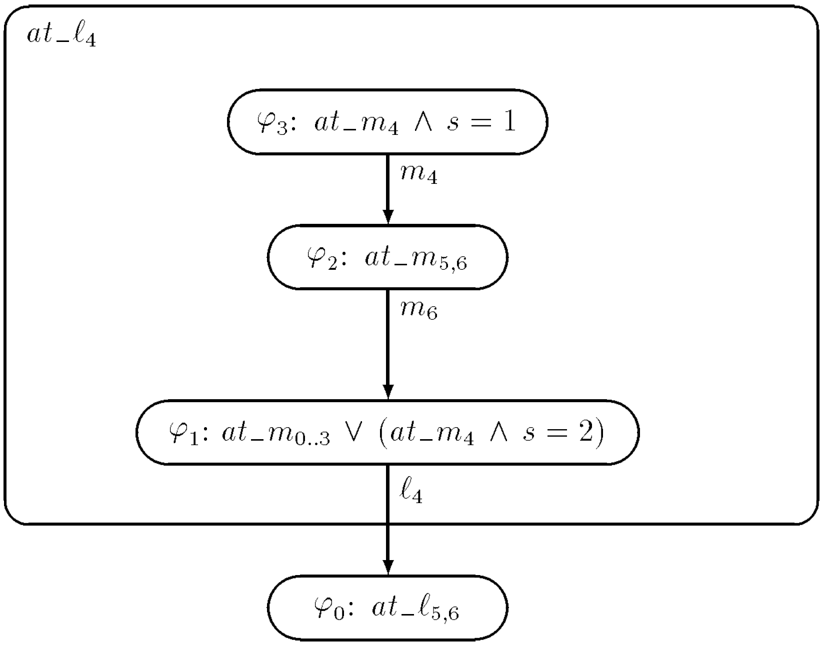

Figure 12: Proof diagram for 1-bounded overtaking in program PET.

[Lam83] L. Lamport. What good is temporal logic. In *Proc. IFIP 9th World Congress* (R.E.A. Mason, editor), pages 657–668. North-Holland, 1983.

[MP83] Z. Manna and A. Pnueli. Proving precedence properties: The temporal way. In *Proc. 10th Int. Colloq. Aut. Lang. Prog.*, volume 154 of *Lect. Notes in Comp. Sci.*, pages 491–512. Springer-Verlag, 1983.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, pages 1–12, 1987.

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors), pages 201–284. Lec. Notes in Comp. Sci. 354, Springer, 1989.

[MP90] Z. Manna and A. Pnueli. A temporal proof methodology for reactive systems. In *5th Jerusalem Conference on Information Technology*, pages 757–773, 1990.

[MP91a] Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[Pet83] G. L. Peterson. A new solution to lamport's concurrent programming problem. *ACM Trans. Prog. Lang. Sys.*, 5(1):56–65, 1983.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pages 46–57, 1977.

[Szy88] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. 1988 International Conference on Supercomputing Systems*, pages 621–626, St. Malo, France, 1988.