A Temporal Proof Methodology for Reactive Systems *

Zohar Manna
Stanford University †
and
Weizmann Institute of Science‡

Amir Pnueli
Weizmann Institute of Science‡

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* and *response* 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*.

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, independently 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.

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 worry about how they interact with one another, he is not tempted to overspecify or actually *design* the system. Consequently, this approach leads to specifications which are considerably 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 in a newcomer to the area 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 absolutely 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 veri-

*This research was supported in part by the National Science Foundation under grants CCR-89-11512 and CCR-89-13641; by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, 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 3096 (SPEC).

†Department of Computer Science, Stanford University, Stanford, CA 94305

‡Department of Applied Mathematics, Weizmann Institute, Rehovot, Israel

fier (hoping that such a position will eventually become a standard in any quality assurance team) typically spends most of his time, usually fall into two or three 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 he needs. This pragmatic approach can be nicely complemented by a more theoretical presentation of a comprehensive theory of a language of specification (temporal logic would have been our choice), its power to express a wide spectrum of program properties, and a comprehensive proof theory and investigation of its completeness (see for example [MP89a]). However, as stated above, it may be an educational mistake to require the study of such a comprehensive approach as an essential requisite for the pragmatic application of the verification tools that result from the general theory.

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.

In this paper we consider two classes of properties, which we believe to cover the majority of properties one would ever wish to verify for a reactive or a concurrent program.

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

The two properties 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 $p \Rightarrow \Diamond q$. In the Unity notation (see [CM88]), this property is called a *leads-to* property, and written as $p \mapsto q$.

We refer the reader to [MP89b] for a top-down approach, which attempt to present the most general proof rules that cover as many properties as possible. Here, however, we take the opposite approach of presenting rules that are closely tailored for the restricted classes that are most frequently needed.

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.

- $V = \{u_0, ..., u_{n-1}\}$ – 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 domain, over which it ranges.

- Σ – A set of *states*. Each state $s \in \Sigma$ is an interpretation of V, assigning to each variable $y \in V$ a value over its domain, which we denote by $s[y]$.

- $\mathcal{T}$ – A set of *transitions*. Each transition $\tau \in \mathcal{T}$ is associated with an assertion $\rho_\tau(V, V')$, called the *transition relation*, which refers to both an unprimed and a primed version 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.

- Θ – The *precondition*. This is an 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.

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 $s'[x]$. That is,

$$\langle s, s'\rangle \models \rho_\tau(V, V').$$

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. We say that a state s is *terminal* if all the transitions $\tau \in \mathcal{T}$ are disabled on it. 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.

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

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

a finite or infinite sequence of states of P.

We say that the 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 (completed) at position* $k+1, k = 0, 1, \ldots$, 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, i.e., $s_0 \models \Theta$.
- *Consecution* For each $j = 0, 1, ...$, 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 $\tau \in \mathcal{T}$.
- *Termination* Either σ is infinite, or it ends in a state s_k which is terminal.
- *Justice* For each transition $\tau \in \mathcal{T}$, 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.

For a program P, we denote by $\mathcal{C}omp(P)$ the set of all computations of P. 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 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 Example — Dekker's Algorithm

The program we will use as our main example is Dekker's algorithm for mutual exclusion [Dij65]. This has been one of the earliest (possibly the first) correct solutions to this problem. In this problem, two processes P_1 and P_2 attempt to coordinate their entry to two *critical sections* that are parts of their code, in a way that guarantees that only one of the two processes is executing in its critical section at any given moment.

In Dekker's solution, each of the processes has 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 a true y_i, we use a tie-breaker, the variable t (short for *turn*). This variable ranges over $\{1, 2\}$, and in case of a tie, the process whose number is in t has the higher priority. To ensure fair accessibility, process P_i sets the variable t to the value corresponding to its rival, i.e., to $3 - i$, on exit from the critical section. Since this assignment is performed immediately following the critical section it is done within a protected environment, secure from any interference by the other process.

The Algorithm

Dekker's algorithm, which is based on these ideas, is presented in Fig. 1.

Let us follow P_1 on exit from the non-critical section. This is where the protocol of coordination between the two processes starts. Process P_1 sets at ℓ_2 its y_1 variable to T. It then enters a while loop that continues as long as P_1 detects a true value of y_2. When P_1 detects a true y_2 it identifies the 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 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 leaves its y_1 variable true and 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. The process P_1 enters its critical section at ℓ_8 only when it detects a false y_2. After termination of the critical section, P_1 sets first t to 2 and then resets y_1 to false.

4 The Program as a Fair Transition System

Let us consider how the program presented in Fig. 1 can be viewed as a concrete case of a fair transition

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

$$
P_1 :: \left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\left[\begin{array}{ll}
\ell_1 : & \text{Non Critical} \\
\ell_2 : & y_1 \ := \ \text{T} \\
\ell_3 : & \textbf{while } y_2 \textbf{ do} \\
 & \quad \ell_4 : \ \textbf{if } (t = 2) \textbf{ then} \\
 & \quad \left[\begin{array}{ll}
 \ell_5 : & y_1 \ := \ \text{F} \\
 \ell_6 : & \textbf{await } (t = 1) \\
 \ell_7 : & y_1 \ := \ \text{T}
 \end{array} \right] \\
\ell_8 : & \text{Critical} \\
\ell_9 : & t \ := \ 2 \\
\ell_{10} : & y_1 \ := \ \text{F}
\end{array} \right]
\end{array} \right]
\quad || \quad
P_2 :: \left[\begin{array}{l}
m_0 : \textbf{loop forever do} \\
\left[\begin{array}{ll}
m_1 : & \text{Non Critical} \\
m_2 : & y_2 \ := \ \text{T} \\
m_3 : & \textbf{while } y_1 \textbf{ do} \\
 & \quad m_4 : \ \textbf{if } (t = 1) \textbf{ then} \\
 & \quad \left[\begin{array}{ll}
 m_5 : & y_2 \ := \ \text{F} \\
 m_6 : & \textbf{await } (t = 2) \\
 m_7 : & y_2 \ := \ \text{T}
 \end{array} \right] \\
m_8 : & \text{Critical} \\
m_9 : & t \ := \ 1 \\
m_{10} : & y_2 \ := \ \text{F}
\end{array} \right]
\end{array} \right]
$$

Figure 1: Dekker's algorithm for mutual exclusion.

system.

Below, we identify the four components of a fair transition system, namely, state variables, states, transitions, and precondition, for Dekker's program. This identification enables us to view the program as a fair transition system, and apply to it the verification methods that will be later presented for a general fair transition system.

- V – The state variables, are given by

  $$\pi_1, \pi_2, y_1, y_2, t.$$

  The variables π_1, π_2 are control variables that range over the set $\{0, ..., 10\}$. At any state of the computation, $\pi_1 = k$, for $k = 0, ..., 10$ if and only if process P_1 is currently in front of the statement labeled ℓ_k. In a similar way, $\pi_2 = k$ if and only if P_2 is currently in front of the statement labeled m_k. We often use the expressions at_ℓ_k and at_m_k as abbreviations for the assertions $\pi_1 = k$ and $\pi_2 = k$, respectively.

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

- Σ – The states, consist of all the possible assignments to the state variables of values in their respective domains.

- Θ – The precondition is given by the assertion

  $$\Theta \quad : \quad (\pi_1 = \ell_0) \wedge (\pi_2 = m_0) \wedge \neg y_1 \wedge \neg y_2 \wedge (t = 1)$$

  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 F, and t is initialized to 1.

To express the movement of control effected by the transitions, we introduce the following abbreviations for $i = 1, 2$:

$$
\begin{array}{lll}
move(i, k, r) & : & (\pi_i = k) \wedge (\pi_i' = r) \\
stay(i, k) & : & \pi_i = \pi_i' = k
\end{array}
$$

Clearly, $move(1, k, r)$ describes the movement of process P_1 from ℓ_k to ℓ_r, while $stay(1, k)$ describes the situation that P_1 is currently at ℓ_k but chooses not to move. The expressions $move(2, k, r)$ and $stay(2, k)$ describe similar situations for process P_2.

The 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.

In defining the transition relation ρ_τ corresponding to the transition τ, we adopt the following convention. We present a *compact transition relation* R_τ, which contains the conditions under which τ is enabled, and the effect τ has on the variables it may modify. The full transition relation ρ_τ is given by a conjunction of R_τ with a list of clauses $u' = u$ for each variable u whose primed version does not appear in R_τ, i.e., a variable that is obviously preserved by τ.

As we will see, there is precisely one transition τ_{ℓ_k} corresponding to each statement ℓ_k in P_1, and one transition τ_{m_k} for each statement in P_2. We denote their full and compact transition relations by ρ_{ℓ_k} and R_{ℓ_k} for P_1 and by ρ_{m_k} and R_{m_k} for P_2. 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 $\rho_\tau^{\mathbf{T}} \vee \rho_\tau^{\mathbf{F}}$, where $\rho_\tau^{\mathbf{T}}$ represents the case that the test evaluated to T, while $\rho_\tau^{\mathbf{F}}$ represents the case that the test evaluated to F. A similar convention is adopted for the compact transition relations of such statements.

- $R_{\ell_0} : move(1, 0, 1)$
  This transition corresponds to the case that P_1 is at ℓ_0 and moves inside the endless **loop** statement.

- $R_{\ell_1} : stay(1, 1) \vee move(1, 1, 2)$

  This compact transition relation consists of two clauses representing a non-deterministic choice. The first clause corresponds to the case that the process P_1 decides to remain in its non-critical section for awhile longer. The situation that, from a certain point on, a process remains forever in its non-critical section (which we want to include) is represented by this process consistently choosing this clause of the transition relation from that point on.

  The second clause of the compact transition relation corresponds to the case that P_1 decides to quit its non-critical section and move from ℓ_1 to ℓ_2.

- $R_{\ell_2} : move(1, 2, 3) \wedge (y_1' = \mathrm{T})$

  This transition corresponds to the case that the process P_1 moves from ℓ_2 to ℓ_3 while setting y_1 to T.

- $R_{\ell_3} : R_{\ell_3}^{\mathbf{T}} \vee R_{\ell_3}^{\mathbf{F}}$, where

$$\begin{aligned} R_{\ell_3}^{\mathbf{T}} &: move(1, 3, 4) \wedge y_2 \\ R_{\ell_3}^{\mathbf{F}} &: move(1, 3, 8) \wedge \neg y_2 \end{aligned}$$

  The first disjunct of this compact transition relation corresponds to the case that the while's test holds (i.e., $y_2 = \mathrm{T}$). In this case P_1 moves to ℓ_4. The second disjunct corresponds to the case that the test evaluated to F, as a result of which, P_1 moves to ℓ_8.

- $R_{\ell_4} : R_{\ell_4}^{\mathbf{T}} \vee R_{\ell_4}^{\mathbf{F}}$, where

$$\begin{aligned} R_{\ell_4}^{\mathbf{T}} &: move(1, 4, 5) \wedge (t = 2) \\ R_{\ell_4}^{\mathbf{F}} &: move(1, 4, 3) \wedge (t \neq 2) \end{aligned}$$

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

- $R_{\ell_6} : move(1, 6, 7) \wedge (t = 1)$

  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.

- $R_{\ell_7} : move(1, 7, 3) \wedge (y_1' = \mathrm{T})$

  The transition sets y_1 to T and moves from ℓ_7 to the beginning of the *while* statement at ℓ_3.

- $R_{\ell_8} : move(1, 8, 9)$

  The activity of the process inside the critical section is represented by the single transition that moves from ℓ_8 to ℓ_9. This represents the commitment that, differently from the non-critical section, the activity within the critical section must always terminate.

- $R_{\ell_{10}} : move(1, 10, 0) \wedge (y_1' = \mathrm{F})$

  This transition sets y_1 to F and moves to ℓ_0 to reiterate the body of the endless loop.

5 Invariance Properties

For an assertion p, we say that p is (generally) *valid*, and write $\models p$, if p is true on all possible states. All the known tautologies and theorems of the predicate calculus are obviously valid.

We say that the assertion p is *valid over the program P* (also described as being *P-valid*), and write $P \models p$, if p holds over all the P-accessible states. Clearly, if the assertion p is P-valid it is an *invariant* property of the program P.

In this section we present several proof rules that are adequate for proving the invariance of an assertion p over a program P, i.e., proving $P \models p$.

We will illustrate these rules by proving the main properties of Dekker's program

The main invariance property of Dekker's program can be expressed by the assertion $\neg(at\_\ell_8 \wedge at\_m_8)$. This assertion excludes states, in which both P_1 and P_2 execute in their critical sections at the same time. Thus, we have to prove

$$P \models \neg(at\_\ell_8 \wedge at\_m_8)$$

for Dekker's program.

Since most of our reasoning is done within the P-validity framework, we omit the $P \models$ prefix and simply write p to mean $P \models p$. The only exception to this convention are rules that deal at the same time with both general and P-validity, such as the IMP rule presented below.

IMP (Import) rule:

$$(\models p) \vdash (P \models p).$$

This rule states that if the assertion p is generally valid, it is in particular P-valid. It is used to import general validities into the P-validity framework.

MP (Modus Ponens) rule:

$$\{p \rightarrow q, p\} \;\vdash\; q.$$

This rule infers the P-validity of q from the P-validity of $p \rightarrow q$ and q.

The above two auxiliary rules are independent of the particular program analyzed. The following INV rule refers to the elements of the program, and is the main working tool for establishing invariance properties.

The rule uses a special case of a particular formula, to which we refer as the *verification condition* of the transition τ, relative to the assertions p and q. This formula has the form

$$(p \wedge \rho_\tau) \rightarrow q'$$

In this formula, 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. Let s and s' be two states. Since ρ_τ holds over the joint interpretation $\langle s, s' \rangle$ iff s' is a τ-successor of s, and q' states that q holds over s', it is not difficult to see that

> If the verification condition $(p \wedge \rho_\tau) \rightarrow q$ is P-valid, then every τ-successor of a p-state is a q-state.

We will standardly denote the verification condition of τ with respect to p and q by the triplet notation

$$\{p\}\tau\{q\}.$$

The INV (invariance) rule is given by

$$\begin{array}{lll} \text{INV} & \text{I1.} & \varphi \rightarrow p \\ & \text{I2.} & \Theta \rightarrow \varphi \\ & \text{I3.} & \{\varphi\}\tau\{\varphi\} \quad \text{for every } \tau \in \mathcal{T} \\ \hline & & p \end{array}$$

The INV rule uses an auxiliary assertion φ which, by premise I2, holds initially, and by premise I3 is propagated from each state to its successor. This shows that φ is an invariant of the program, that is, it holds continuously over all computations of P. Since, by I1, the assertion φ implies p, it follows that p is also an invariant of the program.

Example Consider the trivial program with a single state variable x, the pre-condition $x = 0$, and a single transition τ whose transition relation is given by ρ_τ : $x' = x + 1$. Observe that this program has a single infinite computation, given by

$$\langle x : 0 \rangle, \langle x : 1 \rangle, \langle x : 2 \rangle, \ldots$$

We wish to prove for this program the trivial invariance property

$$x \geq 0.$$

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

I1. $(x \geq 0) \rightarrow (x \geq 0)$

I2. $(x = 0) \rightarrow (x \geq 0)$

I3. $\Big((x \geq 0) \wedge (x' = x + 1)\Big) \rightarrow (x' \geq 0)$

Clearly all the three premises are generally valid, which establishes the invariance of $x \geq 0$.

We proceed to establish several invariants for Dekker's program, which together will yield the desired result.

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

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

Simple Invariants

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

$$\begin{array}{rcl} I_1 & : & (t = 1) \vee (t = 2) \\ I_2 & : & y_1 \equiv (at\_\ell_{3..5} \vee at\_\ell_{8..10}) \\ I_3 & : & y_2 \equiv (at\_m_{3..5} \vee at\_m_{8..10}) \end{array}$$

Invariant I_1 states that t can only assume the values 1 or 2. Invariant I_2 states that $y_1 = \mathrm{T}$ 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 I_3 states a similar property for P_2.

Let us see, for example, how an invariant such as I_1 is established. We apply the INV rule with $p = \varphi : (t = 1) \vee (t = 2)$. There are three premises to verify.

Premise I1 is trivial since $\varphi = p$ for our case. Premise I2 requires showing that Θ implies $(t = 1) \vee (t = 2)$. This is obvious, since Θ implies $t = 1$.

The premise that requires most attention is premise I3. Here we are called for writing a separate implication of the form $(\varphi \wedge \rho_\tau) \rightarrow \varphi'$, 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. 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 $(t = 1) \vee (t = 2)$, we should only be concerned with the transitions that modify t. These are:

ℓ_9 – sets t to 2.

m_9 – sets t to 1.

We thus conclude that the assertion $(t = 1) \vee (t = 2)$ is an invariant of the program.

As a slightly less trivial case, let us consider the invariant $I_2 : y_1 \equiv (at\_\ell_{3..5} \vee at\_\ell_{8..10})$. Let us concentrate on proving premise I3. While the expression $at\_\ell_{3..5}$ is defined in terms of the control variable π_1 that is modified by almost every transition in P_1, it is not difficult to see that the only transitions that affect the value of the expression $at\_\ell_{3..5}$ as a whole are those that either enter or exit the range ℓ_3–ℓ_5. Consequently, we only have to consider the following transitions:

ℓ_2 – This transition sets both y_1 and $at\_\ell_{3..5}$ to $\mathbf{T}$.

ℓ_5 – This transition sets both y_1 and $at\_\ell_{3..5} \vee at\_\ell_{8..10}$ to $\mathbf{F}$.

ℓ_7 – This transition sets both y_1 and $at\_\ell_{3..5}$ to $\mathbf{T}$.

ℓ_{10} – This transition sets both y_1 and $at\_\ell_{3..5} \vee at\_\ell_{8..10}$ to $\mathbf{F}$.

It is clear that these are the only transitions that modify any of the variables on which φ depends.

We conclude that $y_1 \equiv (at\_\ell_{3..5} \vee at\_\ell_{8..10})$ is an invariant assertion.

Proving Mutual Exclusion

Having established several simple invariants, we may proceed to establish the main invariance property of Dekker's program, namely, that of mutual exclusion, given by

$$I_4 \quad : \quad \neg[at\_\ell_{8..10} \wedge at\_m_{8..10}].$$

To show that the assertion I_4 is preserved under all transitions (i.e., premise I3 of the INV rule), 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 the assertion I_4 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 falsifying transitions are those that falsify one of the disjuncts while the other is already false. Consequently, we need only consider the cases:

- $\ell_3^{\mathbf{F}}$ while $at\_m_{8..10}$ – By I_3, $at\_m_{8..10}$ implies $y_2 = \mathbf{T}$ which disables $\ell_3^{\mathbf{F}}$.
- $m_3^{\mathbf{F}}$ while $at\_\ell_{8..10}$ – Similarly impossible.

This establishes the property of mutual exclusion for Dekker's program.
As we will see below, additional invariants are needed for the support of the response properties. 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

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

for assertions p and q. A sequence of states σ is said to satisfy the response formula $p \Rightarrow\!\Diamond\, q$ 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 by $P \models (p \Rightarrow\!\Diamond\, q)$, 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 $\Rightarrow\!\Diamond$ as the combination of the two operators $\Rightarrow$ and $\Diamond$ (see for example [MP89a]). 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 $\Rightarrow\!\Diamond$.

RFLX (Reflexivity) axiom:

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

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

TRNS (Transitivity) rule:

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

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.

MON (Monotonicity) rule:

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

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 formula.

DISJ (Disjunction) rule:

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

This rule combines the two response formulae, $p \Rightarrow\Diamond r$ and $q \Rightarrow\Diamond r$, into the formula $(p \vee q) \Rightarrow\Diamond r$. 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 cases.

The Basic Response Rule

The axiom and three rules listed above are independent of the particular program analyzed, and describe the 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 transition τ_h, to which we refer as the *helpful* transition. It can establish response formulae $p \Rightarrow\Diamond q$, such that a single activation of the transition τ_h is sufficient to achieve q. 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	R1.	$p \rightarrow (q \vee \varphi)$
	R2.	For every $\tau \in \mathcal{T}$ $\{\varphi\}\tau\{q \vee \varphi\}$
	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 the RESP rule on the program presented in Fig. 2

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

$$(at\_\ell_0 \wedge at\_m_0) \Rightarrow\Diamond (at\_\ell_2 \wedge at\_m_1),$$

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)

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

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

To prove the lemma we choose

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

and apply the RESP rule.

It is not difficult to see that p implies φ (establishing premise R1), provided we prove first the obvious invariant $at\_m_0 \rightarrow (x = 0)$. It is also clear

out x, y : **integer where** $x = 0$, $y = 0$

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

Figure 2: Program **terminating**.

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)

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

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

Lemma 3 (from ℓ_1 to ℓ_0)

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

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

Lemma 4 (from ℓ_1 to ℓ_2)

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

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

Lemma 5 (from $\ell_{0,1}$ to ℓ_2)

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

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

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

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

$$(at\_\ell_0 \wedge at\_m_0) \Rightarrow\!\Diamond\, (at\_\ell_2 \wedge at\_m_1)$$

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

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 uses induction over the integers to combine the helpful steps.

The following rule uses several *intermediate* assertions that hold at the positions lying 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 transition $\tau_i \in \mathcal{T}$, 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 the closest state that satisfies the goal q of the formula $p \Rightarrow\!\Diamond\, q$ which is the conclusion of the rule. Thus, the lower is the index, the closer we are to achieving the goal q. For a state satisfying φ_i, we refer to the index i as the *rank* of that state.

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

CHAIN

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

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 a transition τ_i from a φ_i-state, on which it is claimed to be helpful, 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 a rank decrease. Premise C4 claims that the helpful transition τ_i is always enabled on any state satisfying φ_i.

Assume that all the 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 state can either decrease or remain the same. It follows that there must exist some position $k \geq m$, beyond which the rank never decreases.

Assume that φ_i is the assertion holding 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.

Presentation of Proofs by Tables

When presenting a proof by the CHAIN rule, we usually do not conduct a detailed proof of each of the premises. Typically, it is enough to present the intermediate assertions $\varphi_1, \ldots, \varphi_r$, and their corresponding helpful transitions. To convince the reader that the effect of each transition on each intermediate assertion has indeed been considered, we may 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

$$(at\_\ell_0 \wedge at\_m_0) \Rightarrow\Diamond (at\_\ell_2 \wedge at\_m_1)$$

for the program presented in Fig. 2. Previously, we have proven this property by individual applications of the RESP rule. Let us now present a proof of the same property by the CHAIN rule.

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

The meaning of this table can be interpreted as follows. Let τ be a transition that appears in the (φ_i, φ_j) entry, i.e., the entry located on the row corresponding to φ_i and the column corresponding to φ_j. In that case we say that (the table implies that) φ_j is a *τ-successor* of φ_i, and also that *τ leads from φ_i to φ_j*. Note that in the row corresponding to φ_i, we underline each appearance of the transition τ_i that is helpful for φ_i.

For an assertion $\varphi_i, i > 0$, and a transition τ, let $\varphi_{j_1}, \ldots, \varphi_{j_t}$ be all the τ-successors of φ_i. This implies that the 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

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

The last column corresponding to F contains all the transitions that are claimed to be disabled under φ_i. The transitions appearing in this column in some row do not appear in any other entries on that row. Consequently, if we interpret an empty disjunction as F, the implied verification condition for such a transition τ is

$$\{\varphi_i\}\tau\{\mathrm{F}\} \quad \text{i.e.} \quad (\varphi_i \wedge \rho_\tau) \rightarrow \mathrm{F},$$

which implies that τ is disbaled on any φ_i-state. Note that it is not actually necessary to explicitly represent the F column. We can always reconstruct it by gathering into it all the transitions that do not appear in other entries.

We define a transition table to be *well formed* if for each τ that leads from $\varphi_i, i > 0$, to φ_j

- $i \geq j$, and
- if τ is helpful for φ_i, i.e., $\tau = \tau_i$, then $i > j$.

To confirm the correctness of a proof presented by a transition table, one may

- prove premise C1, i.e.,

$$p \rightarrow \bigvee_{i=0,\ldots,r} \varphi_i,$$

- check that the table is well formed,
- prove all the verification conditions implied by the table, and finally
- for each $i = 1, \ldots, r$, prove premise C4, i.e.,

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

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 uniform condition implied 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.

Thus, the table above presents a proof of the property

$$(at\_\ell_0 \wedge at\_m_0) \Rightarrow \Diamond (at\_\ell_2 \wedge at\_m_1).$$

Presentations of Proofs by Diagrams

An alternative but equivalent presentation of proofs by the CHAIN rule 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 will 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 $\varphi_i, i > 0$ 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 other are not, we still must 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. 3 we present a flat diagram representing the transition table presented above.

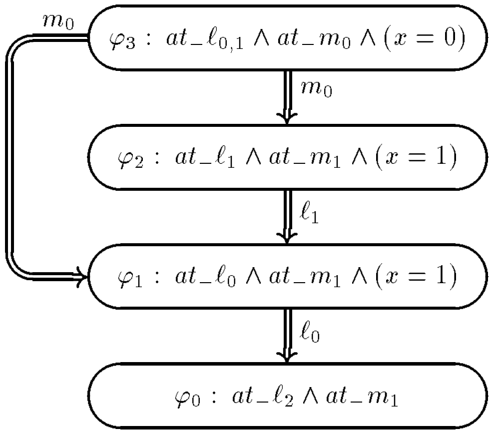

Figure 3: Flat proof diagram for program **terminating**.

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.

A diagram is defined to be *well formed* if for every φ_i, φ_j, and τ, such that φ_j is a τ-successor of φ_i

- $i \geq j$, and
- if τ is helpful for φ_i, i.e., $\tau = \tau_i$, then $i > j$.

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 the CHAIN rule. 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. The 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

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

This interpretation is intended to economize on the number of drawn edges. Thus, transitions that only lead from an assertion to itself need not be represented explicitly. As a result, the information provided by diagrams is somewhat less detailed than that of transitions tables. In particular, diagrams do not distinguish between transitions that are disabled on φ_i and those that are enabled on φ_i but only lead from φ_i to itself. Note that transitions τ that disabled on φ_i-states trivially satisfy the verification condition $\{\varphi_i\}\tau\{\varphi_i\}$.

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.

Distribution of Common Edges

We allow edges to depart from or to 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. 4.

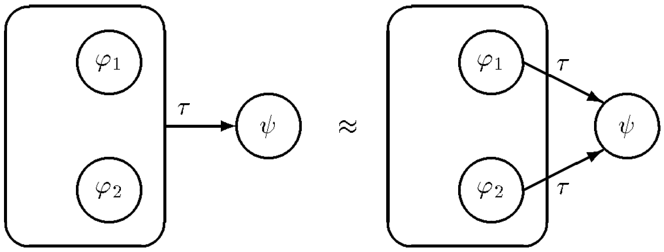

Figure 4: Distribution of Departing Edges

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. 4 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 to a composite node as though it arrived to each of its descendants. This is represented by the graphical equivalence depicted in Fig. 5.

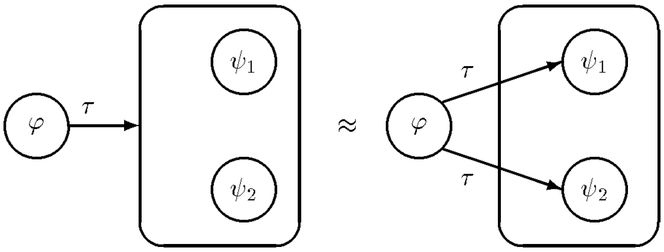

Figure 5: Distribution of Arriving Edges

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 6

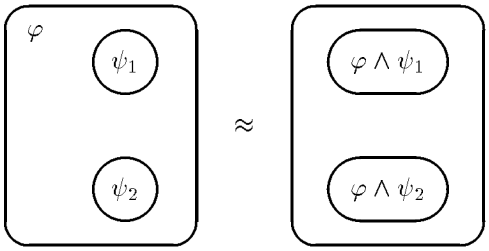

Figure 6: Common Conjunct

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. To check whether a given structured diagram is well formed, or to list the verification conditions implied by such a diagram, it is not necessary to actually construct the equivalent flat diagram.

In Fig. 7 we present a structured version of the proof diagram previously presented in Fig. 3.

This diagram contains a single composite node, whose descendants are (labeled by) φ_2 and φ_1. The double edge labeled by m_0, which connects φ_3 to this composite node, represent the *two* edges connecting φ_3 to the nodes φ_2 and φ_1 in the diagram of Fig. 3. Note also the common conjunct $at\_m_0 \wedge (x = 0)$, that has been factored out of the two descendants, and now labels the composite node.

Proving Accessibility

The main response property one usually wishes to prove for mutual exclusion programs is that of *accessibility*, by which whenever a process departs from its non-critical section it is guaranteed to eventually reach the critical section.

This property can be expressed by the response formula

$$at\_\ell_2 \Rightarrow\Diamond (at\_\ell_8).$$

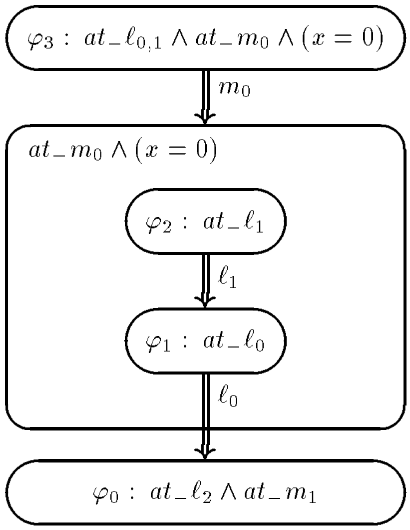

Figure 7: Structured proof diagram for program **terminating**.

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,

Lemma A

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

Lemma B

$$\Big(at\_\ell_{3..7} \wedge (t = 1)\Big) \Rightarrow\Diamond (at\_\ell_8).$$

Obviously, the difficult part of the protocol is the loop at ℓ_3. Lemma B shows 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

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

is presented in the diagram of Fig. 8.

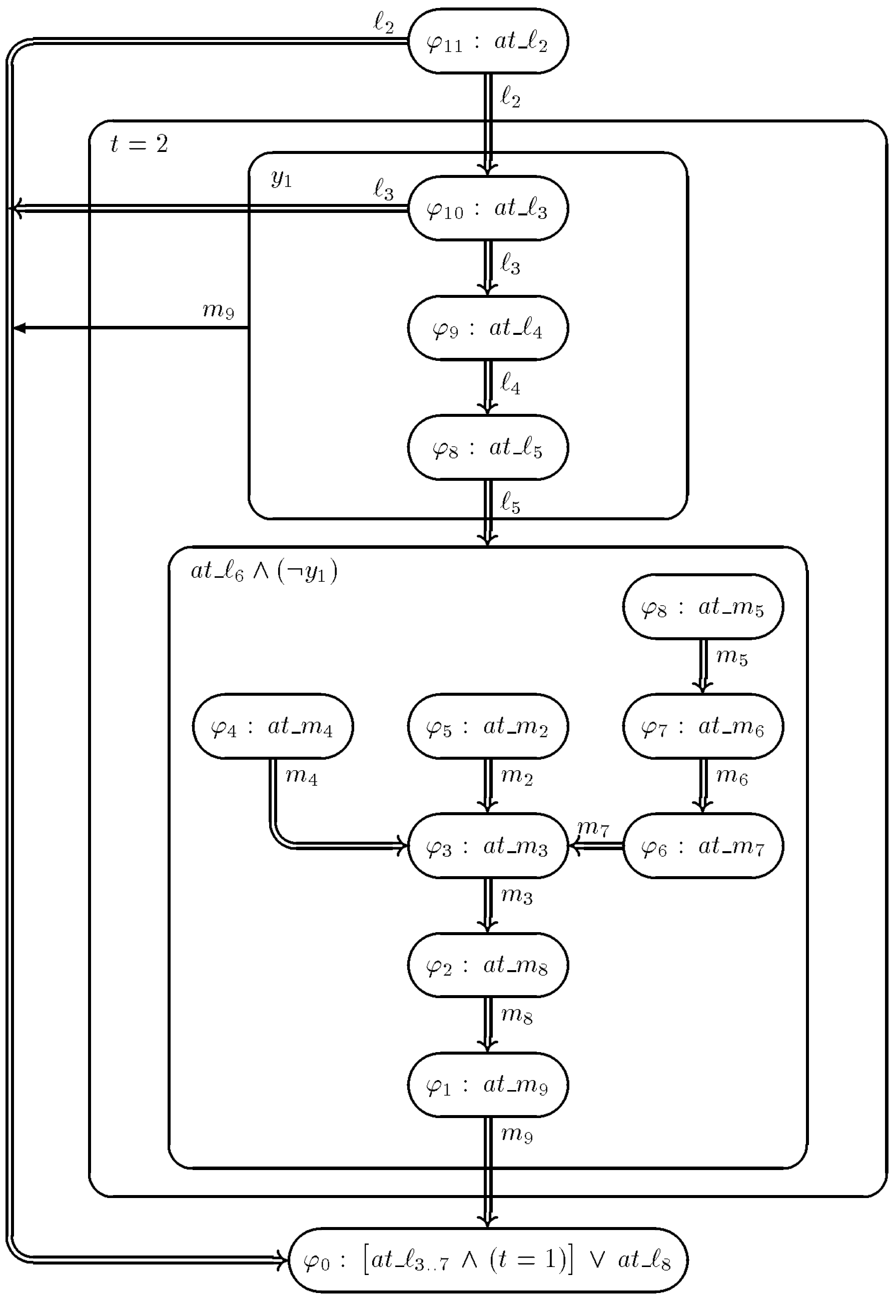

Figure 8: Proof diagram for lemma A.

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 $\ell_3^{\mathbf{F}}$ and reach ℓ_8, which again is part of the goal, or proceed to ℓ_4. From ℓ_4, we proceed to ℓ_5 since $t = 2$, and then to ℓ_6 while resetting y_1 to $\mathtt{F}$. 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_2–m_9, tracing their possible flow under the relatively stable situation of $t = 2$, $y_1 = \mathtt{F}$. 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 $y_1 = \mathtt{F}$ 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 $y_1 = \mathtt{F}$ 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 from P_1.

More Invariants

From the discussion above, it follows that if the algorithm is correct, and guarantees accessibility to both processes, then it must satisfy the following invariant:

$$I_5 \quad : \quad \Big(at\_\ell_{4..6} \wedge (t = 2)\Big) \;\rightarrow\; at\_m_{2..9}.$$

The proof of lemma A only needs $\Big(at\_\ell_6 \wedge (t = 2)\Big) \rightarrow at\_m_{2..9}$, but it is easy to see that if we had $at\_\ell_{4,5} \wedge (t = 2) \wedge (\neg at\_m_{2..9})$, 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

$$I_6 \quad : \quad \Big(at\_m_{4..6} \wedge (t = 1)\Big) \;\rightarrow\; at\_\ell_{2..9}.$$

Let us check the verification conditions for the assertion appearing in I_5.

Clearly I_5 is implied by Θ.

There are three transitions that may potentially falsify the assertion I_5.

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

  Sets t to 1 which makes $t = 2$ false and hence preserving the assertion.

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

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

- $\ell_3^{\mathbf{T}}$ while $t = 2$ — making $at\_\ell_{4..6}$ true.

  This is possible only if $y_2 = \mathtt{T}$ which, by I_3, implies $at\_m_{3..10}$. 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 $t = 2$. 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 I_4, is still a part of the critical section and is therefore exclusive from ℓ_9 that is the only statement capable of changing t to 2.

This leads us to the invariant

$$I_7 \quad : \quad at\_m_{10} \;\rightarrow\; (t = 1)$$

and its symmetric counterpart

$$I_8 \quad : \quad at\_\ell_{10} \;\rightarrow\; (t = 2).$$

To prove I_7, we should inspect two transitions:

- Transition m_9 — making at_m_{10} true.

  Sets t to 1.

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

  Impossible due to I_4.

This establishes I_7 and similarly I_8. Having I_7 we can use it to show that the last transition considered in the proof of I_5, namely $\ell_3^{\mathbf{T}}$ while $t = 2$, implies $at\_m_{3..9}$ (excluding at_m_{10}), which establishes I_5.

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 $\Big(at\_\ell_{3..7} \wedge (t = 1)\Big) \Rightarrow\Diamond(at\_\ell_8)$. The proof is presented in the diagram of Fig. 9.

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 $y_1 = \mathtt{T}$, the diagram recognizes the following cases:

- 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 $\ell_3^{\mathbf{F}}$ first, exiting from the while loop to the critical section. Alternately, P_2 may perform first one of m_2, 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 general situation, it is P_2 which is the helpful process and we have to trace its progress.

On the other hand, when $y_2 = \text{F}$, 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: $m_{0..2} \vee m_7$ 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.

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.

[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.

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

[MP89a] 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.

[MP89b] Z. Manna and A. Pnueli. Completing the temporal picture. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, pages 534–558. Lec. Notes in Comp. Sci. 372, Springer, 1989.

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

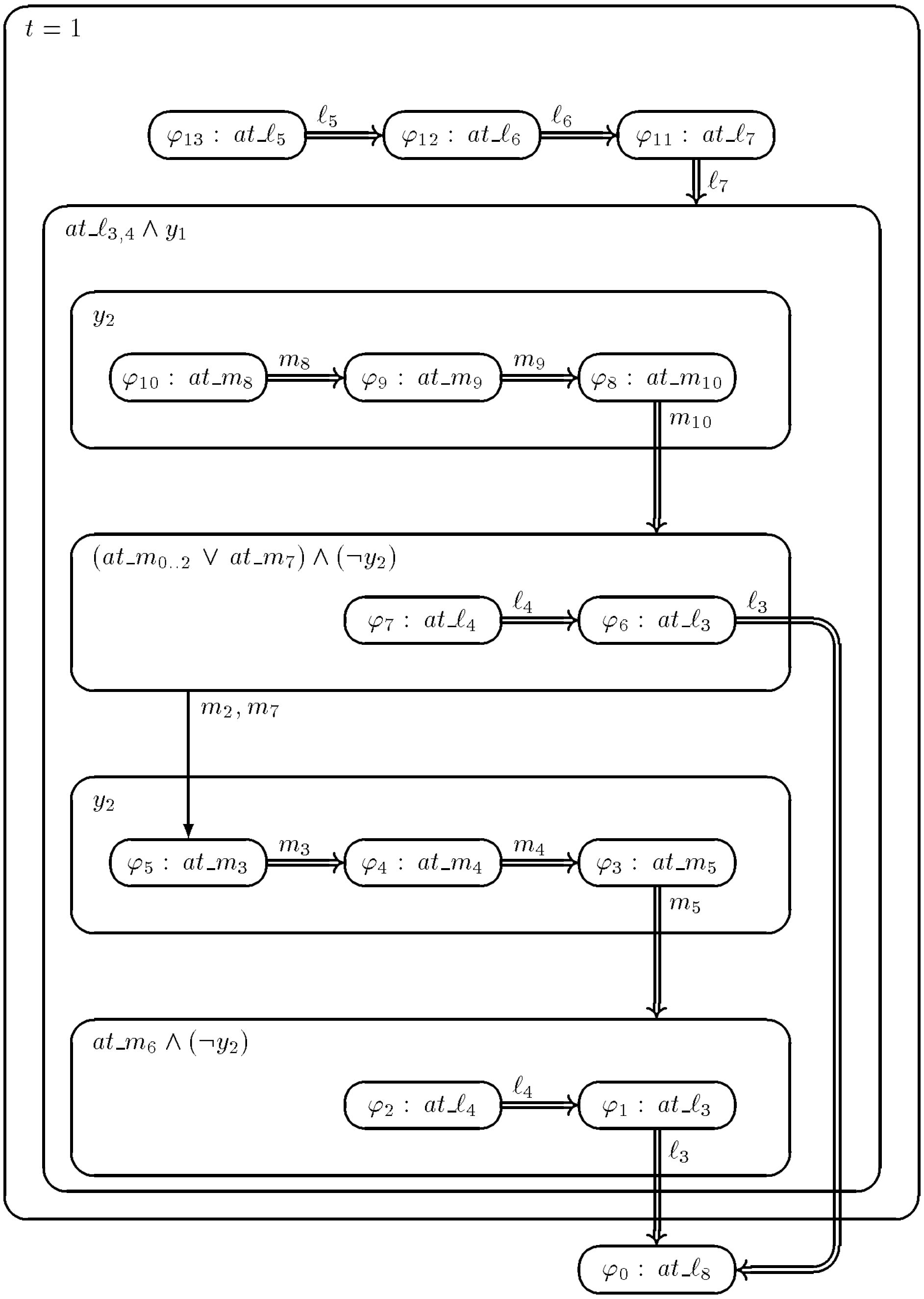

Figure 9: Proof diagram for lemma B.