Temporal Verification of Simulation and Refinement *

Yonit Kesten[1] Zohar Manna[2] Amir Pnueli[1]

[1] Department of Applied Mathematics and Computer Science
Weizmann Institute, Rehovot, Israel
e-mail: amir@wisdom.weizmann.ac.il
[2] Department of Computer Science, Stanford University
Stanford, CA 94305
e-mail: manna@cs.stanford.edu

Abstract. The paper presents temporal logic methods for proving simulation and refinement relations between programs. After introducing the notions of fair transition systems and the specification language of temporal logic, we present proof rules for verifying properties of programs. We then define the relations of simulation and refinement between programs and relate them to inclusion relations between computations and observations of the compared systems.

We then show that these semantic definitions can be formalized in temporal logic by the use of the temporal and observational semantics formulas. This representation expresses simulation and refinement as implications between a pair of such formulas. We provide proof rules, based on the rules for verifying program properties. The proof rules are illustrated on several simple examples. Towards the end of the first part, we recognize the usefulness of having a stuttering-robust version of temporal logic.

The second part of the paper presents a proposed temporal logic, called TLR, which is insensitive to stuttering. This logic is interpreted over sequences of sampling points, alternating between persistent and transient sample points. This logic possesses an idempotent next-time operator, which gives some insight into its stuttering robustness. We present a decision procedure and a complete axiomatic system for the propositional version of TLR. It is shown that, if all system variables are taken to be left-continuous, then most of the rules of regular temporal logic are still sound for TLR.

Finally, we present a stronger proof rule for refinement, and illustrate its use to prove refinement of two programs that cannot be done within the regular temporal logic framework.

* This research was supported in part by the National Science Foundation under grant CCR-92-23226, by the Defense Advanced Research Projects Agency under contract NAG2-703, by the United States Air Force Office of Scientific Research under contract F49620-93-1-0139, by the European Community ESPRIT Basic Research Action Project 6021 (REACT), and by the France-Israel project for cooperation in Computer Science.

Keywords: temporal logic, verification of properties, simulation, refinement, stuttering, invariance, response, reactivity.

Table of Contents

1 Introduction

The central task in rigorous program development is the construction of a program that satisfies a given specification. We can say that the program *implements* the given specification, and refer to this task as *program synthesis*. In the domain of reactive systems, this task is particularly difficult and, so far, only special cases have been considered, and some heuristics proposed for more general cases.

A somewhat more modest task, which can provide the formal justification for various development and synthesis steps is that of *verification*. Here, we are given a proposed implementation and have only to check that the implementation meets its specification and is, therefore, an admissible implementation of the specification.

A language that is widely used for specifying reactive systems is *temporal logic* [MP91b]. There are two distinct styles of specification in temporal logic. One style specifies a system by a list of properties, each expressed by a temporal logic formula, that the system should satisfy. Some of these properties may be *safety* properties stating, for example, that some state property such as mutual exclusion is continuously maintained. Other properties may be *liveness* properties stating, for example, that every request is eventually responded to. We may refer to this style as specification by *property list*, or *requirement specification*, since it often corresponds to the natural language document prepared at the initial stages of a project inception, listing the various requirements expected from the contemplated system. This is the specification style studied in [MP91b].

If a requirement specification is given by the property list

$$\Phi : \quad \{\varphi_1, \ldots, \varphi_m\},$$

then, to show that a program P satisfies the specification Φ, we should prove

$$P \vDash \bigwedge_{i=1}^{m} \varphi_i,$$

or equivalently,

$$P \vDash \varphi_i, \qquad \text{for every } i = 1, \ldots, m.$$

A different approach to the specification of systems within temporal logic is their description as an abstract fair transition system. In this approach, one identifies a fair transition system consisting of a set of system variables, a set of transitions which identify the possible state changes, and two fairness sets of transitions which connect enableness of transitions with requirements for the frequency of their activation. A fair transition system S specifies a program by requiring that the set of computations of a program P is a subset of the set of computations of S. To ensure that the specification S does not impose an implementation bias on the program P, we identify a subset of S's system variables $\mathcal{O}$ as *observable*. Relative to this subset, we define an *observation* of S to be a model that agrees with a computation of S on all variables in $\mathcal{O}$. Program P is said to *implement* or *satisfy* specification S, if every computation of P is an observation of S.

We refer to this style of specification as *action specification*.

As shown in [MP83], for each such system S (and program P, which can also be viewed as a fair transition system), there exists a temporal formula sem_S, to which we refer as the *temporal semantics* of system S. Thus, to prove that a program P satisfies an action specification given by the fair transition system S, we should prove

$$P \models sem_S.$$

For the cnase that not all system variables in S are observable, we hide the unobservable variables by existential quantification.

We can view the style of specification consistently advocated by Lamport [Lam83a], and serving as the main motivation for the temporal logic TLA [Lam91], as a generalization of the action specification style. The TLA style is more general in that the liveness part is not restricted to expressing the fairness requirements of a particular fair transition system S, but may include any arbitrary conjunction of liveness formulas.

The temporal logic we have been studying contains a richer set of operators and can conveniently express both the property style and the action style of specifications. Over the years, we have been developing a detailed proof system for verifying that programs meet their specifications [MP84], [MP91a], [MP94]. This proof system is based on the classification of properties expressible by temporal logic, and the design of proof rules for each of the classes. In the majority of applications, this proof system has been used for verifying properties of programs.

In this paper we demonstrate that the same proof system, with very minor modifications, can be used to prove that a program satisfies a given action specification. As a first step, this establishes that the temporal language we advocate and its associated proof system can be used for specification and verification in both specification styles.

However, a much more important observation is that proving that one system implements an action specification, which essentially describes the behavior of another, more abstract, system, provides the key to proving refinement between systems, using temporal logic.

A development of a complex reactive system may often proceed through several intermediate stages on the way from a top-level abstract specification to a final efficient running version.

$$S_0 \sqsupseteq_{L_0} S_1 \sqsupseteq_{L_1} \cdots S_{k-1} \sqsupseteq_{L_{k-1}} S_k$$

In this development sequence, each system S_{i+1} refines its predecessor S_i, abstracting away the variables in L_i which can be viewed as local to S_i, $i = 0, \ldots, k$. The top-level specification S_0 may be given in either a property or an action style. However, typically all other stages, $S_1, \ldots, S_k$, can be described as fair transition systems. Consequently, for each $i = 1, \ldots, k-1$, S_i serves as an implementation of specification S_{i-1} and as an (action style) specification of implementation S_{i+1}.

A proof rule for establishing

$$S_{i+1} \models \exists L_i : sem_{S_i}$$

provides a formal justification of such a development sequence.

Many formalisms have been proposed for proving refinement between systems. We refer the reader to the collection [dBdRR90] for a comprehensive coverage of the main approaches to refinement proofs. The paper [AL88] presents a semantic framework for the study of refinement and proves, under some assumptions, completeness of a proof method based on refinement mapping. The note [Lam92] shows how these semantic ideas can be worked out within the TLA framework. In this paper, we show how to prove simulation and refinement within the framework of the temporal logic of [MP91b], using the proof system of [MP91a].

The temporal logic we have been using can specify properties that are not invariant under stuttering. In some cases this can be an advantage. However, when we study refinements, invariance under stuttering becomes a necessary requirement. In the second half of the paper, we present a new temporal logic, called TLR, which is stuttering-robust, yet possesses a general *next-time* operator. In comparison, TLA also admits a limited next-time operator (prime) but achieves stuttering robustness by restricting the contexts in which the prime operator may appear.

A preliminary version of the logic TLR appeared in [Pnu92], where it was given a dense-time semantics. The presentation here is based on a discrete semantics, similar to the standard semantics of other temporal logics.

The paper is organized as follows: in section 2, we present the preliminary notions of fair transition systems, a simple programming language for presenting examples, and the temporal logic we will use in the first half of the paper. In section 3, we review the proof rules of our proof system that are used for proving properties, and show how they can be extended to prove properties involving *local formulas*. In section 4, we define the notion of one system simulating another, and present a formal approach to verifying the simulation relation between two systems. In section 5 we show how the notion of refinement generalizes simulation and present a formal approach to verifying that one system refines another. The section contains several nontrivial examples, illustrating the usefulness of refinement in systematic program development.

The second part of the paper, consisting of sections 6–10, introduces the logic TLR, which is a temporal logic designed to deal with stuttering. Section 6 presents the motivation for introducing yet another temporal logic and the expected benefits for a logic that is stuttering-robust. Section 7 presents the syntax and semantics of TLR. Section 8 describes an algorithm for checking whether a given propositional TLR formula is satisfiable. An axiomatic system for propositional TLR is presented in section 9 and proved to be sound and complete. Finally, in section 10 we demonstrate the use of TLR for proving refinement relations between two programs that are not equivalent in the semantics induced by stuttering sensitive temporal logic.

The semantics given to TLR in section 7 is discrete and represents computations of programs by sequences of states. For completeness, we include in the appendix an alternative dense semantics. In spite of the radically different nature of the two semantics they are equivalent in the sense that every TLR formula is valid in one of the semantics iff it is valid in the other.

2 Preliminaries

We assume a universal set of variables $\mathcal{V}$, called the *vocabulary*. Variables in $\mathcal{V}$ are typed, where the type of a variable, such as *boolean*, *integer*, etc., indicates the domain over which the variable ranges.

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

2.1 Computational Model: Fair Transition System

As a computational model for reactive systems we take *fair transition systems*. Such a system consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of *system 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.
- Θ : The *initial condition*. This is an assertion characterizing all the initial states, i.e., states at which a computation of the program can start. A state is defined to be *initial* if it satisfies Θ. It is required that Θ be satisfiable, i.e., there exists at least one state satisfying Θ.
- $\mathcal{T}$: A set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \mapsto 2^{\Sigma}$, mapping each state $s \in \Sigma$ into a (possibly empty) set of τ-successor states $\tau(s) \subseteq \Sigma$.
  A transition τ is *enabled* on s iff $\tau(s) \neq \emptyset$. Otherwise τ is *disabled* on s.
  The function associated with a transition τ is represented by an assertion $\rho_\tau(V, V')$, called the *transition relation*, which relates a state $s \in \Sigma$ to its τ-successor $s' \in \tau(s)$ by referring to both unprimed and primed versions of the system variables. An unprimed version of a system variable refers to its value in s, while a primed version of the same variable refers to its value 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). The requirement of justice for $\tau \in \mathcal{J}$ disallows a computation in which τ is continually enabled beyond a certain point but taken only finitely many times.
- $\mathcal{C} \subseteq \mathcal{T}$: A set of *compassionate* transitions (also called *strongly fair* transitions). The requirement of compassion for $\tau \in \mathcal{C}$ disallows a computation in which τ is enabled infinitely many times but taken only finitely many times.

The transition relation $\rho_\tau(V, V')$ identifies state s' as a τ-*successor* of state s if

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

where $\langle s, s' \rangle$ is the joint interpretation which interprets $x \in V$ as $s[x]$, and x' as $s'[x]$.

The enableness 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 agrees with s' on the values of all system variables.

Let $\mathcal{S}$ be a transition system for which the above components have been identified. We define a *computation* of $\mathcal{S}$ to be an infinite sequence of states $\sigma : s_0, s_1, s_2, ...,$ satisfying the following requirements:

- *Initiation:* 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 $\tau \in \mathcal{T}$. In this case, we say that the transition τ is *taken* at position j in σ.
- *Justice:* For each $\tau \in \mathcal{J}$, it is not the case that τ is continually enabled beyond some position in σ but taken at only finitely many positions in σ.
- *Compassion:* For each $\tau \in \mathcal{C}$, it is not the case that τ is enabled on infinitely many states of σ but taken at only finitely many positions in σ.

2.2 A Simple Programming Language: Syntax

To present programs, we introduce a simple concurrent programming language (SPL) in which processes communicate by shared variables. The following is a list of some of the statements together with an explanation of their intended meaning. We present only the statements that are used in this paper; the reader is referred to [MP91b] for a description of the full language.

- *Assignment*: For a variable y and an expression e of compatible type,

  $$y \ := \ e$$

  is an *assignment* statement.
- *Await*: For a boolean expression c,

  await c

  is an *await* statement. We refer to condition c as the *guard* of the statement. The statement **await** c can be executed only when c evaluates to T.
- *Concatenation*: For statements $S_1, \ldots, S_k$,

  $$S_1; \cdots; S_k$$

  is a *concatenation* statement. Its intended meaning is sequential composition. The first step in an execution of $S_1; \cdots; S_k$ is the first step in an execution of S_1. Subsequent steps continue to execute the rest of S_1, and when S_1 terminates, proceed to execute $S_2, S_3, \ldots, S_k$.

  In a program presented as a multi-line text, we often omit the separator ';' at the end of a line.

- *While*: For a boolean expression c and a statement S,

  while c **do** S

  is a *while* statement. Its execution begins by evaluating c. If c evaluates to F, execution of the statement terminates. Otherwise, subsequent steps proceed to execute S. When S terminates, c is tested again.

  We introduce the notation

  loop forever do S

  as an abbreviation for

  while T **do** S.
- *Request*: For an integer variable r,

  request r

  is a *request* statement. This statement can be executed only when r has a positive value. When executed, it decrements r by 1.
- *Release*: For an integer variable r,

  release r

  is a *release* statement. Execution of this statement increments r by 1.

  The request and release statements are called *semaphore* statements.

Schematic Statements

The following statements provide a schematic representation of segments of code that appear in programs for solving the mutual exclusion problem. Typically, we are not interested in the internal details of this code but only in its behavior with respect to termination.

- *Noncritical*: The statement

  noncritical

  is used to represent the noncritical activity in programs for mutual exclusion. It is not required that this statement terminate.
- *Critical*: The statement

  critical

  represents the critical activity in programs for mutual exclusion, where coordination between the processes is required. It is required that this statement terminate.

It is assumed that none of these statements modifies any program variables.

Statements in a program may be labeled. Labels serve as unique names for the statements and also as possible sites of control.

Programs

A program P has the form

$$P :: \left[\text{declaration};\ \left[P_1 :: [\ell_0^1\colon S_1;\ \ \widehat{\ell}^1 :] \,\|\, \cdots \,\|\, P_m :: [\ell_0^m\colon S_m;\ \ \widehat{\ell}^m :]\right]\right],$$

where $P_1 :: [\ell_0^1\colon S_1;\ \widehat{\ell}^1 :], \ldots, P_m :: [\ell_0^m\colon S_m;\ \widehat{\ell}^m :]$ are *named processes*. The names of the program and of the processes are optional, and may be omitted. The *body* $[\ell_0^i\colon S_i;\ \widehat{\ell}^i :]$ of process P_i consists of a (possibly labeled) statement

$\ell_0^i : S_i$ and an exit label $\widehat{\ell^i}$, which is where control resides after execution of S_i terminates. Label $\widehat{\ell^i}$ can be viewed as labeling an empty statement following S_i.

A declaration consists of a sequence of *declaration statements* of the form

variable, ..., variable: type **where** φ.

Each declaration statement lists several variables that share a common type, i.e., the domain over which the variables range, and identifies their type. We use *basic types* such as **integer**, **character**, etc., as well as *structured types*, such as **array**, **list**, and **set**.

The optional assertion φ imposes constraints on the initial values of the variables declared in this statement.

Let $\varphi_1, \ldots, \varphi_n$ be the assertions appearing in the declaration statements of a program. We refer to the conjunction $\varphi : \varphi_1 \wedge \cdots \wedge \varphi_n$ as the *data-precondition* of the program.

Example (Peterson's algorithm) Consider program PET, presented in Figure 1, which implements Peterson's algorithm for mutual exclusion without semaphores. This program will serve as our running example for most of the verification diagrams we intend to present.

The basic mechanism protecting the accesses to 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_3 and m_3 respectively, at about the same time, with both $y_1 = y_2 = \text{T}$. 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 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 step taken by the process. 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 $s \neq i$ implies that $s = j$, which means that the competitor P_j was the *last* to assign a value to s. Therefore P_i should have priority. ▟

2.3 Semantics of the Programming Language

The semantics of the simple programming language is defined by a translation of programs into fair transition systems. This is done by showing how to construct the five components of a fair transition system for a given program.

Consider a program P given by

$$\Big[\text{declaration; } \big[P_1 :: [\ell_0^1\text{: } S_1;\ \widehat{\ell^1} :] \parallel \cdots \parallel P_m :: [\ell_0^m\text{: } S_m;\ \widehat{\ell^m} :]\big]\Big].$$

Let L_i denote the set of *locations* (labels) of process P_i, $i = 1, \ldots, m$, and $L_P = L_1 \cup \cdots \cup L_m$.

We will show how to define a fair transition system $\mathcal{S}_P$ corresponding to program P.

$$
\begin{array}{c}
\textbf{local}\ \ y_1, y_2 : \textbf{boolean where } y_1 = \text{F}, y_2 = \text{F} \\
\qquad\quad s \qquad : \textbf{integer where } s = 1
\end{array}
$$

$$
P_1 :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } (y_1, s) := (\text{T}, 1) \\
\ell_3\text{: } \textbf{await } \neg y_2 \lor s = 2 \\
\ell_4\text{: } \textbf{critical} \\
\ell_5\text{: } y_1 := \text{F}
\end{array}\right]
\end{array}\right]
\quad \| \quad
P_2 :: \left[\begin{array}{l}
m_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
m_1\text{: } \textbf{noncritical} \\
m_2\text{: } (y_2, s) := (\text{T}, 2) \\
m_3\text{: } \textbf{await } \neg y_1 \lor s = 1 \\
m_4\text{: } \textbf{critical} \\
m_5\text{: } y_2 := \text{F}
\end{array}\right]
\end{array}\right]
$$

Fig. 1. Program PET (Peterson's algorithm for mutual exclusion).

System Variables and States

The *system variables* V for system $\mathcal{S}_P$ consist of the *data variables* $Y : \{y_1, \ldots, y_n\}$ that are declared at the head of the program, and the *control variables* $\pi_1, \ldots, \pi_m$, one for each process[3]. The data variables Y range over their respectively declared data domains. Each control variable π_i, $i = 1, \ldots, m$, ranges over indices of the locations in process P_i. Thus, a value $\pi_i = j$ in a state indicates that the current site of control within process P_i is ℓ_j, or m_j.

For example, the system variables for program PET are $V : \{\pi_1, \pi_2, y_1, y_2, s\}$, where y_1 and y_2 range over the booleans, s ranges over the integers, π_1 and π_2 range over the integer interval $[0..5] = \{0, 1, \ldots, 5\}$.

States assign interpretations to all variables in the vocabulary, including the system variables. For example, restricting our attention to the interpretation given to the system variable, an initial state of program PET is

$$\langle \pi_1 : 0,\ \pi_2 : 0,\ y_1 : \text{F},\ y_2 : \text{F},\ s : 1\rangle.$$

For a statement $\ell_j : S$ belonging to process P_i, we use the abbreviation

$$at\_\ell_j : \quad \pi_i = j.$$

For example, the state $\langle \pi_1 : 0,\ \pi_2 : 0,\ y_1 : \text{F},\ y_2 : \text{F},\ s : 1\rangle$, which is initial for program PET, satisfies at_ℓ_0 and at_m_0 but does not satisfy at_ℓ_1 or at_m_1.

The Initial Condition

Let φ denote the *data-precondition* of program P. We define the *initial condition* Θ for $\mathcal{S}_P$ as

$$\Theta: \quad \pi_1 = 0 \ \land\ \cdots\ \land\ \pi_m = 0\ \land\ \varphi.$$

This implies that the first state in an execution of the program begins with the control variables set to the initial locations of the processes, and the data variables satisfying the data precondition.

For example, the initial condition for program PET is given by

$$\Theta: \quad \pi_1 = 0\ \land\ \pi_2 = 0\ \land\ y_1 = \text{F}\ \land\ y_2 = \text{F}\ \land\ s = 1.$$

[3] Here we deviate from the representation of programs in [MP91b] which uses a single control variable π, ranging over sets of locations.

Transitions

To facilitate the expression of transition relations, we introduce the abbreviation

$$move(\ell_j, \ell_k): \quad \pi_i = j \ \wedge\ \pi_i' = k \ \wedge\ \bigwedge_{r \neq i} (\pi_r' = \pi_r),$$

for locations ℓ_j and ℓ_k in process P_i. This formula expresses the situation that control within P_i is currently at ℓ_j and moves in the next state to location ℓ_k.

We also use the abbreviation

$$pres(U): \quad \bigwedge_{u \in U} (u' = u),$$

to denote that all variables in the set U are preserved by the transition.

To ensure that every state has some transition enabled on it, we uniformly include the idling transition τ_I in the transition system corresponding to each program. The transition relation for τ_I is

$$\rho_I\colon\ V' = V.$$

We proceed to define the transition relations for the transitions associated with each of the previously introduced statements.

- *Assignment*: Consider the statement $[\ell : y := e\,;\ \widehat{\ell} : \ldots]$, where $\widehat{\ell}$ is the location to which control moves after statement terminates. We refer to $\widehat{\ell}$ as the *post-location* of statement ℓ.
  With this statement we associate a transition τ_ℓ, with the transition relation
  $$\rho_\ell\colon \quad move(\ell, \widehat{\ell}) \ \wedge\ y' = e \ \wedge\ pres(Y - \{y\}).$$
  The last conjunct claims that all data variables, excluding y, retain their values over the transition τ_ℓ.
- *Await*: With the statement $[\ell\colon \textbf{await } c\,;\ \widehat{\ell} : \ldots]$, we associate a transition τ_ℓ, with the transition relation
  $$\rho_\ell\colon \quad move(\ell, \widehat{\ell}) \ \wedge\ c \ \wedge\ pres(Y)$$
  The transition τ_ℓ is enabled only when control is at ℓ and the condition c holds. When taken, it moves from ℓ to location $\widehat{\ell}$.
- *While*: With the statement $\Big[\ell\colon \big[\textbf{while } c \textbf{ do } [\widetilde{\ell}\colon \widetilde{S}]\big];\ \widehat{\ell} : \ldots,\Big]$ we associate a transition τ_ℓ, with the transition relation
  $$\rho_\ell\colon \quad \begin{pmatrix} c \ \wedge\ move(\ell, \widetilde{\ell}) \\ \vee \\ \neg c \ \wedge\ move(\ell, \widehat{\ell}) \end{pmatrix} \ \wedge\ pres(Y).$$
  According to ρ_ℓ, when c evaluates to T control moves from ℓ to $\widetilde{\ell}$, and when c evaluates to F control moves from ℓ to $\widehat{\ell}$. Note that, in this context, the post-location of $\widetilde{S}$ is ℓ. Note also that the enabling condition of τ_ℓ is at_ℓ, which does not depend on the value of c.
- *request*: With the statement $[\ell\colon \textbf{request}(r);\ \widehat{\ell}\colon \ldots]$, we associate a transition τ_ℓ, with the transition relation
  $$\rho_\ell\colon \quad move(\ell, \widehat{\ell}) \ \wedge\ r > 0 \ \wedge\ r' = r - 1 \ \wedge\ pres(Y - \{r\}).$$
  Thus, this statement is enabled when control is at ℓ and r is positive. When executed it decrements r by 1.
- *release*: With the statement $[\ell\colon \textbf{release}(r);\ \widehat{\ell}\colon \ldots]$, we associate a transition τ_ℓ, with the transition relation

$$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge\ r' = r + 1 \ \wedge\ pres(Y - \{r\}).$$

This statement increments r by 1.

- *Noncritical*: With the statement [ℓ: **noncritical**; $\widehat{\ell}$:...], we associate a transition τ_ℓ, with transition relation

  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge\ pres(Y).$$

  Thus, the only observable action of this statement is to terminate. The situation that execution of the noncritical section does not terminate is modeled by a computation that does not take transition τ_ℓ. This is allowed by excluding τ_ℓ from the justice set.
- *Critical*: With the statement [ℓ: **critical**; $\widehat{\ell}$:...], we associate a transition τ_ℓ, with transition relation

  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge\ pres(Y).$$

  The only observable action of the critical statement is to terminate.

Thus, the fair transition system corresponding to program PET has the transitions

$$\tau_I,\ \tau_{\ell_0},\ \ldots,\ \tau_{\ell_5},\ \tau_{m_0},\ \ldots,\ \tau_{m_5}.$$

Transition τ_{ℓ_2}, for example, has the transition relation

$$\rho_{\ell_2}: \quad move(\ell_2, \ell_3) \ \wedge\ s' = 1 \ \wedge\ y_1' = \mathrm{T} \ \wedge\ y_2' = y_2.$$

Justice and Compassion

For the simplistic programming language we have presented, the justice and compassion requirements are straightforward.

- *Justice*: As the *justice* set, we take the set of all transitions, excluding the idling transition τ_I and transitions associated with the **noncritical** statement.
- *Compassion*: As the *compassion* set, we take the set of all semaphore **request** transitions in the program. This will suffice for the examples presented in this paper. The programs presented in [MP91b] use additional statements, such as communication statements, that give rise to additional compassionate transitions.

  Note that the transitions associated with **request** statements are both just and compassionate.

This concludes the definition of the transition system $\mathcal{S}_P$ corresponding to a program P. We refer to computations of $\mathcal{S}_P$ also as computations of P.

2.4 The Specification Language: Temporal Logic

As the specification language for both property and action styles of specification, we take *temporal logic*.

We assume an underlying first-order assertion language $\mathcal{L}$ over interpreted symbols for expressing functions and relations over some concrete domains, such as the integers, arrays, and lists of integers. Easy reference to the location of control is provided by the predicate at_ℓ_i. We also use the expression $at\_\ell_{i,j}$ as an abbreviation for the disjunction $at\_\ell_i \vee at\_\ell_j$.

We refer to a formula in the assertion language $\mathcal{L}$ as a *state formula*, or simply as an *assertion*.

A *temporal formula* is constructed out of state formulas by applying the boolean operators $\neg$ and $\vee$ (the other boolean operators can be defined from these), quantifiers $\forall$ and $\exists$, and the following temporal operators:

Future Operators			Past Operators		
$\bigcirc p$	—	Next p	$\ominus p$	—	Previously p
$p\,\mathcal{U}\,q$	—	p Until q	$p\,\mathcal{S}\,q$	—	p Since q
x^+	—	Next value of x			

The next value operator $(\cdot)^+$ is similar to $\bigcirc$ in that it refer to the next state. However, it is applied to system variables and x^+ yields a value in the domain of x. A *local formula* is a formula $\varphi(V, V^+)$ whose only temporal operator is the next-value operator.

A formula that contains no future operators is called a *past* formula. A formula that contains no past operators is called a *future* formula. Note that a state formula is both a past and a future formula.

Semantics

Temporal formulas are interpreted over a *model*, which is an infinite sequence of states σ: $s_0, s_1, \ldots$. Note that all computations are models. Variables of the vocabulary $\mathcal{V}$ are partitioned into flexible and rigid variables. *Flexible variables* may assume different values in different states, while *rigid variables* must assume the same value in all states of the model. System variables are always flexible.

For s_j, a state in a model σ, and a state formula q, we write

$s_j \vDash q$,

to denote that q yields T when evaluated over s_j.

Given a model σ, we present an inductive definition for the notion of a temporal formula p holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \vDash p$.

- For a state formula p,
  $(\sigma, j) \vDash p \iff s_j \vDash p$
  That is, we evaluate p locally, using the interpretation provided by s_j.
- For a local formula $p(V, V^+)$,
  $(\sigma, j) \vDash p \iff \langle s_j, s_{j+1}\rangle \vDash p$, where the evaluation of $p(V, V^+)$ uses $s_j[x]$ for the value of $x \in V$, and uses $s_{j+1}[x]$ for the value of x^+.
- $(\sigma, j) \vDash \neg p \iff (\sigma, j) \nvDash p$
- $(\sigma, j) \vDash p \vee q \iff (\sigma, j) \vDash p$ or $(\sigma, j) \vDash q$
- $(\sigma, j) \vDash \bigcirc p \iff (\sigma, j+1) \vDash p$
- $(\sigma, j) \vDash p\,\mathcal{U}\,q \iff$ for some $k \geq j, (\sigma, k) \vDash q$,
  and $(\sigma, i) \vDash p$ for every i such that $j \leq i < k$
- $(\sigma, j) \vDash \ominus p \iff j > 0$ and $(\sigma, j-1) \vDash p$
- $(\sigma, j) \vDash p\,\mathcal{S}\,q \iff$ for some $k \leq j, (\sigma, k) \vDash q$,
  and $(\sigma, i) \vDash p$ for every i such that $j \geq i > k$

Let σ: $s_0, s_1, \ldots$, and σ': $s'_0, s'_1, \ldots$ be two models, and u a (flexible or rigid) variable. We say that σ' is a *u-variant* of σ if for each $j \geq 0$, s'_j differs from s_j by at most the interpretation given to u.
For the quantifiers,

- $(\sigma, j) \models \exists u \colon p \iff (\sigma', j) \models p$ for some σ', a u-variant of σ,
- $(\sigma, j) \models \forall u \colon p \iff (\sigma', j) \models p$ for every σ', a u-variant of σ.

The other boolean connectives, such as $\wedge$ (conjunction), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence), can be defined in terms of $\neg$ and $\vee$.

Additional temporal operators can be defined as follows:

$\Diamond p$	$= \mathrm{T}\,\mathcal{U}\,p$	— Eventually p
$\Box p$	$= \neg \Diamond \neg p$	— Henceforth p
$p\,\mathcal{W}\,q$	$= \Box p \vee (p\,\mathcal{U}\,q)$	— p Waiting-for q (Unless, Weak Until)
$\widetilde{\circleddash} p$	$= \neg \circleddash \neg p$	— Weak Previously p
$\diamondminus p$	$= \mathrm{T}\,\mathcal{S}\,p$	— Sometimes in the past p
$\boxminus p$	$= \neg \diamondminus \neg p$	— Always in the past p
$p\,\mathcal{B}\,q$	$= \boxminus p \vee (p\,\mathcal{S}\,q)$	— p back-to q

Another useful derived operator is the *entailment* operator, defined by:

$$p \Rightarrow q \iff \Box(p \rightarrow q).$$

As example of a local formula, we define for every transition $\tau \in \mathcal{T}$ with transition relation $\rho(V, V')$ the formula

$$\mathit{taken}(\tau): \quad \rho_\tau(V, V^+).$$

Obviously, formula $\mathit{taken}(\tau)$ holds at position j of a model iff state s_{j+1} is a τ-successor of state s_j.

Validity, Congruence, and *P*-Validity

For a state formula p and a state s such that p holds on s, we say that s is a *p-state*. A state formula that holds on all states is called *state-valid*.

For a temporal formula p and a position $j \geq 0$ such that $(\sigma, j) \models p$, we say that j is a *p-position* (in σ).

If $(\sigma, 0) \models p$, we say that p *holds* on σ, and denote it by $\sigma \models p$. A formula p is called *satisfiable* if it holds on some model. A formula is called (*temporally*) *valid* if it holds on all models.

Two formulas p and q are defined to be *equivalent*, denoted $p \sim q$, if the formula $p \leftrightarrow q$ is valid, i.e., $\sigma \models p$ iff $\sigma \models q$, for all models σ.

In the sequel, we adopt the convention by which a formula p that is claimed to be valid is state-valid if p is an assertion, and is temporally valid if p contains at least one temporal operator.

The formulas p and q are defined to be *congruent*, denoted $p \approx q$, if the formula $\Box(p \leftrightarrow q)$ is valid, i.e., $(\sigma, j) \models p$ iff $(\sigma, j) \models q$, for all models σ and all positions $j \geq 0$. If $p \approx q$ then p can be replaced by q in any context, i.e., $\varphi(p) \approx \varphi(q)$ for any formula $\varphi(p)$ containing occurrences of p.

The notion of (temporal) validity requires that the formula holds over *all* models. Given a program P, we can restrict our attention to the set of models which correspond to computations of P, i.e., $\mathcal{C}omp(P)$. This leads to the notion of P-validity, by which a temporal formula p is *P-valid* (valid over program P) if it holds over all the computations of P. Obviously, any formula that is temporally valid is also P-valid for any program P. In a similar way, we obtain the notions of P-satisfiability and P-equivalence.

A state s that appears in some computation of P is called a *P-accessible* state. A state formula is called *P-state valid* if it holds over all P-accessible states. Obviously, any state formula that is state-valid is also P-state valid for any program P.

Again, we adopt the convention by which we may refer to a P-state valid formula simply as P-valid.

2.5 Classification of Properties

A temporal formula φ that is valid over a program P specifies a property of P, i.e., states a condition that is satisfied by all computations of P. The properties expressible by temporal logic can be arranged in a hierarchy that identifies different classes of properties according to the form of formulas expressing them.

Here we will consider only properties falling into the three classes: *safety*, *response*, and *reactivity*.

For past formulas p, q, and r, we define the following classes of canonical formulas.

$\Box\, p$	is a canonical *safety* formula
$p \Rightarrow \Diamond\, r$	is a canonical *response* formula
$p \,\wedge\, \Box\Diamond\, q \;\Rightarrow\; \Diamond\, r$	is a canonical *reactivity* formula.

A formula φ is called a *safety*, *response*, or *reactivity* formula if it is equivalent to a canonical safety, response, or reactivity formula, respectively. In this paper, we restrict our attention to the cases that p and q are state formulas, and r is either a state formula or a local formula. Even though a local formula is not a past formula, the formulas $p \Rightarrow \Diamond\, r$ and $p \wedge \Box\Diamond\, q \Rightarrow \Diamond\, r$ are response and reactivity formulas, respectively, also for the cases that p and q are state formulas and r is a local formula.

3 Proof Rules

In this section, we present several temporal proof rules that can be used to prove that a given temporal formula is valid over a given program. These rules are taken from [MP94] (appear also in [MP91a]), where they are used for proving properties of programs.

Here, we present the rules and extend them to cases in which r is a local formula. All the rules have a temporal conclusion, but most of their premises are first-order implications, possibly referring to primed variables, which are treated as any other variable in a first-order formula. Since the rules are to be applied in the context of a given system (program), the meaning of stating an assertional implication is that this formula is P-state valid, and stating a temporal formula means that the formula is P-valid.

3.1 A Rule for Safety

For the case that p is an assertion (state formula), we refer to the safety formula $\Box\, p$ also as an *invariance formula*. If $\Box\, p$ is valid over program P, we say that p is an invariant of P.

Rule INV, presented in Figure 2 is a complete rule for proving that assertion p is an invariant of program P.

For assertions φ and p,

$$\begin{array}{ll} \text{I1.} & \Theta \;\rightarrow\; \varphi \\ \text{I2.} & \varphi \;\rightarrow\; p \\ \text{I3.} & \rho_\tau \;\wedge\; \varphi \;\;\rightarrow\;\; \varphi' \quad \text{for every } \tau \in \mathcal{T} \\ \hline & \Box\, p \end{array}$$

Fig. 2. Rule INV (invariance).

Premise I1 of rule INV requires that Θ, the initial condition of P, implies the auxiliary assertion φ. Premise I2 of the rule implies that the auxiliary assertion φ implies assertion p, whose invariance we wish to prove. Premise I3 requires that all transitions in P preserve φ. The premise contains φ', the *primed version* of the assertion φ, which is obtained by replacing each variable occurring in φ by its primed version. Since $\rho_\tau(V, V')$ holds whenever V characterizes one state and V' characterizes its τ-successor, the P-state validity of premise I3 implies that every τ-successor of a φ-state is also a φ-state.

Premises I1 and I3 state that φ holds at the initial state of every computation and that it propagates from any state to its τ-successor, for every transition τ of the system. Thus, every state in each computation of P satisfies φ. Due to the implication I2, every such state also satisfies p. It follows that p is an invariant of system P.

Example Let us illustrate the application of rule INV to prove that assertion

$$p: \quad s = 1 \vee s = 2$$

is an invariant of program PET. We take $\varphi = p : s = 1 \vee s = 2$ in the rule.

Premise I1, requires showing

$$\underbrace{\cdots \;\wedge\; s = 1 \;\wedge\; \cdots}_{\Theta} \quad\rightarrow\quad \underbrace{s = 1 \vee s = 2}_{\varphi}$$

This implication is state-valid and, therefore, also P-state valid.

Premise I2 is trivially valid since $\varphi = p$.

Premise I3 has to be checked for the 13 transitions of program PET, including the idling transition τ_I, transitions $\ell_0, \ldots, \ell_5$ of process P_1, and transitions $m_0, \ldots, m_5$ of process P_2.

We find it sufficient to check three classes of transitions.

- The verification condition for transition ℓ_2 is:
$$\underbrace{\cdots \wedge\ s' = 1\ \wedge\ \cdots}_{\rho_{\ell_2}} \wedge \underbrace{\cdots}_{\varphi} \quad\rightarrow\quad \underbrace{s' = 1 \vee \cdots}_{\varphi'}$$
- The verification condition for transition m_2 is:
$$\underbrace{\cdots \wedge\ s' = 2\ \wedge\ \cdots}_{\rho_{m_2}} \wedge \underbrace{\cdots}_{\varphi} \quad\rightarrow\quad \underbrace{\cdots \vee s' = 2}_{\varphi'}$$
- All other transitions $\tau \in \mathcal{T} - \{\ell_2, m_2\}$ preserve the value of s. Therefore, their verification condition is:
$$\underbrace{\cdots \wedge\ s' = s\ \wedge\ \cdots}_{\rho_{\tau}} \wedge \underbrace{s = 1 \vee s = 2}_{\varphi} \quad\rightarrow\quad \underbrace{s' = 1 \vee s' = 2}_{\varphi'}$$

Obviously, all these implications are state-valid.

3.2 A Rule for Response

In Figure 3, we present rule RESP, which is complete for proving response properties of the form
$$p \ \Rightarrow\ \diamondsuit\, r,$$
where p is an assertion and r is either an assertion or a local formula.

The rule refers to a well-founded domain $(\mathcal{A}, \succ)$. Recall that a well-founded domain $(\mathcal{A}, \succ)$ consists of a set $\mathcal{A}$ and a *well-founded* binary relation $\succ$ on $\mathcal{A}$, such that there does not exist an infinitely descending sequence $a_0, a_1, \ldots$ of elements of $\mathcal{A}$, such that
$$a_0 \ \succ\ a_1 \ \succ\ \cdots.$$

Other constructs used by the rule include a sequence of *intermediate assertions*, $\varphi_1, \ldots, \varphi_m$, a corresponding sequences of *helpful transitions* $\tau_1, \ldots, \tau_m \in \mathcal{J} \cup \mathcal{C}$, and a sequence of *ranking functions* $\delta_1, \ldots, \delta_m$: $\mathcal{S} \mapsto \mathcal{A}$ mapping states into the domain $\mathcal{A}$. The ranking functions measure the distance of the current state from the state (or position) satisfying the goal formula r. As the computation progresses towards the goal r-position the value of the ranking δ_i decreases.

A transition τ is considered helpful for a state s if every τ-successor of s has a rank lower than that of s. The rule selects a set of helpful transitions which are either just or compassionate. Assertion φ_i characterizes all the states that can arise in a computation between a p-position and a later r-position, for which transition τ_i is helpful. The corresponding ranking function δ_i defines the rank of all states satisfying φ_i.

Since the rule allows r to be either an assertion or a local formula, it uses a generalized primed version of formula r which is defined as follows:
$$pr(r): \quad \begin{cases} r(V, V') & \text{if } r(V, V^+) \text{ is a local formula referring to some } x^+ \\ r(V') & \text{if } r(V) \text{ is an assertion, not referring to any } x^+ \end{cases}$$

For assertions $p, \varphi_1, \ldots, \varphi_m$, and formula r
transitions $\tau_1, \ldots, \tau_m \in \mathcal{J} \cup \mathcal{C}$,
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_1, \ldots, \delta_m \colon \mathcal{S} \mapsto \mathcal{A}$

$$\left.\begin{array}{ll}
\text{R1.} & p \;\rightarrow\; r \;\vee\; \bigvee\limits_{j=1}^{m} \varphi_j \\[2ex]
\text{R2.} & \rho_\tau \wedge \varphi_i \;\rightarrow\; \left[\begin{array}{l} pr(r) \;\vee\; \bigvee\limits_{j=1}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j) \\ \vee \;\; (\varphi'_i \;\wedge\; \delta_i = \delta'_i) \end{array}\right] \\
 & \hfill \text{for every } \tau \in \mathcal{T} \\[2ex]
\text{R3.} & \rho_{\tau_i} \wedge \varphi_i \;\rightarrow\; pr(r) \;\vee\; \bigvee\limits_{j=1}^{m} (\varphi'_j \;\wedge\; \delta_i \succ \delta'_j) \\[2ex]
\text{R4.} & \text{If } \tau_i \text{ is compassionate, then} \\
 & \qquad \text{CR4.} \quad \varphi_i \;\Rightarrow\; \Diamond\big(\neg\varphi_i \;\vee\; En(\tau_i)\big) \\
 & \quad \text{Otherwise} \\
 & \qquad \text{JR4.} \quad \varphi_i \;\rightarrow\; En(\tau_i)
\end{array}\right\} \text{ for } i = 1, \ldots, m$$

$$p \;\Rightarrow\; \Diamond\, r$$

Fig. 3. Rule RESP (response).

Premise R1 requires that every p-position either already satisfies r, or satisfies one of the intermediate assertions $\varphi_1, \ldots, \varphi_m$.

Premise R2 requires that, taking any transition from a φ_i-position k, always leads to a successor position $k' = k + 1$, such that

- either r is an assertion which holds at k', or
- $r(V, V^+)$ is a local formula which holds at position k, or
- some φ_j, $j = 1, \ldots, m$, holds at k' with a rank δ'_j lower than δ_i at k, or
- φ_i holds at k' with a rank δ'_i equal to the rank δ_i at k.

The main implication of premise R2 is that if the situation has not improved in any noticeable way in going from k to k', i.e., we have not reached r and the new rank still equals the old rank, at least we have not lost the identity of the helpful transitions and the transition that was helpful in k is also helpful in k'.

Premise R3 requires that transition τ_i, which is helpful for φ_i, always leads from a φ_i-position k to a next position which either satisfies r, or satisfies some φ_j and has a rank lower than that of k, i.e., $\delta_i \succ \delta'_j$.

Premise R4 connects assertion φ_i with enableness of transition τ_i which is the helpful transition for φ_i-states. It distinguishes between the cases that τ_i is compassionate or only just. For the case that τ_i is only just, subpremise JR4 requires that τ_i be enabled on any φ_i-state. For compassionate transitions, subpremise CR4 does not require immediate enableness but is ready to be satisfied by eventual enableness. It requires that every φ_i-state is followed by a position that either enables τ_i or achieves the goal r.

Note that if τ_i is compassionate, we may still attempt to prove for it the simpler (and stronger) subpremise JR4 : $\varphi_i \rightarrow En(\tau_i)$, because this implication obviously implies CR4 : $\varphi_i \Rightarrow \diamondsuit(\neg\varphi_i \vee En(\tau_i))$.

All first-order premises and subpremises of rule RESP are implications required to be P-state valid. This means that if we have previously shown some assertions $\chi_1, \ldots, \chi_k$ to be invariants of the program, i.e., to hold on every P-accessible state, then, we may add them on the left-hand side of any of the implications. For example, if assertions $\chi_1, \ldots, \chi_k$ have been proven to be invariants of P using rule INV, it is sufficient to prove

$$\left(\bigwedge_{i=1}^{k} \chi_i\right) \wedge p \quad \rightarrow \quad r \vee \bigvee_{j=1}^{m} \varphi_j$$

in order to establish premise R1.

Example Let us illustrate the application of rule RESP to prove the accessibility property for program PET, which can be stated by the response formula

$$\psi_1 : \quad at\_\ell_2 \quad \Rightarrow \quad \diamondsuit\, at\_\ell_4.$$

This formula states that every position at which process P_1 is at location ℓ_2, implying it is interested in entering the critical section, is followed by a position at which P_1 is at the critical section (location ℓ_4).

We prove this property by rule RESP with $p : at\_\ell_2$ and $r : at\_\ell_4$. For the well-founded domain, we take $(\mathcal{A}, \succ) = (\mathbb{N}, >)$. That is, the domain is the set of natural numbers and the well-founded ordering is the greater-than relation $>$. For the other constructs we take the following:

i	φ_i	τ_i	δ_i
6	at_ℓ_2	ℓ_2	5
5	$at\_\ell_3 \wedge at\_m_3 \wedge y_1 \wedge y_2 \wedge s = 1$	m_3	4
4	$at\_\ell_3 \wedge at\_m_4 \wedge y_1 \wedge y_2 \wedge s = 1$	m_4	3
3	$at\_\ell_3 \wedge at\_m_5 \wedge y_1 \wedge y_2 \wedge s = 1$	m_5	2
2	$at\_\ell_3 \wedge at\_m_{0..2} \wedge y_1 \wedge \neg y_2 \wedge s = 1$	ℓ_3	1
1	$at\_\ell_3 \wedge at\_m_3 \wedge y_1 \wedge y_2 \wedge s = 2$	ℓ_3	1

The expression $at\_m_{0..2}$ is an abbreviation for the disjunction $at\_m_0 \vee at\_m_1 \vee at\_m_2$.

Precisely the same set of constructs can be used to establish the response property

$$at\_\ell_2 \quad \Rightarrow \quad \diamondsuit(\pi_1 = 3 \wedge \pi_1^+ = 4).$$

For example, premise R3 for assertion φ_1 and helpful transition $\tau_1 = \ell_3$ assumes the form

$$\underbrace{\pi_1 = 3 \;\wedge\; \pi_1' = 4 \;\wedge\; \cdots}_{\rho_{\ell_3}} \wedge \underbrace{\cdots}_{\varphi_1} \quad\rightarrow\quad \underbrace{\pi_1 = 3 \;\wedge\; \pi_1' = 4}_{pr(\pi_1=3 \,\wedge\, \pi_1^+=4)}$$

3.3 A Rule for Reactivity

In Figure 4, we present rule REACT, which is complete for proving reactivity properties of the form

$$p \;\wedge\; \Box\Diamond q \;\Rightarrow\; \Diamond r,$$

where p and q are assertions and r is either an assertion or a local formula.

For assertions p, q, $\varphi_1, \ldots, \varphi_m$, and formula r
extended transitions $\tau_1, \ldots, \tau_m \in \mathcal{J} \cup \mathcal{C} \cup \{\tau_q\}$
a well-founded domain $(\mathcal{A}, \succ)$, and
ranking functions $\delta_1, \ldots, \delta_m\colon\ \rho \mapsto \mathcal{A}$

C1. $p \;\rightarrow\; r \vee \bigvee_{j=1}^{m} \varphi_j$

$$\left.\begin{array}{l}
\text{C2.}\quad \rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \left[\begin{array}{l} pr(r) \;\vee\; \bigvee_{j=1}^{m} (\varphi_j' \;\wedge\; \delta_i \succ \delta_j') \\ \vee\; (\varphi_i' \;\wedge\; \delta_i = \delta_i') \end{array}\right] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{for every } \tau \in \mathcal{T} \\
\text{C3.}\quad \rho_{\tau_i} \;\wedge\; \varphi_i \;\rightarrow\; pr(r) \;\vee\; \bigvee_{j=1}^{m} (\varphi_j' \;\wedge\; \delta_i \succ \delta_j') \\
\text{C4.}\quad \text{If } \tau_i \in \mathcal{C} \\
\qquad \text{CC4.}\quad \varphi_i \;\wedge\; \Box\Diamond q \;\Rightarrow\; \Diamond\big(\neg\varphi_i \;\vee\; En(\tau_i)\big) \\
\quad\ \text{If } \tau_i \in \mathcal{J} - \mathcal{C} \\
\qquad \text{JC4.}\quad \varphi_i \;\rightarrow\; En(\tau_i)
\end{array}\right\} \text{for } i = 1, \ldots, m$$

$$p \;\wedge\; \Box\Diamond q \;\Rightarrow\; \Diamond r$$

Fig. 4. Rule REACT (reactivity).

Rule REACT generalizes rule RESP in introducing the notion of a *virtual transition* associated with the assertion q. We refer to this virtual transition as τ_q, and its transition relation is given by

$$\rho_{\tau_q} = \rho_q : \quad q' \;\wedge\; \bigvee_{\tau \in \mathcal{T}} \rho_\tau$$

Thus, a state s' is a τ_q-successor of the state s if s' satisfies q and is a τ-successor of s, for some transition $\tau \in \mathcal{T}$. The rule refers to *extended transitions* which are either a regular transition $\tau \in \mathcal{T}$ or the virtual transition τ_q.

Note that premise C4 contains no requirement for enabling of τ_q. This is because the conclusion is meaningful only for computations in which q is known to occur infinitely many times, which is analogous to the assurance that the extended transition τ_q is taken infinitely many times.

Example Let us illustrate the application of rule REACT to prove a generalized accessibility property for program PET, which can be stated by the reactivity formula

$$\psi_2 : \quad \Box \Diamond \neg at\_\ell_1 \quad \Rightarrow \quad \Diamond \, at\_\ell_4 .$$

The previously proven accessibility property $\psi_1 : at\_\ell_2 \Rightarrow \Diamond \, at\_\ell_4$ ensured that every at_ℓ_2-state is followed by a critical state. The reactivity property ψ_2 states that, no matter where we start, if it is known that process P_1 is infinitely often outside the noncritical section, then it eventually enters the critical section.

Property ψ_2 is proven by rule REACT with p : T, $q : \neg at\_\ell_1$, and $r : at\_\ell_4$. For the well-founded domain, we take $(\mathcal{A}, \succ) = (\mathbb{N}, >)$. For the other constructs we take the following:

i	φ_i	τ_i	δ_i
9	at_ℓ_5	ℓ_5	8
8	at_ℓ_0	ℓ_0	7
7	at_ℓ_1	$\tau_{\neg at\_\ell_1}$	6
6	at_ℓ_2	ℓ_2	5
5	$at\_\ell_3 \wedge at\_m_3 \wedge y_1 \wedge y_2 \wedge s = 1$	m_3	4
4	$at\_\ell_3 \wedge at\_m_4 \wedge y_1 \wedge y_2 \wedge s = 1$	m_4	3
3	$at\_\ell_3 \wedge at\_m_5 \wedge y_1 \wedge y_2 \wedge s = 1$	m_5	2
2	$at\_\ell_3 \wedge at\_m_{0..2} \wedge y_1 \wedge \neg y_2 \wedge s = 1$	ℓ_3	1
1	$at\_\ell_3 \wedge at\_m_3 \wedge y_1 \wedge y_2 \wedge s = 2$	ℓ_3	1

Let us consider, for example, premise C3 for $i = 6$. This premise is given by the obviously state-valid implication.

$$\underbrace{\neg at'\_\ell_1 \wedge (\cdots \vee \overbrace{at\_\ell_1 \wedge at'\_\ell_2 \wedge \cdots}^{\rho_{\ell_1}} \vee \cdots)}_{\tau_{\neg at\_\ell_1}} \wedge \underbrace{at\_\ell_1}_{\varphi_6} \quad \rightarrow \quad \underbrace{at'\_\ell_2}_{\varphi_5'}$$

The P-state validity of this implication follows from the fact that $at\_\ell_1 \wedge at'\_\ell_2$ is the only disjunct of $\bigvee\limits_{\tau \in \mathcal{T}} \rho_\tau$ which is consistent with $\varphi_6 : at\_\ell_1$.

4 Simulation between Systems

Having shown how temporal logic can be used to specify and verify properties of programs, which are compatible with the property style of specification, we proceed to develop methods which are compatible with the action style of specification.

4.1 The Temporal Semantics of Systems

Our approach to action style specification is based on the observation that, for every fair transition system S, there exists a temporal formula $comp_S$, called the *temporal semantics* of system S, which specifies the set of all computations of S [MP83]. There are several (equivalent) ways to present this formula. The presentation of the temporal semantics formula in [MP91b] uses the past formula $last\text{-}taken(\tau) : \rho_\tau(V^-, V)$. In our presentation here, we prefer to use instead the local formula $taken(\tau) : \rho_\tau(V, V^+)$.

Let system S be given by the components $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}\rangle$. We recall and introduce some formulas that express different properties of computations of S.

- The assertion

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

  states that transition τ with the transition relation $\rho_\tau(V, V')$ is enabled.
- The local formula

$$taken(\tau) : \rho_\tau(V, V^+)$$

  holds at position j in a model if s_{j+1} is a τ-successor of s_j.
- The formula

$$just(\tau) : \Diamond \Box En(\tau) \;\rightarrow\; \Box \Diamond taken(\tau)$$

  states that, if τ is enabled at all but finitely many positions, then τ is taken at infinitely many positions. Consequently, a model that satisfies $just(\tau)$ is just with respect to τ.
- The formula

$$compassionate(\tau) : \Box \Diamond En(\tau) \rightarrow \Box \Diamond taken(\tau)$$

  states that, if τ is enabled at infinitely many positions, then τ is taken at infinitely many positions. Consequently, a model satisfying $compassionate(\tau)$ is compassionate with respect to τ.

The temporal semantics formula of system S is given by

$$comp_S : \quad \Theta \;\wedge\; \Box \bigvee_{\tau\in\mathcal{T}} taken(\tau) \;\wedge\; \bigwedge_{\tau\in\mathcal{J}} just(\tau) \;\wedge\; \bigwedge_{\tau\in\mathcal{C}} compassionate(\tau).$$

Let us consider each clause appearing in the formula $comp_S$:

- The conjunct Θ ensures that the initial state satisfies the initial condition Θ.
- $\Box \bigvee_{\tau\in\mathcal{T}} taken(\tau)$ ensures that, for every $i = 0, 1, \ldots,$ state s_{i+1} is a τ-successor of state s_i, for some transition $\tau \in \mathcal{T}$.
- $\bigwedge_{\tau\in\mathcal{J}} just(\tau)$ states that all justice requirements are satisfied.

- $\bigwedge_{\tau \in \mathcal{C}} \mathit{compassionate}(\tau)$ states that all compassion requirements are satisfied.

The four clauses correspond to and ensure the four requirements a model has to satisfy in order to be a computation of S.

The fact that the formula comp_S precisely characterizes the computations of S is stated by the following claim.

Claim 1 *A model σ satisfies comp_S iff σ is a computation of S.*

4.2 Simulation between Systems

The simplest case of refinement between systems is that of simulation. Our interpretation of this term is that a system $S^{\mathcal{C}}$ *simulates* system $S^{\mathcal{A}}$, denoted

$$S^{\mathcal{C}} \preccurlyeq S^{\mathcal{A}}$$

if every computation of $S^{\mathcal{C}}$ is a computation of $S^{\mathcal{A}}$. Thus, simulation is inclusion between the set of computations of $S^{\mathcal{C}}$ and that of $S^{\mathcal{A}}$. Typically, $S^{\mathcal{A}}$ represents a more *abstract* version of the system, which can be viewed as an (action) specification for $S^{\mathcal{C}}$, while $S^{\mathcal{C}}$ represents a more *concrete* version of the system, which can be viewed as an implementation of $S^{\mathcal{C}}$.

This interpretation is compatible with the observations that $S^{\mathcal{C}}$ eliminates some of the nondeterminism inherent in $S^{\mathcal{A}}$ by eliminating some of the computations that $S^{\mathcal{A}}$ allows, and $S^{\mathcal{C}}$ may contain additional variables that can be used to implement higher level operations of $S^{\mathcal{A}}$.

We illustrate these two observations on the programs $P_1^{\mathcal{A}}$ and $P_1^{\mathcal{C}}$, presented in Figure 5.

$$\begin{array}{ll}
\textbf{out} & y : \textbf{integer where } y = 0 \\
\textbf{local} & x : \textbf{integer where } x = 0
\end{array}$$

$$\left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l} \ell_1\text{: } x := y + 1 \\ \ell_2\text{: } y := x + 1 \end{array}\right]
\end{array}\right]$$

– Concrete Version $P_1^{\mathcal{C}}$ –

$$\textbf{out} \;\; y : \textbf{integer where } y = 0$$

$$\left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l} \ell_1\text{: } \textbf{skip} \\ \ell_2\text{: } \left[\begin{array}{l} \ell_2^a\text{: } y := y + 2 \\ \textbf{or} \\ \ell_2^b\text{: } \textbf{skip} \end{array}\right] \end{array}\right]
\end{array}\right]$$

– Abstract Version $P_1^{\mathcal{A}}$ –

Fig. 5. Two versions of program P_1.

Program $P_1^{\mathcal{A}}$ repeatedly increments variable y by 2. It also has a nondeterministic choice at location ℓ_2, where it may choose to increment y in the current round of the loop or skip. This program uses the operation of adding 2 to a variable. Program $P_1^{\mathcal{C}}$, the concrete version of this program, uses only the simpler operation of adding 1 to a variable. In order to increment y by 2, it uses an auxiliary variable x, which receives the value of $y + 1$ in statement ℓ_1, and is used in statement ℓ_2 to compute the value of $x+1 = y+2$. Unlike the abstract version, $P_1^{\mathcal{C}}$ does not have a nondeterministic choice at location ℓ_2.

We claim that program $P_1^{\mathcal{C}}$ simulates program $P_1^{\mathcal{A}}$. That is, every computation of $P_1^{\mathcal{C}}$ is also a computation of $P_1^{\mathcal{A}}$. If we examine, for example, the computations of $P_1^{\mathcal{C}}$ which do not take the idling transition τ_I, we find that there exists only one such computation. It has the following form, where we list for each state the interpretation it assigns to the system variables π, x, and y:

$$\begin{aligned}\sigma_1 : &\langle \pi:0,\, x:0,\, y:0\rangle,\ \langle \pi:1,\, x:0,\, y:0\rangle,\ \langle \pi:2,\, x:1,\, y:0\rangle\\ &\langle \pi:0,\, x:1,\, y:2\rangle,\ \langle \pi:1,\, x:1,\, y:2\rangle,\ \langle \pi:2,\, x:3,\, y:2\rangle\\ &\langle \pi:0,\, x:3,\, y:4\rangle,\ \cdots\end{aligned}$$

It is not difficult to see that σ_1 is also a computation of $P_1^{\mathcal{A}}$ even though it assigns values to variable x which is not a system variable of $P_1^{\mathcal{A}}$. This because, independently of the considered program, every state interprets all variables in the vocabulary $\mathcal{V}$ and not only the system variables.

On the other hand, consider the following prefix of a model:

$$\begin{aligned}\sigma_2 : &\langle \pi:0,\, x:0,\, y:0\rangle,\ \langle \pi:1,\, x:1,\, y:0\rangle,\ \langle \pi:2,\, x:2,\, y:0\rangle,\\ &\langle \pi:0,\, x:3,\, y:0\rangle\end{aligned}$$

Regardless of the arbitrary values assigned to variable x, this prefix can be extended to a computation of program $P_1^{\mathcal{A}}$. This is because the interpretation of x, which is not a system variable of $P_1^{\mathcal{A}}$, is irrelevant to the requirements a computation of $P_1^{\mathcal{A}}$ should satisfy. In comparison, prefix σ_2 cannot be extended to a computation of $P_1^{\mathcal{C}}$ for two reasons. The interpretation given to x, which is a system variable of $P_1^{\mathcal{C}}$, is incompatible with the transitions of $P_1^{\mathcal{C}}$. Also, $P_1^{\mathcal{C}}$ cannot move from ℓ_2 to ℓ_0 without increasing y by 2, which is a step that is included in the prefix σ_2.

Under the assumption that $P_1^{\mathcal{C}}$ simulates $P_1^{\mathcal{A}}$, which we formally verify in the next subsection, the example of prefix σ_2 shows that the set of computations of $P_1^{\mathcal{C}}$ is strictly included in the set of computations of $P_1^{\mathcal{A}}$.

4.3 Verifying Simulation

Having defined and illustrated the relation of simulation between systems, we proceed to propose a formal method for its verification. First, we identify two necessary and sufficient conditions for simulation.

Claim 2 *For systems $S^{\mathcal{A}}$ and $S^{\mathcal{C}}$,*

$$S^{\mathcal{C}} \preccurlyeq S^{\mathcal{A}} \qquad \textit{iff} \qquad comp_{S^{\mathcal{C}}} \rightarrow comp_{S^{\mathcal{A}}} \qquad \textit{iff} \qquad S^{\mathcal{C}} \models comp_{S^{\mathcal{A}}}$$

Claim 2 is a direct consequence of claim 1, stating that the temporal semantics formula of a system precisely characterizes the computations of a system, and the definitions of validity of a temporal formula over a program and implication between two temporal formulas. All three concepts imply inclusion between the set of models associated with the formal object (system or formula) on the left and the set of models associated with the formal object on the right.

In Figure 6, we present rule SIM, which can be used to verify that system $S^{\mathcal{C}}$ simulates system $S^{\mathcal{A}}$. The rule assumes that the two systems are given by their

S1. $\Theta^{\mathcal{C}} \quad \rightarrow \quad \Theta^{\mathcal{A}}$	
S2. $S^{\mathcal{C}} \models \rho_{\tau^{\mathcal{C}}} \quad \rightarrow \quad \bigvee\limits_{\tau^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}} \rho_{\tau^{\mathcal{A}}}$	for every $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$
S3. $S^{\mathcal{C}} \models En(\tau^{\mathcal{A}}) \quad \Rightarrow \quad \diamondsuit(\neg En(\tau^{\mathcal{A}}) \ \vee \ taken(\tau^{\mathcal{A}}))$	for every $\tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}}$
S4. $S^{\mathcal{C}} \models \Box \diamondsuit En(\tau^{\mathcal{A}}) \quad \Rightarrow \quad \diamondsuit taken(\tau^{\mathcal{A}})$	for every $\tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}}$
$S^{\mathcal{C}} \preccurlyeq S^{\mathcal{A}}$	

Fig. 6. Rule SIM (simulation between systems).

five components, i.e., $S^{\mathcal{A}} : \langle V^{\mathcal{A}}, \Theta^{\mathcal{A}}, \mathcal{T}^{\mathcal{A}}, \mathcal{J}^{\mathcal{A}}, \mathcal{C}^{\mathcal{A}} \rangle$ and $S^{\mathcal{C}} : \langle V^{\mathcal{C}}, \Theta^{\mathcal{C}}, \mathcal{T}^{\mathcal{C}}, \mathcal{J}^{\mathcal{C}}, \mathcal{C}^{\mathcal{C}} \rangle$ and that $V^{\mathcal{A}} \subseteq V^{\mathcal{C}}$.

Premise S1 requires that the initial condition of $S^{\mathcal{C}}$ implies the initial condition of $S^{\mathcal{A}}$. Premise S2 ensures that every state change that can be caused by some concrete transition $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$ can also be explained by some abstract transition. These two premises guarantee that any model that satisfies the requirements of initiality and consecution for the concrete system $S^{\mathcal{C}}$ also satisfies these requirements for the abstract system $S^{\mathcal{A}}$.

Premises S3 and S4 take care of the two fairness requirements of justice and compassion. Premise S3 ensures that any position in an $S^{\mathcal{C}}$-computation in which the just abstract transition $\tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}}$ is enabled is followed by a position at which $\tau^{\mathcal{A}}$ is either disabled or taken. This guarantees that computations of $S^{\mathcal{C}}$ are just with respect to all just transitions of $S^{\mathcal{A}}$. In a similar way, premise S4 ensures that, if a compassionate abstract transition $\tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}}$ is enabled at infinitely many positions of some $S^{\mathcal{C}}$ computation, then any position is followed by a position at which $\tau^{\mathcal{A}}$ is taken. This guarantees that computations of $S^{\mathcal{C}}$ are compassionate with respect to all compassionate transitions of $S^{\mathcal{A}}$.

Consequently, every computation of $S^{\mathcal{C}}$ is also a computation of $S^{\mathcal{A}}$.

Premise S1 is an assertional formula and is required to be state valid. Premise S2 is an assertional implication which is required to be P-state valid. This means that it can be established by proving the state validity of an implication

$$\varphi \ \wedge \ \rho_{\tau^{\mathcal{C}}} \quad \rightarrow \quad \bigvee_{\tau^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}} \rho_{\tau^{\mathcal{A}}},$$

where φ is an invariant of system $S^{\mathcal{C}}$, that can be proven using rule INV.

Premise S3 is a response formula, which is required to be valid over system $S^{\mathcal{C}}$. It can be established using rule RESP. In a similar way, premise S4 is a reactivity formula, whose validity over system $S^{\mathcal{C}}$ can be proven by rule REACT.

Sufficient Conditions Leading to a Simpler rule

From the preceding discussion, one may get the impression that every proof of simulation must involve proofs of several invariants, a proof of one response property for each just abstract transition, and a proof of one reactivity property for each compassionate abstract transition. This is not always the case.

In some cases, we can identify a mapping $\kappa : \mathcal{T}^{\mathcal{C}} \mapsto \mathcal{T}^{\mathcal{A}}$, which maps each concrete transition to a corresponding abstract transition. The intended meaning of the mapping is that $\tau^{\mathcal{C}}$ is always explained in $S^{\mathcal{C}}$ as activation of the corresponding abstract transition $\kappa(\tau^{\mathcal{C}})$.

When there exists a transition mapping κ, we can simplify rule SIM into rule S-SIM, presented in Figure 7.

For a transition mapping $\kappa : \mathcal{T}^{\mathcal{C}} \mapsto \mathcal{T}^{\mathcal{A}}$,				
SS1.	$\Theta^{\mathcal{C}}$	$\rightarrow$	$\Theta^{\mathcal{A}}$	
SS2.	$\rho_{\tau^{\mathcal{C}}}$	$\rightarrow$	$\rho_{\kappa(\tau^{\mathcal{C}})}$	for every $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$
SS3.	$En(\tau^{\mathcal{A}})$	$\Rightarrow$	$\diamondsuit\, \neg En(\tau^{\mathcal{A}})$	for every $\tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}}$
SS4.	$En(\tau^{\mathcal{A}})$	$\Rightarrow$	$\diamondsuit\, taken(\tau^{\mathcal{A}})$	for every $\tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}}$
	$S^{\mathcal{C}} \preccurlyeq S^{\mathcal{A}}$			

Fig. 7. Rule S-SIM (simpler simulation rule).

All premises of the rule are interpreted as $S^{\mathcal{C}}$-state valid (SS1 and SS2) or $S^{\mathcal{C}}$-valid (SS3 and SS4). Premise SS1 requires, as before, that the initial condition of $S^{\mathcal{C}}$ implies the initial condition of $S^{\mathcal{A}}$. Premise SS2 specializes the corresponding premise in rule SIM. It requires that the state change caused by $\tau^{\mathcal{C}}$ can be explained by abstract transition $\kappa(\tau^{\mathcal{C}})$. Premise SS3 simplifies premise S3 of rule SIM, by making a stronger requirement. It requires that every state on which $\tau^{\mathcal{A}}$ is enabled be followed by a state on which $\tau^{\mathcal{A}}$ is disabled. In a similar way, premise SS4 imposes a stronger requirement than premise S4. It requires that every position at which $\tau^{\mathcal{A}}$ is enabled be followed by a position at which $\tau^{\mathcal{A}}$ is taken. In the weaker version of S4, activation of $\tau^{\mathcal{A}}$ was guaranteed only when $\tau^{\mathcal{A}}$ is enabled infinitely many times.

Transition $\tau^{\mathcal{A}}$ is said to have a *unique inverse* under κ if there exists precisely one concrete transition $\tau^{\mathcal{C}}$, such that $\kappa(\tau^{\mathcal{C}}) = \tau^{\mathcal{A}}$. We write $\tau^{\mathcal{C}} = \kappa^{-1}(\tau^{\mathcal{A}})$.

A further simplification is still possible. For fair transitions τ_1 and τ_2 (possibly belonging to different systems), we say that transition τ_2 is *at least as fair as* transition τ_1 if either τ_2 is compassionate or τ_1 is not compassionate. This is consistent with the view that compassion is a stronger type of fairness than simply justice.

A transition mapping κ is said to be *fairness reducing* if $\tau^{\mathcal{C}}$ is at least as fair as $\kappa(\tau^{\mathcal{C}})$ for every fair concrete transition $\tau^{\mathcal{C}}$. For the case of a fairness reducing transition mapping and a fair abstract transition that has a unique inverse, the following simpler premise is a sufficient condition for premises S3 or S4 of rule SIM.

$$ES3. \quad En(\tau^{\mathcal{A}}) \quad \rightarrow \quad En(\kappa^{-1}(\tau^{\mathcal{A}}))$$

Premise ES3 discharges the fairness obligation associated with $\tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}} \cup \mathcal{C}^{\mathcal{C}}$ by relying on the fairness obligation associated with $\kappa^{-1}(\tau^{\mathcal{A}})$.

Consider first the case that $\tau^{\mathcal{A}}$ is just. By fairness reduction, it follows that $\tau^{\mathcal{C}} = \kappa^{-1}(\tau^{\mathcal{A}})$ is at least just (i.e., it is just or compassionate). If $\tau^{\mathcal{A}}$ is continually enabled beyond some position $j \geq 0$ then, by ES3, so is $\tau^{\mathcal{C}}$. Since we are dealing with a computation of $S^{\mathcal{C}}$ which is fair with respect to $\tau^{\mathcal{C}}$, $\tau^{\mathcal{C}}$ must be taken at some later position $k \geq j$. By SS2, when $\tau^{\mathcal{C}}$ is taken, then so is $\tau^{\mathcal{A}}$. It follows that every computation of $S^{\mathcal{C}}$ is just with respect to $\tau^{\mathcal{A}}$.

If $\tau^{\mathcal{A}}$ is compassionate then, by fairness reduction, $\tau^{\mathcal{C}} = \kappa^{-1}(\tau^{\mathcal{A}})$ is compassionate. If $\tau^{\mathcal{A}}$ is enabled infinitely many times, then so is $\tau^{\mathcal{C}}$. By compassion with respect to $\tau^{\mathcal{C}}$, this transition must be taken infinitely many times, and so must be $\tau^{\mathcal{A}}$.

Example Let us illustrate the application of rule S-SIM to prove that program $P_1^{\mathcal{C}}$ simulates program $P_1^{\mathcal{A}}$.

We use a transition mapping κ defined as:

$$\kappa(\tau_I^{\mathcal{C}}) = \tau_I^{\mathcal{A}} \quad , \quad \kappa(\ell_0^{\mathcal{C}}) = \ell_0^{\mathcal{A}} \quad , \quad \kappa(\ell_1^{\mathcal{C}}) = \ell_1^{\mathcal{A}} \quad , \quad \kappa(\ell_2^{\mathcal{C}}) = \ell_2^{a}$$

Note that ℓ_2^b is not the image of any concrete transition under κ. The mapping κ is fairness reducing, since all just concrete transitions are mapped into just abstract transitions.

Premise SS1 assumes the form:

$$\underbrace{\pi = 0 \ \wedge \ x = y = 0}_{\Theta^{\mathcal{C}}} \quad \rightarrow \quad \underbrace{\pi = 0 \ \wedge \ y = 0}_{\Theta^{\mathcal{A}}}$$

which is obviously state-valid.

Before checking premise SS2, we establish an invariant for program $P_1^{\mathcal{C}}$.

$$\varphi : \quad at\_\ell_2 \rightarrow x = y + 1$$

This invariant states that $x = y+1$ whenever control is at location ℓ_1 of program $P_1^{\mathcal{C}}$. It can be proven by taking $p = \varphi$ in rule INV.

Premise SS2 gives rise to the following set of verification conditions.

$$\begin{array}{l}
\rho_{\tau_I^{\mathcal{C}}} \rightarrow \rho_{\tau_I^{\mathcal{A}}} : \ pres(\pi, x, y) \ \rightarrow \ pres(\pi, y) \\
\rho_{\ell_0^{\mathcal{C}}} \rightarrow \rho_{\ell_0^{\mathcal{A}}} : \ move(\ell_0, \ell_1) \ \wedge \ pres(x, y) \ \rightarrow \ move(\ell_0, \ell_1) \ \wedge \ pres(y) \\
\rho_{\ell_1^{\mathcal{C}}} \rightarrow \rho_{\ell_1^{\mathcal{A}}} : \ move(\ell_1, \ell_2) \ \wedge \ x' = y + 1 \ \wedge \ pres(y) \ \rightarrow \ move(\ell_1, \ell_2) \ \wedge \ pres(y) \\
\varphi \wedge \rho_{\ell_2^{\mathcal{C}}} \ \rightarrow \ \rho_{\ell_2^{a}} : \ (at_\ell_2 \rightarrow x = y + 1) \ \wedge \ move(\ell_2, \ell_0) \ \wedge \ y' = x + 1 \ \wedge \ pres(x) \\
\qquad\qquad\qquad\qquad \rightarrow \ move(\ell_2, \ell_0) \ \wedge \ y' = y + 2
\end{array}$$

The condition for transition $\ell_2^{\mathcal{C}}$ uses the invariant $\varphi : at\_\ell_2 \rightarrow x = y + 1$ to obtain a state-valid implication.

Clearly, all these implications are state-valid.

Next, we check premise ES3 for all fair abstract transitions that have a κ-inverse. For these transitions, we have the following set of verification conditions:

$$\begin{array}{lccc} En(\ell_0^{\mathcal{A}}) \rightarrow En(\ell_0^{\mathcal{C}}): & at\_\ell_0 & \rightarrow & at\_\ell_0 \\ En(\ell_1^{\mathcal{A}}) \rightarrow En(\ell_1^{\mathcal{C}}): & at\_\ell_1 & \rightarrow & at\_\ell_1 \\ En(\ell_2^{a}) \rightarrow En(\ell_2^{\mathcal{C}}): & at\_\ell_2 & \rightarrow & at\_\ell_2 \end{array}$$

These conditions are all state-valid.
The just abstract transition ℓ_2^b does not have a κ-inverse. Consequently, we prove for it premise SS3 of rule S-SIM.

$$\underbrace{at\_\ell_2}_{En(\ell_2^b)} \quad \Rightarrow \quad \Diamond(\underbrace{\neg at\_\ell_2}_{\neg En(\ell_2^b)} \wedge \cdots)$$

This response property can be proven by rule RESP, taking $p : at\_\ell_2$, $r : \neg at\_\ell_2$, and a single intermediate assertion $\varphi : at\_\ell_2$, with the helpful transition ℓ_2 and the ranking function $\delta : 1$.

This concludes the proof that program $P_1^{\mathcal{C}}$ simulates program $P_1^{\mathcal{A}}$.

5 Refinement between Systems

Consider program P_2 and P_3 presented in Figure 8.

out y : **integer where** $y = 0$	**out** y : **integer where** $y = 0$ **local** x : **integer where** $x = 0$
$\left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } \textbf{skip} \\ \ell_2\text{: } y := y + 2 \end{array}\right] \end{array}\right]$	$\left[\begin{array}{l} \ell_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_1\text{: } x := y + 1 \\ \ell_2\text{: } y := x + 1 \end{array}\right] \end{array}\right]$
– Program P_2 –	– Program P_3 –

Fig. 8. Programs P_2 and P_3.

Program P_3 is identical to our old acquaintance program $P_1^{\mathcal{C}}$ of Figure 5, while program P_2 is a simplified version of program $P_1^{\mathcal{A}}$ of the same figure, from which we have eliminated the nondeterministic choice at location ℓ_2.

In the previous section, we showed that program P_3 simulates program P_2. What about the other direction? It is obvious that P_2 cannot simulate P_3 because P_3 contains the system variable x which is not a system variable of P_2, and one of the conditions for simulation is that the system variables of the abstract system be a subset of the system variables of the concrete system.

However, the fact that x is defined to be local in P_3 may be interpreted to mean that x is not expected to be an observable variable and is used only as an auxiliary variable to define the behavior observed in the *out* variable y. The whole idea of action style specification depends strongly on the ability to abstract away such auxiliary variables and relieve the implementation from the need to mimic those variables too. This is the significant element in making action specifications free of implementation bias.

Consider, for example, a specification of a buffer system with observable operations of enqueuing and dequeuing data elements in and out of the buffer. A very well known action specification of such a system is a system B which uses an auxiliary variable b which is a queue (list of elements), where each enqueuing operation adds an element to the end of b and each dequeuing operation removes an element from the head of b. We certainly do not wish to force all implementations of this specifications to use precisely the data structure b. Acceptable implementation may elect to implement the buffer by an array, or a set of several buffers, or in many other ways. The only thing required from any implementation of the buffer specification is that every sequence of observable actions, i.e, sequence of $enqueue(d)$ and $dequeue(d)$, generated by the implementation can also be generated by the system B.

To ensure that implementations are not required to also emulate variable b of the specification B, we identify b as local, or nonobservable. This leads to the notion of refinement which we interpret to mean simulation while ignoring unobservable (local) variables.

5.1 Fair Transition Modules

When we identify a subset of the system variables of the system as being local (unobservable), we obtain the notion of a transition module.

A *fair transition module* consists of the components $M : \langle V, L, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}\rangle$, where V, Θ, $\mathcal{T}$, $\mathcal{J}$, and $\mathcal{C}$ have the same role as in fair transition systems, and $L \subseteq V$ is the set of *local variables*, i.e., system variables which are unobservable.

Every program P is associated with a fair transition module. This module can be obtained by constructing first the fair transition system for P and identifying the local variables as consisting of the control variables $\pi_1, \ldots, \pi_m$, and all data variables that are declared under the mode *local*. Thus, the fair transition module associated with program P_3 has π and x as its local variables.

Computations of a transition module M are defined precisely as they are defined for transition systems. However, for transition modules, we also have the more abstract notion of observations.

An *observation* of a fair transition module M is a model which is an L-variant of a computation of M. That is, any model that differs from a computation of M by at most the interpretation of the variables in L.

For example, the model

$$\begin{aligned}\sigma_3 : \; & \langle \pi : 0\,,\, x : 0\,,\, y : 0\rangle,\ \langle \pi : 1\,,\, x : 1\,,\, y : 0\rangle,\ \langle \pi : 2\,,\, x : 2\,,\, y : 0\rangle \\ & \langle \pi : 0\,,\, x : 3\,,\, y : 2\rangle,\ \langle \pi : 1\,,\, x : 4\,,\, y : 2\rangle,\ \langle \pi : 2\,,\, x : 5\,,\, y : 2\rangle \\ & \langle \pi : 0\,,\, x : 6\,,\, y : 4\rangle,\ \cdots\end{aligned}$$

is an observation of the transition module for P_3. This is because σ_3 is an x-

variant of the following computation of P_3.

$$\begin{array}{l}\sigma_4 : \langle \pi : 0\,,\, x : 0\,,\, y : 0\rangle,\ \langle \pi : 1\,,\, x : 0\,,\, y : 0\rangle,\ \langle \pi : 2\,,\, x : 1\,,\, y : 0\rangle \\ \quad\ \ \langle \pi : 0\,,\, x : 1\,,\, y : 2\rangle,\ \langle \pi : 1\,,\, x : 1\,,\, y : 2\rangle,\ \langle \pi : 2\,,\, x : 3\,,\, y : 2\rangle \\ \quad\ \ \langle \pi : 0\,,\, x : 3\,,\, y : 4\rangle,\ \cdots\end{array}$$

We say that a module $M^{\mathcal{C}}$ *refines* module $M^{\mathcal{A}}$, denoted

$$M^{\mathcal{C}} \sqsubseteq M^{\mathcal{A}},$$

if every observation of $M^{\mathcal{C}}$ is an observation of $M^{\mathcal{A}}$. Two modules M_1 and M_2 are said to be *equivalent* if each of them refines the other.

It is not difficult to see that program P_2 refines program P_3. Since simulation implies refinement, we also have that P_3 refines P_2. It follows that programs P_2 and P_3 are equivalent.

5.2 The Observational Semantics of Modules

Similar to the representation of computations of a system S by the formula $comp_S$, we introduce for a module M the formula obs_M, called the *observational semantics* formula of M, which specifies the set of all observations of M. The formula obs_M is given by

$$obs_M : \quad \exists L : comp_M.$$

Thus, a model σ satisfies the formula obs_M if there exists an L-variant of σ which is a computation of M.

The following claim connects the notion of refinement between modules to their observational semantics formula.

Claim 3 *For modules $M^{\mathcal{A}}$ and $M^{\mathcal{C}}$,*

$$M^{\mathcal{C}} \sqsubseteq M^{\mathcal{A}} \quad \textit{iff} \quad obs_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}} \quad \textit{iff} \quad comp_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}} \quad \textit{iff} \quad M^{\mathcal{C}} \models obs_{M^{\mathcal{A}}}$$

Let us justify the passage from $obs_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}}$ to $comp_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}}$ in the claim. We observe the following chain of equivalences:

$$\begin{array}{lcl} obs_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}} & \sim & (\exists L^{\mathcal{C}} : comp_{M^{\mathcal{C}}}) \rightarrow obs_{M^{\mathcal{A}}} \quad \sim \\ & & \qquad\qquad \forall L^{\mathcal{C}} : (comp_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}}) \end{array}$$

Taking the quantification over $L^{\mathcal{C}}$ out of the parenthesis is allowed since (we may assume that) these variables do not occur in $M^{\mathcal{A}}$. It is obvious that the implication $comp_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}}$ is valid iff its universally quantified version $\forall L^{\mathcal{C}}$: $(comp_{M^{\mathcal{C}}} \rightarrow obs_{M^{\mathcal{A}}})$ is.

5.3 Verifying Refinement

The proof obligations for proving refinements, as they appear in claim 3, differ from the proof obligations required for simulation only in the presence of the existential quantification over the local abstract variables L. Consequently, the additional effort required for verifying refinement is focused on methods to handle the existential quantification.

The main rule that can be used to eliminate existential quantification is rule $\exists$-INTR, presented in Figure 9.

$$\frac{p(\overline{x}) \rightarrow q(\overline{x}, \overline{f}(\overline{x}))}{p(\overline{x}) \rightarrow \exists \overline{y} : q(\overline{x}, \overline{y})}$$

Fig. 9. Rule $\exists$-INTR (existential quantification introduction).

The rule considers two lists of variables $\overline{x} : x_1, \ldots, x_a$ and $\overline{y} : y_1 \ldots, y_b$, and two temporal formulas $p(\overline{x})$ and $q(\overline{x}, \overline{y})$. The premise requires finding a vector valued function $\overline{y} = \overline{f}(\overline{x})$ such that the implication $p(\overline{x}) \rightarrow q(\overline{x}, \overline{f}(\overline{x}))$ is valid. The function $\overline{f}(\overline{x})$ is sometimes referred to as a *Skolem* function since it provides a witness to the fact that, for every valuation of $\overline{x}$ satisfying $p(\overline{x})$, there exists a valuation of $\overline{y}$ satisfying $q(\overline{x}, \overline{y})$.

5.4 Refinement Mapping

Consider two modules, $M^{\mathcal{C}}$ and $M^{\mathcal{A}}$, to which we refer as the concrete and abstract modules, wishing to prove that $M^{\mathcal{C}}$ refines $M^{\mathcal{A}}$. We assume that the observable system variables of $M^{\mathcal{A}}$ are also observable system variables of $M^{\mathcal{C}}$.

Let

$$\alpha\colon\ L^{\mathcal{A}} \leftarrow \mathcal{E}(V^{\mathcal{C}})$$

be a *substitution* that replaces each local abstract variable $X \in L^{\mathcal{A}}$ by an expression over the concrete variables. We denote the expression assigned to X by $X_\alpha(V^{\mathcal{C}})$.

For example, the substitution

$$\alpha_{23}: \quad x \ \leftarrow\ \textbf{if}\ at\_\ell_2\ \textbf{then}\ y+1\ \textbf{else}\ y \dot{-} 1$$

expresses the value of the variable x, which is local to program P_3, in terms of the system variables π and y of program P_2. The operation $\dot{-}$ denotes natural number subtraction, such that $y \dot{-} 1$ equals $y - 1$ if y is positive and equals 0 otherwise.

Such substitution α induces a mapping on states as follows:

Let $s^{\mathcal{C}}$ be a state which appears in a computation of $M^{\mathcal{C}}$. We refer to $s^{\mathcal{C}}$ as a *concrete state*. The abstract state $s^{\mathcal{A}} = m_\alpha(s^{\mathcal{C}})$ corresponding to the concrete state $s^{\mathcal{C}}$ is such that the value of each local abstract variable $X \in L^{\mathcal{A}}$ in $s^{\mathcal{A}}$ is the value of the expression $X_\alpha(V^{\mathcal{C}})$ when evaluated in $s^{\mathcal{C}}$. We say that m_α is a *module mapping* from $M^{\mathcal{C}}$ to $M^{\mathcal{A}}$.

For example, state $\langle \pi : 1\,,\, x : 4\,,\, y : 2\rangle$ that may appear in a computation of program P_2, is mapped by $m_{\alpha_{23}}$ to the state $\langle \pi : 1\,,\, x : 1\,,\, y : 2\rangle$ which appears in a computation of P_3. The difference between the two states is in the interpretation of variable x which has been changed to $x_{\alpha_{23}}(\pi : 1, y : 2) = 2 \dot{-} 1 = 1$.

For an expression or state formula Φ over $V^{\mathcal{A}}$, we denote by $\Phi[\alpha]$ the expression (state formula) over $V^{\mathcal{C}}$, obtained by replacing each occurrence of $X \in L^{\mathcal{A}}$ by $X_\alpha(V^{\mathcal{C}})$. In the case that Φ contains a primed version X', it is replaced by $X_\alpha\big((V^{\mathcal{C}})'\big)$, priming every occurrence of $v \in V^{\mathcal{C}}$ in X_α.

Given a concrete state $s^{\mathcal{C}}$, it is obvious that the value of Φ evaluated over $m_\alpha(s^{\mathcal{C}})$ is the same as the value of $\Phi[\alpha]$ evaluated over $s^{\mathcal{C}}$. This is because, in both cases, we evaluate Φ using, for each $X \in L^{\mathcal{A}}$, its value in $m_\alpha(s^{\mathcal{C}})$ which is the same as the value of X_α in $s^{\mathcal{C}}$. In particular, for a state formula Φ

$$m_\alpha(s^{\mathcal{C}}) \models \Phi \qquad \text{iff} \qquad s^{\mathcal{C}} \models \Phi[\alpha].$$

The mapping m_α can be extended to a mapping between models. Thus, if σ: $s_0, s_1, \ldots$ is a model, then its image under m_α is $m_\alpha(\sigma)$: $m_\alpha(s_0), m_\alpha(s_1), \ldots$.

Consider, for example, model σ_3 which is a computation of program P_2. Its mapping under $m_{\alpha_{23}}$ yields model σ_4, which is a computation of program P_3. This is not a coincidence. The mapping $m_{\alpha_{23}}$ always maps computations of P_2 to computations of P_3.

A system mapping m_α from $M^{\mathcal{C}}$ to $M^{\mathcal{A}}$ is called a *refinement mapping* if it maps computations of $M^{\mathcal{C}}$ to computations of $M^{\mathcal{A}}$. For example, the mapping α_{23} is a refinement mapping from program P_2 to program P_3.

Obviously, the existence of a refinement mapping from $M^{\mathcal{C}}$ to $M^{\mathcal{A}}$ is a sufficient condition for $M^{\mathcal{C}}$ to be a refinement of $M^{\mathcal{A}}$. To see this, consider $\sigma^{\mathcal{C}}$ a computation of $M^{\mathcal{C}}$. By the definition of a refinement mapping, $\sigma^{\mathcal{A}} = m_\alpha(\sigma^{\mathcal{C}})$ is a computation of $M^{\mathcal{A}}$. Since m_α only modifies variables that are local for $M^{\mathcal{A}}$, $\sigma^{\mathcal{C}}$ is an $L^{\mathcal{A}}$-variant of a computation of $M^{\mathcal{A}}$, namely $\sigma^{\mathcal{A}}$. It follows that $\sigma^{\mathcal{C}}$ is an observation of $M^{\mathcal{A}}$. This shows that every computation of $M^{\mathcal{C}}$ is an observation of $M^{\mathcal{A}}$ and, therefore, $M^{\mathcal{C}}$ refines $M^{\mathcal{A}}$.

Our approach to verify that module $M^{\mathcal{C}}$ refines module $M^{\mathcal{A}}$ is to find a module mapping m_α induced by the substitution α, and prove that it is a refinement mapping.

The remaining question is how to prove that a proposed mapping m_α from $M^{\mathcal{C}}$ to $M^{\mathcal{A}}$ is a refinement mapping. The following claim answers this question.

Claim 4 *The substitution* α: $L^{\mathcal{A}} \leftarrow \mathcal{E}(V^{\mathcal{C}})$ *induces a refinement mapping* m_α *from* $M^{\mathcal{C}}$ *to* $M^{\mathcal{A}}$ *iff*

$$comp_{M^{\mathcal{C}}} \rightarrow comp_{M^{\mathcal{A}}}[\alpha] \qquad \textit{iff} \qquad M^{\mathcal{C}} \models comp_{M^{\mathcal{A}}}[\alpha]$$

To justify the claim, we observe that $M^{\mathcal{C}} \models comp_{M^{\mathcal{A}}}[\alpha]$ if and only if every σ, a computation of $M^{\mathcal{C}}$, satisfies $comp_{M^{\mathcal{A}}}[\alpha]$, if and only if $\widetilde{\sigma} = m_\alpha$ satisfies $comp_{M^{\mathcal{A}}}$, if and only if α is a refinement mapping from $M^{\mathcal{C}}$ to $M^{\mathcal{A}}$.

The purpose of the refinement mapping is to represent the values of the local abstract variables in terms of system variables of the concrete module, under the assumption that these variables are not recognized as system variables by the concrete module. The observable (nonlocal) abstract variables are required to be recognized as system variables by the concrete module. In some cases, not all local abstract variables are foreign to the concrete module. For example, the local variables of program P_3 are π and x. Of these, x is not recognized by program P_2. However, the control variable π is recognized by P_2 and is also a local variable there.

In the cases that some local abstract variables coincide with local concrete variables, there is no need to map them by the substitution α. We therefore allow module mapping which map only part of the local abstract variables, provided the remaining local abstract variables are also local concrete variables. Formally, this is equivalent to having a mapping which maps all local abstract variables, but where the common local variables are mapped to themselves by an identity substitution.

5.5 A Proof Rule for Refinement

The necessary and sufficient conditions listed in claim 3 generalize the similar conditions listed in claim 2 for establishing simulation. The extension is the refinement substitution α applied to $comp_{M^{\mathcal{A}}}$. Based on this observation, it is straightforward to extend in a compatible way rule SIM and its simplified version S-SIM to establish refinement between modules.

The adapted version of rule SIM is rule REF, presented in Figure 10.

$$
\begin{array}{|llcll|}
\hline
\multicolumn{5}{|c|}{\text{For modules } M^{\mathcal{C}} \text{ and } M^{\mathcal{A}}, \text{ and a substitution } \alpha\colon L^{\mathcal{A}} \leftarrow \mathcal{E}(V^{\mathcal{C}}),} \\
\text{F1.} & \Theta^{\mathcal{C}} & \rightarrow & \Theta^{\mathcal{A}}[\alpha] & \\
\text{F2.} & M^{\mathcal{C}} \models \rho_{\tau^{\mathcal{C}}} & \rightarrow & \bigvee_{\tau^{\mathcal{A}} \in \mathcal{T}^{\mathcal{A}}} \rho_{\tau^{\mathcal{A}}}[\alpha] & \text{for every } \tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}} \\
\text{F3.} & M^{\mathcal{C}} \models En(\tau^{\mathcal{A}})[\alpha] & \Rightarrow & \diamondsuit(\neg En(\tau^{\mathcal{A}})[\alpha] \vee taken(\tau^{\mathcal{A}})[\alpha]) & \text{for every } \tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}} \\
\text{F4.} & M^{\mathcal{C}} \models \square\diamondsuit\, En(\tau^{\mathcal{A}})[\alpha] & \Rightarrow & \diamondsuit\, taken(\tau^{\mathcal{A}})[\alpha] & \text{for every } \tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}} \\
\hline
 & M^{\mathcal{C}} \sqsubseteq M^{\mathcal{A}} & & & \\
\hline
\end{array}
$$

Fig. 10. Rule REF (refinement between modules).

The rule consistently replaces every possible occurrence of a local abstract variable $X \in L^{\mathcal{A}}$ by its expression $X[\alpha]$ in terms of the system variables of module $M^{\mathcal{C}}$. For the nonlocal (observable) system variables of $M^{\mathcal{A}}$, the assumption is that they coincide with some system variables of $M^{\mathcal{C}}$.

As in the case of simulation, the situation becomes much simpler if we can find a transition mapping $\kappa : \mathcal{T}^{\mathcal{C}} \mapsto \mathcal{T}^{\mathcal{A}}$ between the transitions of $M^{\mathcal{C}}$ and those of $M^{\mathcal{A}}$. Rule S-REF of Figure 11 presents the simpler premises that have to be proven for this case.

For a surjective mapping mapping $\kappa : \mathcal{T}^{\mathcal{C}} \mapsto \mathcal{T}^{\mathcal{A}}$,				
SF1.	$\Theta^{\mathcal{C}}$	$\rightarrow$	$\Theta^{\mathcal{A}}[\alpha]$	
SF2.	$\rho_{\tau^{\mathcal{C}}}$	$\rightarrow$	$\rho_{\kappa(\tau^{\mathcal{C}})}[\alpha]$	for every $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$
SF3.	$En(\tau^{\mathcal{A}})[\alpha]$	$\Rightarrow$	$\diamondsuit \neg En(\tau^{\mathcal{A}})[\alpha]$	for every $\tau^{\mathcal{A}} \in \mathcal{J}^{\mathcal{A}}$
SF4.	$En(\tau^{\mathcal{A}})[\alpha]$	$\Rightarrow$	$\diamondsuit\, taken(\tau^{\mathcal{A}})[\alpha]$	for every $\tau^{\mathcal{A}} \in \mathcal{C}^{\mathcal{A}}$
	$M^{\mathcal{C}} \sqsubseteq M^{\mathcal{A}}$			

Fig. 11. Rule S-REF (simpler refinement rule).

All premises of the rule are interpreted as $S^{\mathcal{C}}$-state valid (SF1 and SF2) or $S^{\mathcal{C}}$-valid (SF3 and SF4).

If κ is fairness reducing and fair transition $\tau^{\mathcal{A}}$ has a unique κ-inverse, then we may establish the following simplified premise, instead of SF3 or SF4:

$$\text{EF3.}\quad En(\tau^{\mathcal{A}})[\alpha] \quad\rightarrow\quad En(\kappa^{-1}(\tau^{\mathcal{A}}))$$

Example Let us illustrate the application of rule S-REF to prove that program P_2 refines program P_3.

Of the two variables local to P_3, variable π is common to P_2 and need not be mapped. Variable x is unknown to P_2, and we use the substitution

$$\alpha_{23}: \quad x \;\leftarrow\; \textbf{if } at\_\ell_2 \textbf{ then } y+1 \textbf{ else } y \dot{-} 1,$$

expressing x in terms of the concrete variables π (through the control predicate at_ℓ_2) and y.

We use a fairness reducing transition mapping κ defined as:

$$\kappa(\tau_I^{\mathcal{C}}) = \tau_I^{\mathcal{A}} \quad,\quad \kappa(\ell_0^{\mathcal{C}}) = \ell_0^{\mathcal{A}} \quad,\quad \kappa(\ell_1^{\mathcal{C}}) = \ell_1^{\mathcal{A}} \quad,\quad \kappa(\ell_2^{\mathcal{C}}) = \ell_2$$

Premise SF1 of the rule assumes the form:

$$\underbrace{\pi = 0 \;\wedge\; y = 0}_{\Theta^{\mathcal{C}}} \quad\rightarrow\quad \underbrace{\pi = 0 \;\wedge\; \overbrace{\textbf{if } at\_\ell_2 \textbf{ then } y+1 \textbf{ else } y \dot{-} 1}^{x[\alpha]} = 0 \;\wedge\; y = 0}_{\Theta^{\mathcal{A}}[\alpha]}$$

The antecedent conjunct $\pi = 0$ implies $\neg at\_\ell_2$. Consequently, $x[\alpha]$ reduces to $y \dot{-} 1 = 0 \dot{-} 1 = 0$, establishing the right-hand side of the implication.

In proving premise SF2, we use the following invariant

$$\varphi : y \geq 0,$$

which can be easily established for program P_2.

Premise SF2 gives rise to the following set of verification conditions, where we list on the left the concrete transition to which the condition corresponds.

$$
\begin{array}{ll}
\rho_{\tau_I}: & pres(\pi, y) \;\rightarrow\; pres(\pi, \textbf{if } at\_\ell_2 \textbf{ then } y+1 \textbf{ else } y \dot{-} 1, y) \\
\ell_0: & move(\ell_0,\ell_1) \;\wedge\; pres(y) \;\rightarrow \\
 & \qquad\qquad move(\ell_0,\ell_1) \;\wedge\; pres(\textbf{if } at\_\ell_2 \textbf{ then } y+1 \textbf{ else } y \dot{-} 1, y) \\
\ell_1: & move(\ell_1,\ell_2) \;\wedge\; pres(y) \;\rightarrow\; move(\ell_1,\ell_2) \;\wedge\; pres(y) \;\wedge \\
 & \qquad\qquad \underbrace{\textbf{if } at'\_\ell_2 \textbf{ then } y'+1 \textbf{ else } y' \dot{-} 1}_{x'[\alpha]} = y'+1 \\
\ell_2: & \underbrace{y \geq 0}_{\varphi} \;\wedge\; \underbrace{move(\ell_2,\ell_0) \;\wedge\; y' = y+2}_{\rho^C_{\ell_2}} \;\rightarrow\; move(\ell_2,\ell_0) \;\wedge \\
 & \quad \underbrace{\textbf{if } at'\_\ell_2 \textbf{ then } y'+1 \textbf{ else } y' \dot{-} 1}_{x'[\alpha]} = \underbrace{\textbf{if } at\_\ell_2 \textbf{ then } y+1 \textbf{ else } y \dot{-} 1}_{x[\alpha]} \;\wedge \\
 & \qquad\qquad y' = \underbrace{\textbf{if } at\_\ell_2 \textbf{ then } y+1 \textbf{ else } y \dot{-} 1}_{x[\alpha]} + 1
\end{array}
$$

Note that the condition for ℓ_2 uses the invariant φ as part of its antecedent. All these implications are state valid. Consider, for example, the implication for ℓ_1. The antecedent implies $\pi = 1$, $\pi' = 2$, and $y' = y$. From this, we can infer $x'[\alpha] = y + 1 = y' + 1$.

In a similar way, the antecedent of the implication for ℓ_2 implies $y \geq 0$, $\pi = 2$, $\pi' = 0$, and $y' = y + 2$. From this we can infer $x[\alpha] = y + 1$, $x'[\alpha] = y' \dot{-} 1 = y + 1 = x[\alpha]$, and $y' = y + 2 = x[\alpha] + 1$.

Finally, premise EF3 holds trivially for all fair abstract transitions, since the enabling conditions of both the concrete and the corresponding abstract transitions are identical and, respectively, given by

$$En(\ell_0) = at\_\ell_0 \qquad En(\ell_1) = at\_\ell_1 \qquad En(\ell_2) = at\_\ell_2.$$

Refining the Control Variable

In the preceding example, we refined the data variable x, representing it in terms of the other variables. However, the other local variable, the control variable π remained the same. We illustrate now a case where it is necessary to refine the control variable.

Consider program P_4 presented in Figure 12.

$$
\textbf{out}\;\; y : \textbf{integer where } y = 0
$$
$$
\left[\begin{array}{l} L_0\text{: } \textbf{loop forever do} \\ \quad L_2\text{: } y := y + 2 \end{array}\right]
$$

Fig. 12. Program P_4.

Program P_4 can be obtained from P_2 by omitting the *skip* command at ℓ_1, which does not seem to serve any useful purpose anyway. Let us verify that program P_2 refines program P_4. The only local variable of P_4 is the control variable. However, unlike the previous refinement, we can no longer assume a one-to-one correspondence between the control variable of program P_2 and that of program P_4. To distinguish between the two, we represent the control variable of program P_4 by Π and it ranges over $\{0,2\}$. In accordance with this distinction, we also label the locations in P_4 by L_0 and L_2.

As the refinement mapping, we use the following substitution.

$$\alpha: \quad \Pi \ \leftarrow \ \textbf{if } at\_\ell_1 \textbf{ then } 2 \textbf{ else } \pi$$

The effect of this substitution on control predicates of the form at_L_i can be summarized as follows:

$$at\_L_0 \leftrightarrow at\_\ell_0 \qquad \text{and} \qquad at\_L_2 \leftrightarrow at\_\ell_{1,2}$$

We use a fairness reducing transition mapping κ defined as:

$$\kappa(\tau_I^{\mathcal{C}}) = \tau_I^{\mathcal{A}} \quad , \quad \kappa(\ell_0^{\mathcal{C}}) = L_0^{\mathcal{A}} \quad , \quad \kappa(\ell_1^{\mathcal{C}}) = \tau_I^{\mathcal{A}} \quad , \quad \kappa(\ell_2^{\mathcal{C}}) = L_2$$

Note that the transition corresponding to the skip statement at ℓ_1 is mapped to the abstract idling transition. All fair transitions of P_4 have unique κ-inverses. Therefore, we use rule S-REF.

Premise SF1 of the rule assumes the form:

$$\underbrace{\pi = 0 \ \wedge \ y = 0}_{\Theta^{\mathcal{C}}} \quad \rightarrow \quad \underbrace{\overbrace{\textbf{if } at\_\ell_1 \textbf{ then } 2 \textbf{ else } \pi}^{\Pi[\alpha]} = 0 \ \wedge \ y = 0}_{\Theta^{\mathcal{A}}[\alpha]}$$

The antecedent conjunct $\pi = 0$ implies $\neg at\_\ell_1$. Consequently, $\Pi[\alpha]$ reduces to $\pi = 0$, establishing the right-hand side of the implication.

Premise SF2 gives rise to the following set of verification conditions.

$$\begin{array}{ll}
\rho_{\tau_I}: & pres(\pi, y) \ \rightarrow \ pres(\Pi[\alpha], y) \\
\ell_0: & \pi = 0 \ \wedge \ \pi' = 1 \ \wedge \ pres(y) \ \rightarrow \\
 & (\textbf{if } at_\ell_1 \textbf{ then } 2 \textbf{ else } \pi) = 0 \ \wedge \ (\textbf{if } at'_\ell_1 \textbf{ then } 2 \textbf{ else } \pi') = 2 \ \wedge \ pres(y) \\
\ell_1: & \underbrace{\pi = 1 \ \wedge \ \pi' = 2 \ \wedge \ pres(y)}_{\ell_1} \ \rightarrow \\
 & \qquad \underbrace{(\textbf{if } at'_\ell_1 \textbf{ then } 2 \textbf{ else } \pi') = (\textbf{if } at_\ell_1 \textbf{ then } 2 \textbf{ else } \pi) \ \wedge \ pres(y)}_{\tau_I^{\mathcal{A}}[\alpha]} \\
\ell_2: & \pi = 2 \ \wedge \ \pi' = 0 \ \wedge \ y' = y + 2 \ \rightarrow \ y' = y + 2 \ \wedge \\
 & \qquad (\textbf{if } at_\ell_1 \textbf{ then } 2 \textbf{ else } \pi) = 2 \ \wedge \ (\textbf{if } at'_\ell_1 \textbf{ then } 2 \textbf{ else } \pi') = 0
\end{array}$$

It is not difficult to check that all these implications are state-valid.

Next we should check premises SF3 or EF3, for all fair abstract transitions.

For transition L_0, it is sufficient to check premise EF3. This premise holds since

$$En(L_0)[\alpha] = at\_L_0[\alpha] = at\_\ell_0 = En(\ell_0) = En(\kappa^{-1}(L_0))$$

For transition L_2, whose enabling transition yields $En(L_2)[\alpha] = at\_L_2[\alpha] = at\_\ell_{1,2}$, premise EF3 does not hold, and we have to use premise SF3. Since $\kappa^{-1}(L_2) = \ell_2$, premise SF3 requires proving the following response property over program P_2:

$$at\_\ell_{1,2} \quad \Rightarrow \quad \diamondsuit \neg at\_\ell_{1,2}.$$

This property can be easily proved using rule RESP.

This concludes the proof that program P_2 refines program P_4. This is satisfying, since the addition or deletion of a *skip* statement should not change a program in any noticeable way. However, according to this criterion, we should also be able to prove that program P_4 refines program P_2. This cannot be done in the framework we have so far developed.

Program P_4 does not refine program P_2 since the following computation of program P_4 is not an observation of program P_2.

$$\begin{aligned} \sigma_5 : {} & \langle \Pi : 0\,,\, y : 0\rangle,\ \langle \Pi : 2\,,\, y : 0\rangle,\ \langle \Pi : 0\,,\, y : 2\rangle,\ \langle \Pi : 2\,,\, y : 2\rangle, \\ & \langle \Pi : 0\,,\, y : 4\rangle,\ \cdots \end{aligned}$$

Model σ_5 is not an observation of program P_2 because there is no way that P_2 can obtain a nonzero value for y in the 3rd state. The earliest that P_2 can assign a nonzero value to y is in the 4th state of a computation.

This is where we pay the price for having a logic (and a semantics) which is sensitive to stuttering. In the second half of the paper, we introduce the logic TLR, which solves this problem. As we will see, under the TLR framework, program P_4 also refines program P_2 and the two are equivalent.

In practice, this price is not very high. One possible approach to circumventing the stuttering problem is to introduce a coarser equivalence between programs, to which we may refer as *stuttering equivalence*. A useful program transformation, called *spacing transformation*, replaces a statement S in a program P by the concatenation $[S; \mathbf{skip}]$ or $[\mathbf{skip}; S]$ to obtain a new program $\widetilde{P}$. We can prove that, if program $\widetilde{P}$ is obtained from program P by a sequence of spacing transformations, then P and $\widetilde{P}$ are stuttering equivalent.

In many cases that two programs Q_1 and Q_2 are stuttering equivalent but not equivalent according to our definition, we can derive programs $\widetilde{Q}_1$ and $\widetilde{Q}_2$ that are spaced versions of Q_1 and Q_2, such that $\widetilde{Q}_1$ is equivalent to $\widetilde{Q}_2$ according to the definition given here. Thus, a combination of spacing transformations and the verification techniques given here can prove many cases of stuttering equivalence.

This is trivially illustrated in the case of programs P_2 and P_4 just considered. It is a trivial case, since P_2 can be obtained from P_4 by a spacing transformation, replacing statement $L_1 : y := y + 2$ of P_4 by the concatenation $[\ell_1 : \mathbf{skip};\ \ell_2 : y := y + 2]$. Consequently, this case is completely resolved by the spacing transformation and no additional proving is required.

5.6 Mutual Exclusion for Two Processes

For a more advanced example, we reconsider program PET of Figure 1, and show how it can be refined into a more realistic implementation. For convenience, we replicate the program in Figure 13.

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

$$
P_1 :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_1\text{:} & \textbf{noncritical} \\
\ell_2\text{:} & (y_1, s) := (\mathrm{T}, 1) \\
\ell_3\text{:} & \textbf{await } \neg y_2 \lor s = 2 \\
\ell_4\text{:} & \textbf{critical} \\
\ell_5\text{:} & y_1 := \mathrm{F}
\end{array}\right]
\end{array}\right]
\quad \| \quad
P_2 :: \left[\begin{array}{l}
m_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
m_1\text{:} & \textbf{noncritical} \\
m_2\text{:} & (y_2, s) := (\mathrm{T}, 2) \\
m_3\text{:} & \textbf{await } \neg y_1 \lor s = 1 \\
m_4\text{:} & \textbf{critical} \\
m_5\text{:} & y_2 := \mathrm{F}
\end{array}\right]
\end{array}\right]
$$

Fig. 13. Program PET (Peterson's algorithm for mutual exclusion).

Usually, the correctness of program PET is verified with respect to a property style specification which includes the following properties:

Mutual Exclusion	ψ_0 :	$\square \neg(at\_\ell_4 \land at\_m_4)$
Accessibility for process P_1	ψ_1 :	$at\_\ell_2 \Rightarrow \diamond\, at\_\ell_4$
Accessibility for process P_2	ψ_2 :	$at\_m_2 \Rightarrow \diamond\, at\_m_4$

To establish these properties, we prove first the following auxiliary invariants:

$$
\begin{array}{ll}
\varphi_0\text{:} & s = 1 \lor s = 2 \\
\varphi_1\text{:} & y_1 \leftrightarrow at\_\ell_{3..5} \\
\varphi_2\text{:} & y_2 \leftrightarrow at\_m_{3..5} \\
\varphi_3\text{:} & at\_\ell_3 \land at\_m_{4,5} \rightarrow y_2 \land s = 1. \\
\varphi_4\text{:} & at\_\ell_{4,5} \land at\_m_3 \rightarrow y_1 \land s = 2.
\end{array}
$$

When we have at our disposal also the tool of action specification, it is possible to establish the correctness of program PET by showing that it refines an abstract program that can be viewed as the specification of all mutual exclusion algorithms.

In Figure 14, we present program MUX-PROT, which can be viewed as the "mother of all mutual exclusion algorithms", in the sense that every acceptable solution to the mutual exclusion problem (for two processes) must refine program MUX-PROT.

Program MUX-PROT uses the statement **enter**, which is a schematic statement intended to represent the conditions under which a process may enter its critical section. The transition relations for statements L_2 and M_2 are given by:

$$
\begin{array}{l}
\rho_{L_2} : move(L_2, L_3) \land \neg at\_M_3 \\
\rho_{M_2} : move(M_2, M_3) \land \neg at\_L_3
\end{array}
$$

Thus, process P_1 may enter its critical section only if P_2 is not occupying its critical section at the moment. Both transitions τ_{L_2} and τ_{M_2} are taken to be

$$P_1 :: \left[\begin{array}{l} L_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} L_1\text{: } \textbf{noncritical} \\ L_2\text{: } \textbf{enter} \\ L_3\text{: } \textbf{critical} \end{array}\right] \end{array}\right] \quad \Big\| \quad P_2 :: \left[\begin{array}{l} M_0\text{: } \textbf{loop forever do} \\ \quad \left[\begin{array}{l} M_1\text{: } \textbf{noncritical} \\ M_2\text{: } \textbf{enter} \\ M_3\text{: } \textbf{critical} \end{array}\right] \end{array}\right]$$

Fig. 14. Program MUX-PROT: prototype of mutual exclusion programs.

compassionate. This will prevent computations in which process P_1 is waiting forever at L_2 while process P_2 keeps entering and exiting its critical section.

Program PET Refines Program MUX-PROT

As a first step in the development sequence, we prove that program PET refines the specification program MUX-PROT.

We use a refinement substitution α which expresses Π_1 and Π_2, the local system variable of program MUX-PROT, in terms of π_1 and π_2, the control variable of program PET. Rather than present the expression for $\Pi_1[\alpha]$ and $\Pi_2[\alpha]$, we prefer to display the effect α has on the control predicates at_L and at_M.

$$\begin{array}{rclcrcl} at\_L_0[\alpha] & \leftrightarrow & at\_\ell_0 & \quad & at\_L_1[\alpha] & \leftrightarrow & at\_\ell_1 \\ at\_L_2[\alpha] & \leftrightarrow & at\_\ell_{2,3} & & at\_L_3[\alpha] & \leftrightarrow & at\_\ell_{4,5} \\ at\_M_0[\alpha] & \leftrightarrow & at\_m_0 & & at\_M_1[\alpha] & \leftrightarrow & at\_m_1 \\ at\_M_2[\alpha] & \leftrightarrow & at\_m_{2,3} & & at\_M_3[\alpha] & \leftrightarrow & at\_m_{4,5} \end{array}$$

For the transition mapping between concrete and abstract transitions, we take:

$$\begin{array}{llll} \kappa(\ell_0) = L_0 & \kappa(\ell_1) = L_1 & \kappa(\ell_2) = \tau_I^{\mathcal{A}} & \kappa(\ell_3) = L_2 \\ \kappa(\ell_4) = \tau_I^{\mathcal{A}} & \kappa(\ell_5) = L_3 & & \\ \kappa(m_0) = M_0 & \kappa(m_1) = M_1 & \kappa(m_2) = \tau_I^{\mathcal{A}} & \kappa(m_3) = M_2 \\ \kappa(m_4) = \tau_I^{\mathcal{A}} & \kappa(m_5) = M_3 & & \end{array}$$

This mapping is fairness reducing for all concrete transitions, except for the just transition ℓ_3 (and m_3) which is mapped to the compassionate transition L_2 (M_2, respectively). It remains to check the premises of rule S-REF.

For premise SF1, we have to check the implication

$$at\_\ell_0 \;\wedge\; at\_m_0 \;\wedge\; \cdots \quad \rightarrow \quad at\_L_0[\alpha] \;\wedge\; at\_M_0[\alpha].$$

Since at_ℓ_0 implies $at\_L_0[\alpha]$ and at_m_0 implies $at\_M_0[\alpha]$, the implication is state-valid.

For premise SF2, we have to check that $\rho_{\tau^{\mathcal{C}}}$ implies $\rho_{\kappa(\tau^{\mathcal{C}})}$ for every concrete transition $\tau^{\mathcal{C}} \in \mathcal{T}^{\mathcal{C}}$. Most of the implications are straightforward. The two transitions that require more attention are ℓ_3 and m_3. Consider premise SF2 for $\tau^{\mathcal{C}} = \ell_3$ and $\kappa(\tau^{\mathcal{C}}) = \kappa(\ell_3) = L_2$.

$$\begin{array}{l} at\_\ell_3 \;\wedge\; at'\_\ell_4 \;\wedge\; (\neg y_2 \vee s = 2) \;\wedge\; \pi'_2 = \pi_2 \;\wedge\; \cdots \quad \rightarrow \\ \qquad\qquad at\_L_2[\alpha] \;\wedge\; at'\_L_3[\alpha] \;\wedge\; \neg at\_M_3[\alpha] \;\wedge\; \Pi'_2[\alpha] = \Pi_2[\alpha] \end{array}$$

It follows from the definitions that $\pi'_2 = \pi_2$ implies $\Pi'_2[\alpha] = \Pi_2[\alpha]$, and that $at\_\ell_3 \wedge at'\_\ell_4$ implies $at\_L_2[\alpha] \wedge at'\_L_3[\alpha]$. We must use invariants $\varphi_2 : y_2 \leftrightarrow at\_m_{3..5}$ and φ_3: $at\_\ell_3 \wedge at\_m_{4,5} \;\rightarrow\; y_2 \wedge s = 1$ to infer that $at\_\ell_3 \wedge (\neg y_2 \vee s = 2)$ implies $\neg at\_m_{4,5} = \neg at\_M_3[\alpha]$.

The case of transition m_3 is verified in a similar way.

For premises SF3–SF4 or EF3, we have to check each of the fair abstract transitions. For each $\tau^{\mathcal{A}} \in \{L_0, L_1, M_0, M_1\}$, $En(\tau^{\mathcal{A}})$ is identical to $En(\kappa^{-1}(\tau^{\mathcal{A}}))$ and their mapping is fairness reducing, so premise EF3 holds. It remains to check premises SF3 or SF4 for the transitions $\tau^{\mathcal{A}} \in \{L_2, L_3, M_2, M_3\}$.

For the compassionate transition L_2, premise SF4 assumes the form

$$\underbrace{at\_\ell_{2,3} \;\wedge\; \neg at\_m_{4,5}}_{En(L_2)[\alpha]} \quad \Rightarrow \quad \Diamond\, taken(\ell_3).$$

This response property can be proven by rule RESP. In fact, we proved a property very similar to this in section 3.

For just transition L_3, premise SF3 assumes the form

$$\underbrace{at\_\ell_{4,5}}_{En(L_3)[\alpha]} \quad \Rightarrow \quad \Diamond\, \underbrace{\neg at\_\ell_{4,5}}_{\neg En(L_3)[\alpha]}\,.$$

This response property can also be proven by a simple application of rule RESP.

Premise SF4 for transition M_2 and premise SF3 for transition M_3 can be proven in a completely symmetric way.

This concludes the proof that program PET refines the action specification for mutual exclusion, as represented by program MUX-PROT.

An interesting fact is that we observe here a principle of "conservation of verification effort". There is a surprising similarity between the reasoning we used for verifying that PET refines MUX-PROT and the reasoning we use to prove, independently, that program PET satisfies the three properties $\psi_0 - \psi_2$ that comprise a property specification for mutual exclusion. The two proofs needed precisely the same auxiliary invariants and response proofs.

5.7 Further Refinement of the Mutual Exclusion Program

In Figure 15, we present a version of program PET. This version can be obtained from the program presented in Figure 13 by renaming statement labels and some of the variables.

In our previous study of program PET, we established for it a sequence of invariants culminating in the property of mutual exclusion. Transcribing these invariants to the presentation of the program in Figure 15, we obtain the following list of invariants:

$$\begin{array}{c}
\textbf{local } Y_1,\ Y_2 : \textbf{boolean where } Y_1 = \text{F},\ Y_2 = \text{F} \\
\quad s \qquad : \textbf{integer where } s = 1 \\[1em]
P_1 :: \left[\begin{array}{l} L_0\text{: } \textbf{loop forever do} \\ \left[\begin{array}{l} L_1\text{: } \textbf{noncritical} \\ L_3\text{: } (Y_1,\ s) := (\text{T},\ 1) \\ L_4\text{: } \textbf{await } \neg Y_2 \ \vee\ s \neq 1 \\ L_5\text{: } \textbf{critical} \\ L_6\text{: } Y_1 := \text{F} \end{array}\right] \end{array}\right] \quad \Bigg\| \quad P_2 :: \left[\begin{array}{l} M_0\text{: } \textbf{loop forever do} \\ \left[\begin{array}{l} M_1\text{: } \textbf{noncritical} \\ M_3\text{: } (Y_2,\ s) := (\text{T},\ 2) \\ M_4\text{: } \textbf{await } \neg Y_1 \ \vee\ s \neq 2 \\ M_5\text{: } \textbf{critical} \\ M_6\text{: } Y_2 := \text{F} \end{array}\right] \end{array}\right]
\end{array}$$

Fig. 15. Program PET1 (Peterson's algorithm — abstract version).

$$\begin{array}{ll}
\Phi_0\text{:} & s = 1 \ \vee\ s = 2 \\
\Phi_1\text{:} & Y_1 \ \leftrightarrow\ at_L_{4..6} \\
\Phi_2\text{:} & Y_2 \ \leftrightarrow\ at_M_{4..6} \\
\Phi_3\text{:} & at_L_4 \ \wedge\ at_M_{5,6} \ \rightarrow\ Y_2 \ \wedge\ s = 1 \\
\Phi_4\text{:} & at_L_{5,6} \ \wedge\ at_M_4 \ \rightarrow\ Y_1 \ \wedge\ s = 2 \\
\Phi_5\text{:} & \neg(at_L_5 \ \wedge\ at_M_5).
\end{array}$$

While program PET1 is an acceptable solution to the mutual exclusion problem, it cannot be directly executed on every available computer architecture. This is because statements L_3 (and M_3) require that y_1 and s (y_2 and s) be assigned new values in one atomic step. This is not always possible on available architectures. In the terminology of [MP91b], program PET1 (and PET) does not obey the LCR (*limited critical reference*) restriction. This is sometimes considered as a drawback and limits the applicability of the algorithm.

To rectify the situation, we wish to consider a refinement of program PET1, which breaks the multiple assignments of L_3 and M_3 into separate statements.

Refinement of PET1

The problematic statement that contains multiple assignments is L_3 (and its counterpart M_3), which assigns values to both Y_1 and s. The obvious refinement is to break L_3 into two statements ℓ_2 and ℓ_3, one assigning to y_1 and the other assigning to s. The only question is in what order we should arrange these two assignments. The two possible refinements for

$$L_3\text{: } (Y_1,\ s) := (\text{T},\ 1)$$

are

$$R_{y,s}\text{: } \begin{cases} \ell_2\text{: } y_1 := \text{T} \\ \ell_3\text{: } s := 1 \end{cases} \qquad\qquad R_{s,y}\text{: } \begin{cases} \ell_2\text{: } s := 1 \\ \ell_3\text{: } y_1 := \text{T} \end{cases}$$

with corresponding refinements for M_3.

We show that $R_{s,y}$ is unacceptable. This is because it leads to the following computation which violates mutual exclusion (listing the values of π, y_1, y_2, and s):

$$\big\langle \{\ell_0,\ m_0\},\ \text{F},\ \text{F},\ 1 \big\rangle \ \rightarrow\ \cdots\ \rightarrow\ \big\langle \{\ell_1,\ m_1\},\ \text{F},\ \text{F},\ 1 \big\rangle \ \rightarrow\ \cdots\ \rightarrow$$

$$\langle\{\ell_3,\ m_1\},\ \text{F},\ \text{F},\ 1\rangle \rightarrow \cdots \rightarrow \langle\{\ell_3,\ m_4\},\ \text{F},\ \text{T},\ 2\rangle \rightarrow$$
$$\langle\{\ell_3,\ m_5\},\ \text{F},\ \text{T},\ 2\rangle \rightarrow \langle\{\ell_4,\ m_5\},\ \text{T},\ \text{T},\ 2\rangle \rightarrow$$
$$\langle\{\ell_5,\ m_5\},\ \text{T},\ \text{T},\ 2\rangle.$$

Note that the transition from $\{\ell_3, m_4\}$ to $\{\ell_3, m_5\}$ is possible because $y_1 = \text{F}$. The transition from $\{\ell_4, m_5\}$ to $\{\ell_5, m_5\}$ is possible because $s = 2$.

A possible explanation of this problem is that P_1 established its priority over P_2 by executing ℓ_2 before P_2 executed m_2. However, P_1 failed to declare its active participation in the contest, by staying at ℓ_3 with $y_1 = \text{F}$. Since the tie-breaking mechanism is relevant only when both processes actively contest, P_2 was allowed to enter. Immediately after that, P_1 entered the contest by setting y_1 to T, and, having already acquired the priority, also entered the critical section.

Thus, the only possibility left is $R_{y,s}$, which declares active participation first and only later sets the priority. Let us check whether $R_{y,s}$ leads to a correct program. The refined program according to $R_{y,s}$ is presented in Figure 16.

$$\begin{array}{c}
\textbf{local}\ y_1,\ y_2 : \textbf{boolean where}\ y_1 = \text{F},\ y_2 = \text{F} \\
s \quad : \textbf{integer where}\ s = 1
\end{array}$$

$$P_1 :: \left[\begin{array}{l}\ell_0\colon \textbf{loop forever do} \\ \left[\begin{array}{l}\ell_1\colon \textbf{noncritical} \\ \ell_2\colon y_1 := \text{T} \\ \ell_3\colon s := 1 \\ \ell_4\colon \textbf{await}\ \neg y_2\ \vee\ s \neq 1 \\ \ell_5\colon \textbf{critical} \\ \ell_6\colon y_1 := \text{F}\end{array}\right]\end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l}m_0\colon \textbf{loop forever do} \\ \left[\begin{array}{l}m_1\colon \textbf{noncritical} \\ m_2\colon y_2 := \text{T} \\ m_3\colon s := 2 \\ m_4\colon \textbf{await}\ \neg y_1\ \vee\ s \neq 2 \\ m_5\colon \textbf{critical} \\ m_6\colon y_2 := \text{F}\end{array}\right]\end{array}\right]$$

Fig. 16. Program PET2 (Peterson's algorithm — concrete version).

We first establish some simple auxiliary invariants for program PET2, that will aid us in proving the refinement

$$\varphi_0\colon \quad s = 1\ \vee\ s = 2$$
$$\varphi_1\colon \quad y_1 \leftrightarrow at\_\ell_{3..6}$$
$$\varphi_2\colon \quad y_2 \leftrightarrow at\_m_{3..6}.$$

It is easy to verify that these assertions are invariant.

To prove that program PET2 refines program PET1, we use the substitution α, given by:

$$\begin{aligned}
\Pi_1[\alpha] &= \textbf{if}\ at_\ell_2\ \textbf{then}\ 3\ \textbf{else}\ \pi_1 \\
\Pi_2[\alpha] &= \textbf{if}\ at_m_2\ \textbf{then}\ 3\ \textbf{else}\ \pi_2 \\
Y_1[\alpha] &= \textbf{if}\ at_\ell_3\ \textbf{then}\ \text{F}\ \textbf{else}\ y_1 \\
Y_2[\alpha] &= \textbf{if}\ at_m_3\ \textbf{then}\ \text{F}\ \textbf{else}\ y_2.
\end{aligned}$$

and a transition mapping κ, given by

$$\begin{array}{lll} \kappa(\tau_I) = \tau_I & & \\ \kappa(\ell_2) = \tau_I & \kappa(\ell_i) = L_i & \text{for every } i \in [0..6],\ i \neq 2 \\ \kappa(m_2) = \tau_I & \kappa(m_i) = M_i & \text{for every } i \in [0..6],\ i \neq 2 \end{array}$$

It is straightforward to verify that premise SF1 holds. It is easy to verify premise SF2 for most of the transitions. We will examine in detail transitions $\ell_1 - \ell_4$.

- Premise SF2 for ℓ_1 is given by:

$$\underbrace{at\_\ell_1 \ \wedge\ at'\_\ell_2 \ \wedge\ pres(\pi_2, y_1, y_2, s)}_{\ell_1} \quad \rightarrow$$

$$\underbrace{at\_L_1[\alpha] \ \wedge\ at'\_L_3[\alpha] \ \wedge\ pres(\Pi_2[\alpha], Y_1[\alpha], Y_2[\alpha], s)}_{L_1}.$$

  The conjunction $at\_\ell_1 \wedge at'\_\ell_2$ implies $at\_L_1[\alpha]$, $at'\_L_3[\alpha]$, $Y_1[\alpha] = y_1$, and $Y_1'[\alpha] = y_1'$. Consequently, the implication is state-valid.
- Premise SF2 for ℓ_2 is given by:

$$\underbrace{at\_\ell_2 \ \wedge\ at'\_\ell_3 \ \wedge\ y_1' = \text{T} \ \wedge\ pres(\pi_2, y_2, s)}_{\ell_2} \quad \rightarrow$$

$$\underbrace{pres(\Pi_1[\alpha], \Pi_2[\alpha], Y_1[\alpha], Y_2[\alpha], s)}_{\tau_I}.$$

  The conjunction $at\_\ell_2 \wedge at'\_\ell_3$ implies $at\_L_3[\alpha]$, $at'\_L_3[\alpha]$, $Y_1[\alpha] = y_1$, and $Y_1'[\alpha] = \text{F}$. By invariant φ_1, at_ℓ_2 implies $y_1 = \text{F}$. It follows that $Y_1'[\alpha] = Y_1[\alpha] = \text{F}$. The conjunct $pres(\pi_2, y_2, s)$ implies $pres(\Pi_2[\alpha], Y_2[\alpha], s)$. Consequently, the implication is P-state valid over program PET2.
- Premise SF2 for ℓ_3 is given by:

$$\underbrace{at\_\ell_3 \ \wedge\ at'\_\ell_4 \ \wedge\ s' = 1 \ \wedge\ pres(\pi_2, y_1, y_2)}_{\ell_3} \quad \rightarrow$$

$$\underbrace{at\_L_3[\alpha] \ \wedge\ at'\_L_4[\alpha] \ \wedge\ Y_1'[\alpha] = \text{T} \ \wedge\ s' = 1 \ \wedge\ pres(\Pi_2[\alpha], Y_2[\alpha])}_{L_3}.$$

  The conjunction $at\_\ell_3 \wedge at'\_\ell_4$ implies $at\_L_3[\alpha]$, $at'\_L_4[\alpha]$, and $Y_1'[\alpha] = y_1'$. By invariant φ_1, at_ℓ_3 implies $y_1 = \text{T}$ which, together with $y_1' = y_1$, yields $y_1' = \text{T}$. It follows that $Y_1'[\alpha] = \text{T}$. The conjunct $s' = 1 \wedge pres(\pi_2, y_1, y_2)$ implies $s' = 1 \wedge pres(\Pi_2[\alpha], Y_2[\alpha])$. Consequently, the implication is P-state valid over program PET2.
- Premise SF2 for ℓ_4 is given by:

$$\underbrace{at\_\ell_4 \ \wedge\ at'\_\ell_5 \ \wedge\ (\neg y_2 \vee s \neq 1) \ \wedge\ pres(\pi_2, y_1, y_2, s)}_{\ell_4} \quad \rightarrow$$

$$\underbrace{at\_L_3[\alpha] \ \wedge\ at'\_L_4[\alpha] \ \wedge\ (\neg Y_2[\alpha] \vee s \neq 1) \ \wedge\ pres(\Pi_2[\alpha], Y_1[\alpha], Y_2[\alpha], s)}_{L_4}.$$

The conjunction $at\_\ell_4 \wedge at'\_\ell_5$ implies $at\_L_4[\alpha]$, $at'\_L_5[\alpha]$, $Y_1[\alpha] = y_1$, and $Y_1'[\alpha] = y_1'$. If y_2 is false, then so is $Y_2[\alpha] = \textbf{if } at\_m_3 \textbf{ then } \text{F} \textbf{ else } y_2$. It follows that $\neg y_2$ implies $\neg Y_2[\alpha]$. Consequently, the implication is state-valid.

The case of transitions $m_1 - m_4$ follows in a completely symmetric way.

Premise EF3 is straightforwardly established for all transitions, except for L_3 and M_3. Consider the case of L_3. The enabling condition $En(L_3)[\alpha]$ is $at\_\ell_{2,3}$ which does not imply the enabling condition of the source transition $\kappa^{-1}(L_3) = \ell_3$. Therefore, for transitions L_3 and M_3, we prove premise SF3 which is given by the following two response properties:

$$at\_\ell_{2,3} \;\Rightarrow\; \diamondsuit \neg at\_\ell_{2,3} \qquad \text{and} \qquad at\_m_{2,3} \;\Rightarrow\; \diamondsuit \neg at\_m_3$$

These response properties can be easily established for program PET2.

This concludes the proof that program PET2 refines program PET1.

5.8 Inheritance of Properties

So far, we have emphasized the use of refinement to prove that a proposed concrete program implements a given abstract program, that can be viewed as an action specification. An equally important use of refinement is in simplifying proofs of properties.

Assume that we have to prove that a property, expressed by a temporal formula φ, is valid over program P. In some cases, proving property φ directly over P may be complicated. It may be possible to construct a simpler program $P^{\mathcal{A}}$ that can be viewed as an abstracted version of P, and show that P refines $P^{\mathcal{A}}$ with refinement substitution α. Assume further, that we can abstract φ in a way similar to the abstraction of the program, obtaining a formula Φ such that $\Phi[\alpha]$ is equivalent to φ. Then the following claim states that instead of verifying φ over P, it is sufficient to verify Φ over $P^{\mathcal{A}}$. This may prove to be a considerably simpler task.

Claim 5 *If program $P^{\mathcal{C}}$ refines program $P^{\mathcal{A}}$ with refinement substitution α and formula Φ is valid over program $P^{\mathcal{A}}$, then $\Phi[\alpha]$ is valid over program $P^{\mathcal{C}}$.*

We assume throughout that the substitution α is admissible for Φ, i.e., it never causes capture of a free variable in a substituting term by a quantifier enclosing the substituted variable.

Let us prove the claim. The fact that $P^{\mathcal{C}}$ refines program $P^{\mathcal{A}}$ with refinement substitution α leads to the implication

$$comp_{P^{\mathcal{C}}} \quad \rightarrow \quad comp_{P^{\mathcal{A}}}[\alpha].$$

Formula Φ being valid over program $P^{\mathcal{A}}$ is the same as

$$comp_{P^{\mathcal{A}}} \quad \rightarrow \quad \Phi.$$

Applying substitution α to both sides, we obtain

$$comp_{P^{\mathcal{A}}}[\alpha] \quad \rightarrow \quad \Phi[\alpha].$$

Combining this with the first implication yields

$$comp_{P^{\mathcal{C}}} \quad \rightarrow \quad \Phi[\alpha],$$

which shows that $\Phi[\alpha]$ is valid over program $P^{\mathcal{C}}$.

Example (integer square root)

Let us illustrate the application of this approach.

Consider program SQUARE-ROOT of Figure 17 for computing the integer square root of a nonnegative integer.

$$\begin{array}{l}
\textbf{in} \quad\; x \quad\;\; : \textbf{integer where } x \geq 0 \\
\textbf{local } u, v : \textbf{integer where } u = 1, v = 1 \\
\textbf{out} \quad z \quad\;\; : \textbf{integer where } z = 0 \\
\\
\left[\begin{array}{l}
\ell_0\text{: } \textbf{while } v \leq x \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_1\text{: } \; u := u + 2 \\
\ell_2\text{: } \; v := v + u \\
\ell_3\text{: } \; z := z + 1
\end{array}\right] \\
\ell_4\text{:}
\end{array}\right]
\end{array}$$

Fig. 17. Program SQUARE-ROOT (square root).

We wish to prove for this program the invariance of the assertion

$$\psi\text{:} \quad at\_\ell_4 \;\rightarrow\; z^2 \leq x < (z+1)^2.$$

This property states partial correctness with respect to the postcondition $z^2 \leq x < (z+1)^2$, which identifies z as the maximal integer which is not higher than the square root of x, thus providing the best integer approximation (from below) of the square root of x.

A direct proof of this property for program SQUARE-ROOT involves establishing three auxiliary invariants relating the values of u and v to the value of z. These invariants are further complicated by the fact that each of u, v, and z, is modified by a different statement in the loop.

Instead, we may consider program SQUARE-ROOT$^{\mathcal{A}}$, presented in Figure 18, which can be viewed as an abstraction of program SQUARE-ROOT.

$$\begin{array}{l}
\textbf{in} \quad\; x \quad\;\;\, : \textbf{integer where } x \geq 0 \\
\textbf{local } U, V : \textbf{integer where } U = 1, V = 1 \\
\textbf{out} \quad z \quad\;\;\, : \textbf{integer where } z = 0 \\
\\
\left[\begin{array}{l}
L_0\text{: } \textbf{while } V \leq x \textbf{ do} \\
\quad L_3\text{: } (U, V, z) := (U + 2, V + U + 2, z + 1) \\
L_4\text{:}
\end{array}\right]
\end{array}$$

Fig. 18. Program SQUARE-ROOT$^{\mathcal{A}}$ (abstracted version of square root).

Program $\textsc{square-root}^{\mathcal{A}}$ is simpler to analyze, since all variables are modified by the same statement L_3. Partial correctness for program $\textsc{square-root}^{\mathcal{A}}$is expressed by the invariance of the assertion

$$\Psi: \quad at\_L_4 \;\rightarrow\; z^2 \leq x < (z+1)^2.$$

The proof is based on proving first the three auxiliary invariants

$$\begin{array}{ll} \Phi_1: & U = 2 \cdot z + 1 \\ \Phi_2: & V = (z+1)^2 \\ \Phi_3: & (z + at\_L_3)^2 \leq x \end{array}$$

Assume that Φ_1–Φ_3 and Ψ have all been proven invariant. As the next step, we show that program $\textsc{square-root}$ refines program $\textsc{square-root}^{\mathcal{A}}$.

To do so, we use the refinement substitution α defined by:

$$\begin{array}{ll} \Pi[\alpha]: & \textbf{if } 1 \leq \pi \leq 3 \textbf{ then } 3 \textbf{ else } \pi \\ U[\alpha]: & u - 2 \cdot at\_\ell_{2,3} \\ V[\alpha]: & v - u \cdot at\_\ell_3 \end{array}$$

and the fairness reducing transition mapping

$$\kappa(\ell_0) = L_0, \qquad \kappa(\tau_I) = \kappa(\ell_1) = \kappa(\ell_2) = \tau_I^{\mathcal{A}}, \qquad \kappa(\ell_3) = L_3$$

Let us check premise SF2 for two transitions. For transition ℓ_2, we have to show

$$\begin{array}{l} at\_\ell_2 \;\wedge\; at'\_\ell_3 \;\wedge\; u' = u \;\wedge\; v' = v + u \;\wedge\; z' = z \quad \rightarrow \\ \qquad\qquad \underbrace{\Pi'[\alpha] = \Pi[\alpha] \;\wedge\; U'[\alpha] = U[\alpha] \;\wedge\; V'[\alpha] = V[\alpha] \;\wedge\; z' = z}_{\tau_I^{\mathcal{A}}[\alpha]} \end{array}$$

The conjunction $at\_\ell_2 \wedge at'\_\ell_3$ implies $\Pi[\alpha] = \Pi'[\alpha] = 3$, $U'[\alpha] = U[\alpha] = u - 2$, $V[\alpha] = v$, and $V'[\alpha] = v' - u'$. Together with $u' = u$ and $v' = v + u$, this yields

$$V'[\alpha] = v' - u' = (v + u) - u = v = V[\alpha],$$

as required.

Premise SF2 for transition ℓ_3 is

$$\begin{array}{l} at\_\ell_3 \;\wedge\; at'\_\ell_0 \;\wedge\; u' = u \;\wedge\; v' = v \;\wedge\; z' = z + 1 \quad \rightarrow \\ \underbrace{at\_L_3 \;\wedge\; at'\_L_0 \;\wedge\; U'[\alpha] = U[\alpha] + 2 \;\wedge\; V'[\alpha] = V[\alpha] + U[\alpha] + 2 \;\wedge\; z' = z + 1}_{L_3[\alpha]}. \end{array}$$

The conjunction $at\_\ell_3 \wedge at'\_\ell_0$ implies at_L_3, at'_L_0, $U[\alpha] = u - 2$, $V[\alpha] = v - u$, $U'[\alpha] = u'$, and $V'[\alpha] = v'$. Together with $u' = u$ and $v' = v$, this yields $U'[\alpha]U[\alpha] + 2$ and $V'[\alpha] = V[\alpha] + U[\alpha] + 2$.

We conclude that program $\textsc{square-root}$ refines program $\textsc{square-root}^{\mathcal{A}}$ with the refinement mapping α. Since formula $\Psi: at\_L_4 \rightarrow z^2 \leq x < (z+1)^2$ was proven to be an invariant of program $\textsc{square-root}^{\mathcal{A}}$, we invoke claim 5 to conclude that

$$\psi = \Psi[\alpha]: \quad at\_\ell_4 \quad \rightarrow \quad z^2 \leq x < (z+1)^2$$

is an invariant of program $\textsc{square-root}$.

Properties Inherited from Program MUX-PROT to Program PET

It is straightforward to apply rules INV and rule RESP to establish the following properties of the prototype mutual exclusion program MUX-PROT:

Mutual Exclusion	Ψ_0 :	$\square \neg(at\_L_3 \ \wedge \ at\_M_3)$
Accessibility for process P_1	Ψ_1 :	$at\_L_2 \ \Rightarrow \ \diamondsuit \, at\_L_3$
Accessibility for process P_2	Ψ_2 :	$at\_M_2 \ \Rightarrow \ \diamondsuit \, at\_M_3$

Previously, we have shown that program PET refines program MUX-PROT with the following refinement substitution:

$$\Pi_1[\alpha] = \textbf{if } at\_\ell_{4,5} \textbf{ then } 3 \textbf{ else if } at\_\ell_{2,3} \textbf{ then } 2 \textbf{ else } \pi_1$$
$$\Pi_2[\alpha] = \textbf{if } at\_m_{4,5} \textbf{ then } 3 \textbf{ else if } at\_m_{2,3} \textbf{ then } 2 \textbf{ else } \pi_2$$

Consequently, we can derive the properties of mutual exclusion and accessibility for program PET, by applying the substitution α to the three properties $\Psi_0 - \Psi_2$. This yields the following properties for program PET:

$$\begin{array}{ll} \psi_0 = \Psi_0[\alpha] : & \square \neg(at\_\ell_{4,5} \ \wedge \ at\_m_{4,5}) \\ \psi_1 = \Psi_1[\alpha] : & at\_\ell_{2,3} \ \Rightarrow \ \diamondsuit \, at\_\ell_{4,5} \\ \psi_2 = \Psi_2[\alpha] : & at\_m_{2,3} \ \Rightarrow \ \diamondsuit \, at\_m_{4,5} \end{array}$$

Properties Inherited from Program PET to Program PET2

In a similar way, we can port these properties from program PET1 (equivalent to PET) to program PET2. An interesting point is that, in refining program MUX-PROT into program PET, we have acquired a new property, possessed by PET, but not by its ancestor MUX-PROT. This is the property of *bounded overtaking*.

Program MUX-PROT required a certain degree of even-handedness on the treatment of the two processes. By making the transitions associated with the statement *enter* compassionate, we ensure that no process will be stuck forever at an enter statement, while its rival breezes in and out of its critical section. Such an absolute discrimination against one of the processes is ruled out by the requirement of compassion. However, the program still allows process P_2, for example, to enter twice its critical section for every single entry of process P_1.

The situation is much more regulated in program PET. Here, it is impossible that, while process P_1 is continually waiting at location ℓ_3, process P_2 enters twice its critical section at location m_4.

We say that a mutual exclusion program has the *k-bounded overtaking* property if one of the processors cannot successively enter its critical section more than k times while its competitor is continually waiting to enter its critical section.

For program PET, the property of 1-bounded overtaking can be expressed by the formula

$$\psi_3\!: \quad at\_\ell_3 \ \Rightarrow \ (\neg at\_m_4) \ \mathcal{W} \ at\_m_4 \ \mathcal{W} \ (\neg at\_m_4) \ \mathcal{W} \ at\_\ell_4.$$

Formula ψ_3 is a nested waiting-for formula, which should be interpreted by association to the right. Thus, the above formula is equivalent to the following fully parenthesized form:

$$at\_\ell_3 \;\Rightarrow\; (\neg at\_m_4)\; \mathcal{W}\; (at\_m_4\; \mathcal{W}\; ((\neg at\_m_4)\; \mathcal{W}\; at\_\ell_4)).$$

Formula ψ_3 states that any position satisfying at_ℓ_3 is followed by an interval in which P_2 is not at m_4, followed by an interval in which P_2 is at m_4, followed by an interval in which P_2 is again not at m_4, which can be terminated only by a state in which P_1 is at ℓ_4. It therefore implies that from the time P_1 starts waiting at ℓ_3, there can be at most one continuous interval in which P_2 is at m_4, before P_1 is admitted to ℓ_4. The case of P_2 not reaching m_4 at all is obtained by taking an empty at_m_4 interval.

We can refer the reader to [MP94] for a rule that is sound and complete for proving nested waiting-for formulas such as ψ_3. This rule can be used to establish the validity of ψ_3 over program PET.

On refining program PET, in its renamed form PET1, into program PET2, we can use property inheritance to derive the four main properties of program PET2, which can be written as:

Mutual Exclusion	$\chi_0:$	$\Box\,\neg(at\_\ell_5 \;\wedge\; at\_m_5)$
Accessibility for P_1	$\chi_1:$	$at\_\ell_4 \;\Rightarrow\; \Diamond\, at\_\ell_5$
Accessibility for P_2	$\chi_2:$	$at\_m_4 \;\Rightarrow\; \Diamond\, at\_m_5$
Bounded Overtaking	$\chi_3:$	$at\_\ell_4 \;\Rightarrow$ $(\neg at\_m_5)\; \mathcal{W}\; at\_m_5\; \mathcal{W}\; (\neg at\_m_5)\; \mathcal{W}\; at\_\ell_5$

An interesting lesson that can be learned from this example is that, when we have a development sequence of programs each refining its predecessor, it is recommended to establish their main properties as early as possible, and then just port the accumulated sum of properties along the sequence. For example, in the mutual exclusion development sequence, the exclusion and accessibility properties were already possessed by the first program in the sequence, program MUX-PROT. Bounded overtaking appeared first in program PET, and then was inherited from this level on.

6 Having your Next-Time Operator and Eating it Too

One of the central debates among proponents of different versions of temporal logic has been whether the *next-time* operator $\bigcirc$ should be included in the logic. The axiomatic systems presented in [GPSS80] and [BAMP83] for propositional linear-time and branching-time temporal logics relied very strongly on the presence of the *next* operator to formulate axioms such as the induction axiom

$$(p \Rightarrow \bigcirc p) \rightarrow (p \Rightarrow \Box p)$$

This axioms states that if formula p propagates from any state to its successor then, once p is true, it remains true ever after. The induction axiom serves as the basis for verification rules such as the rule used for establishing the validity of a safety formula $\Box p$ over all computations of a given program. It is also a central element in tableaux-based decision methods for propositional temporal logics and model checking, using expansion theorems such as

$$p \,\mathcal{U}\, q \quad \Leftrightarrow \quad q \vee \Big(p \wedge \bigcirc(p \,\mathcal{U}\, q)\Big)$$

for all the temporal operators.

The opposition to the next-time operator [Lam83b] is based on the argument that two programs such as

$$P_1 : \textbf{loop forever do } [x \ := \ x+1]$$
$$P_2 : \textbf{loop forever do } [x \ := \ x+1; \quad \textbf{skip}]$$

should be considered equivalent, even though P_1 appears to increment x at every step while P_2 increments x only every other step.

This argument led to the requirement that the semantics of programs should be *stuttering closed*, i.e., should contain, together with each state sequence σ, all of its *stuttering variants* which are obtained from σ by duplicating some states an arbitrary finite number of times or deleting some consecutive duplicates. Consistently with this requirement, the argument continues, the logical specification language should be insensitive to stuttering, i.e., unable to distinguish between a state sequence and any of its stuttering variants.

Unfortunately, the formula

$$\Box(x^{+} = x+1)$$

holds over the state sequence

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

but not over its stuttering variant

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

The inevitable conclusion is that the specification logic should not contain the next-time operator which is the one that distinguishes between these two sequences. Such, indeed, is the logic proposed in [Lam83b] for specifying properties of programs.

However, as soon as one uses the logic for describing the full semantics of programs, as is necessary for proving equivalence between programs or that one program implements another, the next-time operator is crucially needed to express the effect of transitions on the state.

Indeed, the logic TLA [Lam91], recently proposed by Lamport, reintroduces the notation of primed variables: $x' = x+1$, that essentially means $x^{+} = x+1$. To guarantee that stuttering robustness is maintained, it is required that the primed variables notation be used only in subformulas of specific form, where it can do no harm.

In the following sections, we introduce a temporal logic TLR that reconciles the apparently conflicting wishes to have an unrestricted next-time operator while maintaining stuttering robustness. Models for the logic are sequences of instantaneous descriptions, alternating between *persistent* descriptions and *transient* descriptions. All instantaneous descriptions provide local interpretation of the state variables. Persistent descriptions represent states that persist along some interval of positive duration. Transient descriptions represent transient situations that often describe change in the values of some state variables. We refer to instantaneous descriptions of the two sorts also as *persistent states* and *transient states*. A TLR model is presented as a sequence of states $s_0, s_1, \ldots$ where all the even-indexed states $s_0, s_2, \ldots$ are interpreted as persistent states while the odd-indexed states $s_1, s_3, \ldots$ are interpreted as transient states.

The key idea in TLR is that, unlike standard TL, the *next* operator does not always shift the reference point from s_j to s_{j+1}. If j is even, i.e., s_j is a persistent state, the reference points remain at j. If j is odd, i.e., s_j is a transient state, then the *next* operator shifts the reference point to $j+1$. Thus, we may describe the operation of the next operator as shifting the reference point to the nearest even position.

A preliminary version of TLR was proposed in [BKP86]. The semantics presented there combines a dense-time model with the *finite variability* assumption, by which any state variable may change its value only finitely many times in any finite interval. The density of the underlying time structure is the factor that guarantees stuttering robustness, since between any two sampling points, we can always introduce finitely many additional sampling points.

Unfortunately, the axiomatic system presented in [BKP86] contains several errors and omissions. Our first efforts in reviving the logic started by correcting these errors and deriving a complete axiomatic system and a decision procedure, while retaining the dense-time semantics. This work is reported in Appendix A, and a preliminary version of it appeared at [Pnu92]. The discrete semantics emerged out of the observation that every dense model that obeys the finite variability restriction has at most countably many distinct states. It is possible to represent such a model by a sequence of sampled states, interpreting some of them as representing persistent states and the others as representing transient states.

Other dense models with the finite variability or similar hypotheses were discussed in [Sta84], [Sta88], and [DK90]. However, none of these papers investigated the questions of decidability and completeness of these logics, and all of them restricted their attention to interpretations that are constant over half-open, half closed intervals, an assumption not made in our version.

The plan of the coming sections is as follows. In Section 7, we introduce the syntax and semantics of the logic TLR. In Section 8, we present an exponential-time decision procedure for satisfiability of propositional TLR formulas. In Section 9, we present an axiomatic system for propositional TLR, and prove its completeness. Finally, in Section 10, we illustrate the use of TLR for proving implementation (and equivalence) relations between programs. The appendix presents the dense finite-variability semantics for TLR.

We find it particularly intriguing that the same logic with the same complete axiomatic system can be based on both a discrete semantics and a dense one.

For purposes of comparison with TLR, we refer to the temporal logic that is introduced in section 2 as *regular temporal logic* or, simply, as TL.

7 The Logic TLR

We use the notions of vocabulary, states, rigid and flexible variables, and the underlying assertion language $\mathcal{L}$, as they were introduced in section 2.

A TLR *formula* is constructed out of terms that are combined into formulas, as follows:

- Each variable $y \in \mathcal{V}$ is a term. Further terms are obtained by applying functions of $\mathcal{L}$, such as integer addition and multiplication, to existing terms.
- If t is a term, then so are t^+, called *the next value of* t, and t^-, called *the previous value of* t.
- A predicate of $\mathcal{L}$ applied to terms is a formula, e.g., $x^+ > y - 1$.
- If p and q are formulas, then so are

$$\neg p, \qquad p \vee q, \qquad \bigcirc p, \qquad \ominus p, \qquad p\, \mathcal{U}\, q, \qquad \forall x.\, p, \qquad \exists x.\, p$$

A formula that does not contain any temporal operators (including the *next* and *previous value* operators applied to terms) is called a *state formula* or an *assertion*.

7.1 Semantics

As in the case of regular temporal logic, a model is an infinite sequence of states.

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

In TLR, we view the even-numbered states $s_0, s_2, \ldots$ as representing *persistent states*, i.e., states that persist along a non-empty open interval, and the odd-numbered states $s_1, s_3, \ldots$ as representing *transient* states that last for zero time. Thus, the history associated with model σ can be depicted as

$$(\text{—}\ s_0\ \text{—})\ s_1\ (\text{—}\ s_2\ \text{—})\ s_3\ (\text{—}\ s_4\ \text{—})\ \cdots$$

An *even-even stretching function* $f : \mathbb{N} \mapsto \mathbb{N}$ is a function on the naturals, which is monotonically nondecreasing and satisfies

- $|\{m \mid f(m) = n\}|$ is *odd* for all n. That is, the number of natural numbers m such that $f(m) = n$ is odd.
- $|\{m \mid f(m) = n\}| = 1$ for all odd n. That is, for every odd n, there exists precisely one m such that $f(m) = n$.

For an even-even stretching function f, we call model σ' an *f-stretching* of a model σ if $\sigma'_n = \sigma_{f(n)}$, for every $n \in \mathbb{N}$.

Consider, for example, the function f whose sequence of first values is given by

$$\begin{array}{ll} f(0) = f(1) = f(2) = 0, & f(3) = 1, \\ f(4) = f(5) = f(6) = 2, & f(7) = 3, \quad \cdots \end{array}$$

The model

$$\sigma' : \quad s_0,\ s_0,\ s_0,\ s_1,\ s_2,\ s_2,\ s_2,\ s_3,\ \ldots$$

is an f-stretching of the model $\sigma : s_0, s_1, s_2, \ldots$. The requirements imposed on even-even stretching functions ensure that every state in σ is mapped on at least one state in σ', and that every odd-numbered state in σ is mapped to a unique odd-numbered state in σ'. Thus, odd-numbered states are not duplicated at all, while even-numbered states may be duplicated an even number of times. This explains the name given to even-even stretching functions.

As a result of these requirements, even-even stretching preserves the identity of the transient states. This can be seen in the following presentation of σ'.

$$(\text{—}\ s_0\ \text{—})\ s_0\ (\text{—}\ s_0\ \text{—})\ s_1\ (\text{—}\ s_2\ \text{—})\ s_2\ (\text{—}\ s_2\ \text{—})\ s_3 \cdots$$

Model σ' is an *even-even stretching* of model σ if it is an f-stretching of σ for some even-even stretching function f.

We say that model σ' is a *stuttering variant* of σ if there exists a model σ'' which is an even-even stretching of both σ and σ'. For a variable $x \in \mathcal{V}$, $\sigma' : s'_0, s'_1, \ldots$ is called an *x-variant* of $\sigma : s_0, s_1, \ldots$ if s'_j agrees with s_j on the interpretation of all variables in $\mathcal{V}$, except possibly x, for every $j \in \mathbb{N}$.

TLR formulas are interpreted over a model σ and position $j \in \mathbb{N}$. First, we define the value of a term t at position j in σ, denoted by $val(\sigma,\ j,\ t)$.

- For a variable $x \in \mathcal{V}$,
  $val(\sigma,\ j,\ x) = s_j[x]$
  That is, the value of x at j equals the interpretation of x by the state s_j.
- For a function f and terms $t_1, \ldots, t_k$,
  $val(\sigma,\ j,\ f(t_1, \ldots, t_k)) = I[f](val(\sigma,\ j,\ t_1) \ldots, val(\sigma,\ j,\ t_k))$.
  That is, the value obtained by applying the interpretation of f to the values $val(\sigma,\ j,\ t_1) \ldots, val(\sigma,\ j,\ t_k)$.
- For a term t,
  $val(\sigma,\ 2 \cdot j\ ,\ t^+) = val(\sigma,\ 2 \cdot j\ ,\ t)$
  $val(\sigma,\ 2 \cdot j + 1,\ t^+) = val(\sigma,\ 2 \cdot j + 2,\ t)$
  Thus, the next value of a term at any position i is the value of this term at the lowest even position not smaller than i.
- For a term t,
  $val(\sigma,\ 2 \cdot j\ ,\ t^-) = val(\sigma,\ 2 \cdot j,\ t)$
  $val(\sigma,\ 2 \cdot j + 1,\ t^-) = val(\sigma,\ 2 \cdot j,\ t)$
  Thus, the previous value of a term at any position i is the value of this term at the highest even position not larger than i.

Next, we define the notion of a formula p holding at position j in a model σ, denoted by $(\sigma, j) \models p$.

- For a predicate P and terms $t_1, \ldots, t_k$,
  $(\sigma, j) \models P(t_1, \ldots, t_k) \iff I[P](val(\sigma,\ j,\ t_1) \ldots, val(\sigma,\ j,\ t_k)) = \mathrm{T}$
- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma,\ 2 \cdot j\ ) \models \bigcirc p \iff (\sigma,\ 2 \cdot j\ ) \models p$
- $(\sigma,\ 2 \cdot j + 1) \models \bigcirc p \iff (\sigma,\ 2 \cdot j + 2) \models p$
- $(\sigma,\ 2 \cdot j\ ) \models \ominus p \iff (\sigma,\ 2 \cdot j\ ) \models p$
- $(\sigma,\ 2 \cdot j + 1) \models \ominus p \iff (\sigma,\ 2 \cdot j\ ) \models p$
- $(\sigma, j) \models p\,\mathcal{U}\,q \iff$ for some $k \geq j, (\sigma, k) \models q$, and $(\sigma, i) \models p$, for all $i,\ j \leq i < k$.
- $(\sigma, j) \models \exists x.\ p \iff (\sigma'', j') \models p$ for some σ' an f-stretching of σ, σ'' an x-variant of σ', and some position $j' \in f^{-1}(j)$.
- $(\sigma, j) \models \forall x.\ p \iff (\sigma'', j') \models p$ for all σ' an f-stretching of σ, σ'' an x-variant of σ', and all positions $j' \in f^{-1}(j)$.

A formula p is said to *hold* over model σ, denoted $\sigma \models p$ if $(\sigma, 0) \models p$. Formula p is called *satisfiable* if it holds over some model. It is called *valid*, denoted $\models p$ if p holds over all models.

Claim 6 *The satisfiability of a* TLR *formula p is invariant to (even-even) stuttering. That is, p holds on σ if and only if it holds on every stuttering variant of σ.*

In a model σ, we refer to all even positions as *persistent positions* and to all odd positions as *transient positions*.

In the following two sections, we concentrate on PTLR, the propositional fragment of TLR. This logic is obtained by restricting our attention to formulas in which all variables are boolean, and excluding the quantifiers. We refer to boolean variables also as propositions.

8 A Decision Procedure for PTLR

In this section, we present a tableau algorithm for the satisfiability problem of PTLR, the propositional fragment of TLR. The algorithm can also be used for testing validity by checking the negation of the considered formula for satisfiability.

Let φ be the PTLR formula whose satisfiability we wish to check.

Closure and Atoms

Definition: The *closure* of φ, $CL(\varphi)$, is the smallest set of formulas containing φ and satisfying:

- $\mathrm{T}, \mathrm{F} \in CL(\varphi)$.

- $p \in CL(\varphi)$ iff $\neg p \in CL(\varphi)$ (We identify $\neg\neg p$ with p and $\neg$T with F).
- If $p \wedge q \in CL(\varphi)$ then $p, q \in CL(\varphi)$.
- If $p\,\mathcal{U}\,q \in CL(\varphi)$ then $p, q \in CL(\varphi)$.
- $p \in CL(\varphi)$ iff $\bigcirc p \in CL(\varphi)$ (We identify $\bigcirc\bigcirc p$ with $\bigcirc p$ and $\bigcirc\neg p$ with $\neg\bigcirc p$.)
- $p \in CL(\varphi)$ iff $\ominus p \in CL(\varphi)$ (We identify $\ominus\ominus p$ with $\ominus p$, $\ominus\bigcirc p$ with $\ominus p$, $\bigcirc\ominus p$ with $\bigcirc p$, $\ominus\neg\ominus p$ with $\ominus\neg p$ and $\bigcirc\neg\ominus p$ with $\bigcirc\neg p$.)

Definition: A *maximal saturated set* M is a subset of $CL(\varphi)$ such that

- T, $\bigcirc$T $\in M$.
- For every $p \in CL(\varphi)$,
  $$p \in M \quad \text{iff} \quad \neg p \notin M.$$
- For every $\ominus p \in CL(\varphi)$,
  $$\ominus p \in M \quad \text{iff} \quad \ominus\neg p \notin M.$$
- For every $p \wedge q \in CL(\varphi)$,
  $$p \wedge q \in M \quad \text{iff} \quad p \in M \text{ and } q \in M.$$
- For every $p\,\mathcal{U}\,q \in CL(\varphi)$,
  $$p\,\mathcal{U}\,q \in M \quad \text{iff} \quad q \in M \text{ or } \{p, \bigcirc(p\,\mathcal{U}\,q)\} \subseteq M.$$
  $$\ominus(p\,\mathcal{U}\,q) \in M \quad \text{iff} \quad \ominus q \in M \text{ or } \{\ominus p, p\,\mathcal{U}\,q\} \subseteq M.$$

An *atom* is a pair $\langle t, F\rangle$, where F is a maximal saturated set of formulas and $t \in \{P, T\}$ is the atom's *type*. An atom of type P is called a *persistent atom*. An atom of type T is called a *transient atom*. An atom $A = \langle t, F\rangle$ is *continuous* if, for all $p \in CL(\varphi)$,

$$\ominus p \in F \text{ iff } p \in F \text{ iff } \bigcirc p \in F.$$

An atom which is not continuous is called *discontinuous*. We require that all persistent atoms be continuous.

The set of all atoms associated with a formula φ is denoted by A^φ. For an atom A, we denote by t_A the type of A, and by F_A the set of formulas belonging to A. When there is no danger of confusion, we write $p \in A$ as abbreviation for $p \in F_A$.

Tableau Construction

Let φ be a PTLR formula. Let $\mathcal{A}_P, \mathcal{A}_T$ be the sets of persistent and transient atoms respectively, associated with φ.

A *structure for the formula* φ is a graph $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi\rangle$, where

- $\mathcal{A}^\varphi = \mathcal{A}_P \cup \mathcal{A}_T$ is the set of nodes.
- $\mathcal{R}^\varphi$ is a set of edges, defined by the following relation:
  For $P \in \mathcal{A}_P$ and $T \in \mathcal{A}_T$

$$(T, P) \in \mathcal{R}^\varphi \iff \left\{ \begin{array}{c} \text{for every } \bigcirc p \in CL(\varphi) \\ \bigcirc p \in T \text{ iff } p \in P \end{array} \right.$$

$$(P, T) \in \mathcal{R}^\varphi \iff \left\{ \begin{array}{c} \text{for every } \ominus p \in CL(\varphi) \\ \ominus p \in T \text{ iff } p \in P \end{array} \right.$$

An immediate result from the definition of $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$ is that a continuous transient atom T can only be connected to a persistent atom P such that $F_P = F_T$. We use the notation P_T and say that P_T is the persistent atom associated with the continuous transient atom T.
The following definitions are taken from [LPZ85] with appropriate modifications:

- An *initial atom* is a persistent atom $P \in \mathcal{A}_P$ such that $\varphi \in P$.
- A *path* $\pi = A_0, A_1, \ldots$ in $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$ is an infinite sequence of atoms, such that A_0 is initial and $(A_i, A_{i+1}) \in \mathcal{R}^\varphi$, for every $i \geq 0$. Obviously, all atoms appearing at even positions are persistent and all those appearing at odd positions are transient.
- A path $\pi = A_0, A_1, \ldots$ in $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$ is a *fulfilling path for* φ if, for every $i \geq 0$ and every $p\,\mathcal{U}\,q \in A_i$, there exists some $j \geq i$ such that $q \in A_j$.

Lemma 1. *Let $A_0, A_1, \ldots$ be a path in $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$. Let $p\,\mathcal{U}\,q \in A_j$ and $q \notin A_i$ for all i, $j \leq i < k$, for some $k \geq j$. Then $p, p\,\mathcal{U}\,q \in A_i$ for all i, $j \leq i < k$.*

The procedure for checking satisfiability of a formula φ attempts to find a model for φ in the structure $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$.
For a model σ, we denote by $\pi_\sigma = A_0, A_1, \ldots$ the fulfilling path associated with σ, where

$$F_{A_i} = \{p \in CL(\varphi) \mid (\sigma, i) \models p\}$$

On the other hand, every path in the tableau $\pi = A_0, A_1, \ldots$, induces a model $\sigma_\pi : s_0, s_1, \ldots$, obtained by taking

$$s_j[p] = \begin{cases} \mathrm{T} & \text{if } \ p \in A_j \\ \mathrm{F} & \text{if } \neg p \in A_j \end{cases}$$

for every $j \geq 0$, and boolean variable (proposition) $p \in CL(\varphi)$. The values of $s_j[x]$ for other variables $x \in \mathcal{V}$ can be determined arbitrarily.

Theorem 2. *Let φ be a* PTLR *formula. Then φ is satisfiable iff there exists a fulfilling path for φ in $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$.*

Proof: If φ is satisfiable, and σ is a model for φ, then it is easy to check that π_σ is a fulfilling path for φ.
Assume $\pi = A_0, A_1, \ldots$ is a fulfilling path for φ. We claim that σ_π is a model for φ.

Claim 7 *For every $p \in CL(\varphi)$ and every $j \geq 0$, $p \in A_j$ iff $(\sigma_\pi, j) \models p$.*

We prove this claim by induction on the structure of p. Within the proof we use the abbreviated notation A for F_A.

- p is a proposition:
  $p \in A_j \iff s_j[p] = \mathrm{T} \iff (\sigma_\pi, j) \models p$.
- p is the formula $\neg\chi$:
  $p \in A_j \iff \neg\chi \in A_j \overset{\text{atom-def.}}{\iff} \chi \notin A_j \overset{\text{ind.-hyp.}}{\iff} (\sigma_\pi, j) \not\models \chi \overset{\text{semantic-def}}{\iff} (\sigma_\pi, j) \models \neg\chi$.

- p is the formula $\chi_1 \wedge \chi_2$:
$p \in A_j \iff \chi_1 \wedge \chi_2 \in A_j \overset{\text{atom-def}}{\iff} \chi_1, \chi_2 \in A_j \overset{\text{ind.-hyp.}}{\iff}$ $(\sigma_\pi, j) \models \chi_1$ and $(\sigma_\pi, j) \models \chi_2 \overset{\text{semantic-def.}}{\iff} (\sigma_\pi, j) \models \chi_1 \wedge \chi_2$.
- p is the formula $\bigcirc \chi$:
  (a) j is even: $p \in A_j \iff \bigcirc \chi \in A_j \overset{\text{persistent-atom-def}}{\iff} \chi \in A_j \overset{\text{ind.-hyp.}}{\iff}$
  $(\sigma_\pi, j) \models \chi \overset{\text{semantic-def.}}{\iff} (\sigma_\pi, j) \models \bigcirc \chi$.
  (b) j is odd: $p \in A_j \iff \bigcirc \chi \in A_j \overset{\mathcal{R}_\varphi\text{-def.}}{\iff} \chi \in A_{j+1} \overset{\text{ind.-hyp.}}{\iff}$
  $(\sigma_\pi, j+1) \models \chi \overset{\text{semantic-def.}}{\iff} (\sigma_\pi, j) \models \bigcirc \chi$
- p is the formula $\ominus \chi$:
  (a) j is even: $p \in A_j \iff \ominus \chi \in A_j \overset{\text{persistent-atom-def.}}{\iff} \chi \in A_j \overset{\text{ind.-hyp.}}{\iff}$
  $(\sigma_\pi, j) \models \chi \overset{\text{semantic-def.}}{\iff} (\sigma_\pi, j) \models \ominus \chi$.
  (b) j is odd: $p \in A_j \iff \ominus \chi \in A_j \overset{\mathcal{R}_\varphi\text{-def.}}{\iff} \chi \in A_{j-1} \overset{\text{ind.-hyp.}}{\iff}$
  $(\sigma_\pi, j-1) \models \chi \overset{\text{semantic-def.}}{\iff} (\sigma_\pi, j) \models \ominus \chi$
  Note that since j is odd, $j \geq 1$.
- p is the formula $\chi_1 \,\mathcal{U}\, \chi_2$:
From the fact that π is a fulfilling path and lemma 1, there exists a $k \geq j$ such that $\chi_2 \in A_k$ and for all i, $j \leq i < k$, $\chi_1 \in A_i \overset{\text{ind.-hyp.}}{\iff}$ for every $i, j \leq i < k, (\sigma_\pi, i) \models \chi_1$ and $(\sigma_\pi, k) \models \chi_2 \overset{\text{semantic-def.}}{\iff} (\sigma_\pi, j) \models \chi_1 \,\mathcal{U}\, \chi_2$.

Since π is a fulfilling path for φ, then $\varphi \in A_0$ which, together with the above claim, imply $\sigma_\pi \models \varphi$.

To determine the satisfiability of φ, it is sufficient to look for a fulfilling path for φ in $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$. Fulfilling paths are found by decomposing $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$ into maximal strongly connected subgraphs (*mscs*), as described in [LPZ85] for PTL.

Definition: Let $\mathcal{S}$ be a strongly connected subgraph (*scs*) in $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$. Subgraph $\mathcal{S}$ is *self-fulfilling* if, for every atom $A \in \mathcal{S}$ and formula $p \,\mathcal{U}\, q \in F_A$, there exists an atom $B \in \mathcal{S}$ such that $q \in F_B$.

Lemma 3. *Let $\mathcal{S}$ and $\mathcal{S}'$ be scs's such that $\mathcal{S} \subseteq \mathcal{S}'$. If $\mathcal{S}$ is self-fulfilling then so is $\mathcal{S}'$.*

A fulfilling path for φ must start at an initial atom and eventually remain contained in some self-fulfilling *scs*. The search for a fulfilling path proceeds by successively removing from $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$ *mscs* that can not participate in a fulfilling path.

Definitions: Let $\mathcal{S}$ be an *mscs* in $\langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$.

- $\mathcal{S}$ is *terminal* if it has no outgoing $\mathcal{R}^\varphi$ edges.
- $\mathcal{S}$ is *useless* if it satisfies one of the following:

– $\mathcal{S}$ is terminal and not self-fulfilling.
– $\mathcal{S}$ is terminal and consists of a single atom.

The Satisfiability Algorithm

let $\langle \mathcal{A}^0, \mathcal{R}^0 \rangle = \langle \mathcal{A}^\varphi, \mathcal{R}^\varphi \rangle$ be the initial structure.
repeat with the last defined structure $\langle \mathcal{A}^i, \mathcal{R}^i \rangle$
  let $\mathcal{S}$ be a useless *mscs* in $\langle \mathcal{A}^i, \mathcal{R}^i \rangle$
  then define $\langle \mathcal{A}^{i+1}, \mathcal{R}^{i+1} \rangle$ by :
    $\mathcal{A}^{i+1} = \mathcal{A}^i - \mathcal{S}$
    $\mathcal{R}^{i+1} = \mathcal{R}^i \cap (\mathcal{A}^{i+1} \times \mathcal{A}^{i+1})$
until $\langle \mathcal{A}^i, \mathcal{R}^i \rangle$ is empty or contain no useless *mscs*.
if there is an initial atom in the final structure
  then report *success*
  else report *failure*

The final structure obtained by this removal process is called the *tableau for* φ. The soundness of the algorithm is based on the following lemma:

Lemma 4. *π is a fulfilling path in $\langle \mathcal{A}^i, \mathcal{R}^i \rangle$ iff it is a fulfilling path in $\langle \mathcal{A}^{i+1}, \mathcal{R}^{i+1} \rangle$.*

Theorem 5. *The formula φ is satisfiable iff the algorithm above reports success.*

9 Completeness

In this section, we present an axiomatic system for PTLR, and prove it complete. It is straightforward to show that the system is sound.

As basic temporal operators we take $\bigcirc$, $\ominus$ and $\mathcal{U}$. Additional operators are defined by:

$$\begin{array}{ll} \Diamond p \;\; = \mathrm{T}\, \mathcal{U}\, p & \qquad p \Rightarrow q = \Box (p \rightarrow q) \\ \Box p \;\; = \neg \Diamond \neg p & \qquad p \Leftrightarrow q = \Box (p \leftrightarrow q) \\ p\, \mathcal{W}\, q = \Box p \vee (p\, \mathcal{U}\, q) & \end{array}$$

Since all variables in PTLR are booleans, we can express x^+ as $\bigcirc x$ and x^- as $\ominus x$. Therefore, we do not consider expressions of the form x^+ and x^-.

Axioms and Rules

We use the following axioms:

FX0. $\square p \;\rightarrow\; p$

FX1. $\bigcirc \neg p \;\Leftrightarrow\; \neg \bigcirc p$ PX1. $\ominus \neg p \;\Leftrightarrow\; \neg \ominus p$

FX2. $\bigcirc(p \rightarrow q) \;\Leftrightarrow\; (\bigcirc p \rightarrow \bigcirc q)$ PX2. $\ominus(p \rightarrow q) \;\Leftrightarrow\; (\ominus p \rightarrow \ominus q)$

FX3. $\square(p \rightarrow q) \;\Rightarrow\; (\square p \rightarrow \square q)$

FX4. $\square p \;\rightarrow\; \square \bigcirc p$ PX4. $\square p \;\rightarrow\; \square \ominus p$

FX5. $(p \Rightarrow \bigcirc p) \wedge (\ominus p \Rightarrow p) \;\rightarrow\; (p \Rightarrow \square p)$

FX6. $(p\, \mathcal{U}\, q) \;\Leftrightarrow\; \big(q \;\vee\; [p \;\wedge\; \bigcirc(p\, \mathcal{U}\, q)]\big)$

PX6. $\ominus(p\, \mathcal{U}\, q) \;\Leftrightarrow\; \ominus q \;\vee\; \big[\ominus p \;\wedge\; (p\, \mathcal{U}\, q)\big]$

FX7. $p \mathcal{U} q \;\Rightarrow\; \diamondsuit q$

FX8. $\bigcirc p \;\Leftrightarrow\; \bigcirc \bigcirc p$ PX8. $\ominus p \;\Leftrightarrow\; \ominus \ominus p$

FX9. $\bigcirc \ominus p \;\Leftrightarrow\; \bigcirc p$ PX9. $\ominus \bigcirc p \;\Leftrightarrow\; \ominus p$

FX10. $\square \neg p \;\Leftrightarrow\; \neg \diamondsuit p$

FX11. $p \rightarrow \bigcirc p$ PX11. $p \rightarrow \ominus p$

Axiom scheme TGI: For every φ, a temporal instantiation of a state-valid formula.

$$\square \varphi$$

Axioms FX0 - FX4, FX6 - FX7 and PX2 - PX6 are part of the axiomatic system for regular temporal logic [MP91b], and will not be explained. We provide some commentary for the axioms which are specific to TLR.

- Axiom PX1 establishes the Previous operator $\ominus$ as self dual. This is a consequence of the fact that the first position (state) has an even index and applying $\ominus$ there leaves us at the same position. This is obviously not the case in TL, where any formula of the form $\ominus p$ evaluates to F on the initial state, because it has no predecessor.
- Axiom FX5 is the induction axiom. Unlike the induction axiom in TL, p needs to propagate not only from each position to its successor, but also from the predecessor of each position to the position itself.
- Axioms FX8, FX9 ,PX8, PX9 establish the *idempotence* property of both the Previous and the Next operators. Namely, any consecutive sequence of $\ominus$ and $\bigcirc$ can shift the reference point, forward or backward, by at most one position. Always ending up at an even position. This is obviously not the case in TL where a sequence of n Next (Previous) operators, shifts the reference point n positions forward (backward), and the combination $\bigcirc \ominus$ remains at the same place, after making one forward shift, followed by a backward shift.
- Axioms FX11 and PX11 establish that the initial position of a computation is a persistent one.

Rules

As a single inference rule we take:

Rule MP: $\quad p \to q,\ p \quad \vdash \quad q$

Derived Rules and Theorems

Using this deductive system, we can prove many derived rules and theorems. The ones we need for the proof of completeness can be summarized as follows:

- All the temporal operators are *monotonic*. That is, if $p \Rightarrow p'$ and $q \Rightarrow q'$, then $\otimes p \Rightarrow \otimes p'$, where $\otimes$ stands for any of the unary operators $\diamondsuit, \square, \bigcirc, \circleddash$, and $p \oplus q \Rightarrow p' \oplus q'$, where $\oplus$ stands for any of the binary operators $\mathcal{U}$, $\mathcal{W}$.
- The relation $\Leftrightarrow$ is a *congruence*. That is, if $p \Leftrightarrow q$ then $\varphi(p) \Leftrightarrow \varphi(q)$, where $\varphi(q)$ is obtained from $\varphi(p)$ by replacing one or more occurrences of p by q.
- Conjunction distributes over the operators $\square, \bigcirc, \circleddash$, and the first argument of $\mathcal{U}$ and $\mathcal{W}$.
- Disjunction distributes over the operators $\diamondsuit, \bigcirc, \circleddash$, and the second argument of $\mathcal{U}$ and $\mathcal{W}$.

Let φ be a valid PTLR formula. To show that φ is provable ($\vdash \varphi$), we follow the process of checking satisfiability of the formula $\neg\varphi$, using semantic tableaux. We will show that each step in the checking procedure, corresponds to a deductive step in the proof of $\vdash \varphi$.

Theorem 6. *The system is complete for* PTLR, *i.e., for every* PTLR *formula, if φ is valid then $\vdash \varphi$.*

Proof: Suppose φ is valid, thus $\neg\varphi$ is unsatisfiable. In the application of the satisfiability algorithm to $\neg\varphi$, we construct a sequence of graphs

$$\langle \mathcal{A}^{\neg\varphi}, \mathcal{R}^{\neg\varphi} \rangle = \langle \mathcal{A}^0, \mathcal{R}^0 \rangle,\ \ldots,\ \langle \mathcal{A}^n, \mathcal{R}^n \rangle$$

which eventually converges, leading to a graph with no initial atoms.
For an atom A we denote $\widehat{A} = \bigwedge_{p \in F_A} p$, that is, $\widehat{A}$ is the conjunction of all formulas contained in the formula set F_A of atom A.
Let $A, B \in \mathcal{A}^0$, $P \in \mathcal{A}^0_P$ and $T \in \mathcal{A}^0_T$.

Claim 6.1:

$$a. \quad \vdash \square\Big(\bigvee_{A \in \mathcal{A}^0} \widehat{A}\Big) \qquad\qquad b. \quad \vdash \square \bigcirc\Big(\bigvee_{P \in \mathcal{A}^0_P} \widehat{P}\Big)$$

Proof of b:

$$\vdash \square \bigcirc\Big(\bigvee_{A \in \mathcal{A}^0} \widehat{A}\Big) \qquad a, \text{FX}4 \tag{1}$$

We will show that, for every $A \in \mathcal{A}^0$, there exists a persistent atom $P_A \in \mathcal{A}_P^0$ such that

$$\bigcirc \widehat{A} \Rightarrow \bigcirc \widehat{P}_A$$

In view of (1), this establishes $\square \bigcirc (\bigvee_{P \in \mathcal{A}_P^0} \widehat{P})$.

For a persistent A, we take P_A to be A itself.
Let $T \in \mathcal{A}_T^0$ be a discontinuous transient atom. Then, for some $p \in CL(\varphi)$, either $p, \neg \bigcirc p \in T$ or $p, \neg \ominus p \in T$.

- If $p, \neg \bigcirc p \in T$, then

$$\vdash \bigcirc \widehat{T} \Rightarrow \text{F} \qquad \textbf{FX}8, \textbf{FX}1$$

- If $p, \neg \ominus p \in T$, then

$$\vdash \bigcirc \widehat{T} \Rightarrow \text{F} \qquad \textbf{FX}9, \textbf{PX}1$$

In both cases, $\bigcirc \widehat{T}$ entails F and, therefore, $\bigcirc \widehat{T} \Rightarrow \bigcirc \widehat{P}$ for any persistent atom P.

Let $T \in \mathcal{A}_T^0$ be a continuous transient atom. Then, there exists a persistent atom $P_T \in \mathcal{A}_P^0$, such that $\widehat{T} \Leftrightarrow \widehat{P}_T$, and we can take P_A to be P_T.

∎

Claim 6.2: For every $p \in CL(\varphi)$,

$$\vdash p \Leftrightarrow \bigvee_{A \in \mathcal{A}^0, p \in A} \widehat{A}$$

Proof: From claim 6.1-a.

Claim 6.3: Let $\mathcal{A}_{init}$ denote the subset of initial atoms in $\mathcal{A}^0$.

$$\neg\varphi \rightarrow \bigvee_{P \in A_{init}} \widehat{P}$$

Proof: From claim 6.2 and axioms **FX**0 **FX**11 and **PX**11.

Claim 6.4: For every transient atom $T \in \mathcal{A}_T^0$ and persistent atom $P \in \mathcal{A}_P^0$,

$$a. \vdash \widehat{T} \Rightarrow \bigcirc (\bigvee_{P \in \mathcal{A}_P^0} \widehat{P}) \qquad b. \vdash \widehat{P} \Rightarrow \bigcirc \widehat{P} \qquad c. \vdash \ominus \widehat{P} \Rightarrow \bigvee_{T \in \mathcal{A}_T^0} \widehat{T}$$

Proof: (a)– from claim 6.1-b. (b)– from the definition of persistent atoms and axioms **FX**1, **PX**1, **FX**8 and **FX**9. (c) – From claim 6.1-a and the fact that for every persistent atom $P \in \mathcal{A}_P^0$ there is a (continuous) transient atom $T \in \mathcal{A}_T^0$ such that $F_T = F_P$.

Claim 6.5: For every persistent $P \in \mathcal{A}_P^0$ and transient $T \in \mathcal{A}_T^0$,

a. If $(T,P) \notin \mathcal{R}^0$ then $\vdash \widehat{T} \Rightarrow \bigcirc(\neg \widehat{P})$
b. If $(P,T) \notin \mathcal{R}^0$ then $\vdash \ominus \widehat{P} \Rightarrow \neg \widehat{T}$

Proof: Consider the different reasons for $(T,P) \notin \mathcal{R}^0$:

case 1: There is a formula $\bigcirc p \in CL(\varphi)$ such that $\bigcirc p \in T$ and $\neg p \in P$.

1. $\widehat{P} \Rightarrow \neg p$	given
2. $\bigcirc \widehat{P} \Rightarrow \bigcirc(\neg p) \Leftrightarrow \neg \bigcirc p$	
3. $\bigcirc p \Rightarrow \neg \bigcirc \widehat{P} \Leftrightarrow \bigcirc(\neg \widehat{P})$	2, **FX**1
4. $\widehat{T} \Rightarrow \bigcirc p$	given
5. $\widehat{T} \Rightarrow \bigcirc(\neg \widehat{P})$	3, 4

case 2: There is a formula $\bigcirc p \in CL(\varphi)$ such that $\neg \bigcirc p \in T$ and $p \in P$. This case can be proven in a way similar to case 1.

Similarly, consider the different reasons for $(P,T) \notin \mathcal{R}^0$:

case 3: There is a formula $\ominus p \in CL(\varphi)$ such that $\ominus p \in T$ and $\neg p \in P$.

case 4: There is a formula $\ominus p \in CL(\varphi)$ such that $\neg \ominus p \in T$ and $p \in P$.

Note that if T is continuous, both $(T,P) \notin \mathcal{R}^0$ and $(P,T) \notin \mathcal{R}^0$, for all $P \in \mathcal{A}^0_P - \{P_T\}$. That is, the only persistent atom that can be connected to a continuous T is P_T.

Claim 6.6:

$$a. \quad \vdash \widehat{T} \Rightarrow \bigcirc\Big(\bigvee_{(T,P)\in\mathcal{R}^0} \widehat{P}\Big) \qquad\qquad b. \quad \vdash \ominus \widehat{P} \Rightarrow \bigvee_{(P,T)\in\mathcal{R}^0} \widehat{T}$$

Proof: From claims 6.4, 6.5 and $\bigcirc$ distributivity over disjunction.
Let $(\mathcal{R}^i)^*$ denote the reflexive transitive closure of $\mathcal{R}^i$.

Claim 6.7:

$$\widehat{A} \Rightarrow \Box\Big(\bigvee_{(A,B)\in(\mathcal{R}^0)^*} \widehat{B}\Big).$$

Proof: Let $A_0 \in \mathcal{A}^0$. Let us denote by α the formula

$$\bigvee_{(A_0,B)\in(\mathcal{R}^0)^*} \widehat{B}.$$

We have to show that

$$\widehat{A}_0 \Rightarrow \Box\, \alpha \qquad\qquad \text{for every } A_0 \in \mathcal{A}^0.$$

Due to reflexivity of $(\mathcal{R}^0)^*$,$(A_0, A_0) \in (\mathcal{R}^0)^*$ and we, therefore, have

$$\widehat{A}_0 \Rightarrow \bigvee_{(A_0,B)\in(\mathcal{R}^0)^*} \widehat{B} \tag{2}$$

Let $T \in \mathcal{A}_T^0$, be such that $(A_0, T) \in (\mathcal{R}^0)^*$. From claim 6.6-$a$, we know

$$\widehat{T} \Rightarrow \bigcirc(\bigvee_{(T,P)\in\mathcal{R}^0} \widehat{P}).$$

Since

$$\{A \mid (T, A) \in \mathcal{R}^0\} \subseteq \{A \mid (A_0, A) \in (\mathcal{R}^0)^*\}$$

it follows that

$$\widehat{T} \Rightarrow \bigcirc\, \alpha. \tag{3}$$

For $P \in A_P^0$, we get a similar result from claim 6.4-b.
Since entailment 3 is true for any $A \in \mathcal{A}^0$ such that $(A_0, A) \in (\mathcal{R}^0)^*$, we get

$$\left(\bigvee_{(A_0,A)\in(\mathcal{R}^0)^*} \widehat{A}\right) \Rightarrow \bigcirc\, \alpha$$

or

$$\alpha \Rightarrow \bigcirc\, \alpha \tag{4}$$

Let $P \in \mathcal{A}_P^0$, such that $(A_0, P) \in (\mathcal{R}^0)^*$. From claim 6.6-$b$

$$\circleddash\, \widehat{P} \Rightarrow \bigvee_{(P,T)\in\mathcal{R}^0} \widehat{T} \tag{5}$$

and as for $T \in \mathcal{A}_T^0$ for the previous case, we get

$$\circleddash\, \widehat{P} \Rightarrow \alpha \tag{6}$$

Let $T \in \mathcal{A}_T^0$. Then,

If T is discontinuous: $\circleddash\, \widehat{T} \Rightarrow \mathrm{F}$ (Similar to the proof of 6.1-b)
If T is continuous: $\circleddash\, \widehat{T} \Rightarrow \circleddash\, \widehat{P}_T \Rightarrow \bigvee_{(P_T,T')\in\mathcal{R}^0} \widehat{T}'$

from which we derive

$$\circleddash\, \widehat{T} \Rightarrow \alpha \tag{7}$$

From 6 and 7 we get by the distributivity of $\circleddash$ over disjunction

$$\circleddash\, \alpha \Rightarrow \alpha \tag{8}$$

The three entailments 2, 4 and 8 together with axiom FX5, conclude the proof of claim 6.7.

∎

We generalize claims 6.6 and 6.7 to $\mathcal{R}^i$, $1 \leq i \leq n$ and prove by induction on i, $0 \leq i \leq n$, the following three claims:
Claim 6.8: For every $A \in \mathcal{A}^0 - \mathcal{A}^i$,

$$\vdash \Box(\neg A)$$

This claim states that any removed atom can be proven to be contradictory.

Claim 6.9: Let $A, B \in \mathcal{A}^i$, $P \in \mathcal{A}^i_P$ and $T \in \mathcal{A}^i_T$.

$$\vdash a. \quad \widehat{T} \Rightarrow \bigcirc(\bigvee_{(T,P)\in\mathcal{R}^i} \widehat{P}) \qquad\qquad \vdash b. \quad \ominus\widehat{P} \Rightarrow \bigvee_{(P,T)\in\mathcal{R}^i} \widehat{T}$$

Claim 6.10:

$$\widehat{A} \Rightarrow \Box(\bigvee_{(A,B)\in(\mathcal{R}^i)^*} \widehat{B})$$

For $i = 0$, claim 8 is vacuously true, and claims 6.9, 6.10 reduce to claims 6.6, 6.7 respectively.

Consider the move from i to $i+1$. This move removes a strongly connected subgraph $\mathcal{S}$ from $\langle \mathcal{A}^i, \mathcal{R}^i \rangle$. To prove the inductive step for claim 6.8, we show that $\Box(\neg\widehat{A})$ is provable for every $A \in \mathcal{S}$.
Consider the two different reasons for $\mathcal{S}$ being declared useless in $\langle \mathcal{A}^i, \mathcal{R}^i \rangle$.

- $\mathcal{S}$ is terminal and not self-fulfilling.
  Since $\mathcal{S}$ is both strongly connected and not self-fulfilling, there is some formula $p\,\mathcal{U}\,q \in CL(\varphi)$, such that for every $A \in \mathcal{S}$, $p\,\mathcal{U}\,q \in A$, $q \notin A$. It follows that, for every $A \in \mathcal{S}$,

  $$\widehat{A} \Rightarrow (p\,\mathcal{U}\,q) \wedge \neg q$$

  Since $\mathcal{S}$ is terminal, any atom B such that $(A, B) \in (\mathcal{R}^i)^*$ is contained in $\mathcal{S}$. Thus, by claim 6.10

  $$\vdash \widehat{A} \Rightarrow p\,\mathcal{U}\,q \wedge \Box\neg q$$

  which, combined with axiom FX7, implies

  $$\vdash \Box(\neg\widehat{A})$$

- $\mathcal{S}$ is terminal and consists of a single atom A.
  Since no atom in our structures is connected to itself, this means that A has no successors. We consider separately the cases that A is transient or persistent.

– If A is transient then, from claim 6.9(a) and the fact that the set of $\mathcal{R}^i$-successors of A is empty, it follows that
$$\vdash \widehat{A} \Rightarrow \bigcirc \text{F}.$$
By axiom **FX1** and the definition of F, it follows that
$$\vdash \widehat{A} \Rightarrow \neg \bigcirc \text{T}.$$
On the other hand, since $\bigcirc$ T is included in the formula sets of all atoms, it follows that $\widehat{A} \Rightarrow \bigcirc$ T. We may conclude $\vdash \widehat{A} \Rightarrow$ F or, equivalently,
$$\vdash \neg\widehat{A}.$$
– The case of a maximal strongly connected component consisting of a persistent atom A alone is impossible. When A is first created as a member of $\mathcal{A}^0$, it is created together with a corresponding (continuous) transient atom T_A whose set of formulas coincides with F_A. The atoms A and T_A are connected to one another and are, therefore, included in the same maximal strongly connected component. From that point on, each step of the algorithm only removes complete *mscs*. Consequently, A and T_A are never separated.

Claim 6.9 for $\mathcal{R}^{i+1}$ is obtained from $\mathcal{R}^i$ and claim 6.8, namely, that every atom $A \in (\mathcal{A}^{i+1} - \mathcal{A}^i)$ satisfies $\vdash \square(\neg\widehat{A})$.
Repeating the proof of claim 6.7, but using $(\mathcal{R}^{i+1})^*$ instead of $(\mathcal{R}^0)^*$, we can derive claim 6.10 from claim 6.9. ■

We return now to the main proof. Since $\chi = \neg\varphi$ is unsatisfiable, there is no initial atom P in $\langle \mathcal{A}^n, \mathcal{R}^n \rangle$, such that $\neg\varphi \in P$. Consequently, by claim 6.8

$$\vdash \bigwedge_{P \in \mathcal{A}^0_{init}} \square(\neg\widehat{P}). \tag{9}$$

According to claim 6.3, we have

$$\vdash \neg\varphi \rightarrow \bigvee_{P \in \mathcal{A}^0_{init}} \widehat{P} \tag{10}$$

Combining 9 and 10, we get $\vdash \neg\neg\varphi$, which leads to

$$\vdash \varphi$$

■

10 Verification of Properties and Refinement in TLR

In this section, we briefly sketch the usage of TLR for verification of properties and simulation or refinement relations between programs.

At a first glance, it may appear that TLR is a considerably more complicated language than regular temporal logic. In particular, it seems as though one should always be aware of whether his current reference point is persistent (even-indexed) or transient (odd-indexed).

Some of the unique features of TLR is the idempotence of the *next* and *previous* operators, as expressed in axioms FX8 and FX9. The next-value and previous-value operators share this property, as expressed in the following two axioms:

$$\Box(x^{++} = x^{+}) \qquad \text{and} \qquad \Box(\mathrm{x}^{--} = \mathrm{x}^{-})$$

Another obvious difference from regular temporal logic is the induction axiom FX5, which requires propagation of p not just from each position to its successor, but also from the predecessor of a position to the position itself. The two requirements are unrelated, i.e., none of them necessarily implies the other.

Consider, for example, the following model, in which we list the assignments given to integer variable x.

$$\sigma : \quad \langle x : 0\rangle,\ \langle x : 1\rangle,\ \langle x : 2\rangle, \langle x : 3\rangle,\ \cdots$$

This model satisfies $x = 0$ and $(x = 0) \Rightarrow \bigcirc(x = 0)$, yet it does not satisfy $\Box(x = 0)$. The missing factor is that σ does not satisfy $\ominus(x = 0) \Rightarrow (x = 0)$. In particular, $\ominus(x = 0) \rightarrow (x = 0)$ does not hold at position 1 (second position) of σ, since $\ominus(x = 0)$ is true there, but $x = 0$ is false.

Left-Continuous Variables

Much of this extra complexity can be alleviated by assuming that all system variables are left-continuous. Similar to the partition of variables into rigid and flexible variables, we single out a subset of the flexible variables and call them *left-continuous* variables. For every TLR-model $\sigma : s_0, s_1, \ldots$ and every left-continuous variable x, it is required that

$$s_j[x] = s_{j-1}[x] \qquad\qquad \text{for every odd } j > 0.$$

Thus, if the left-continuous variable x is not constant over σ, it may change its value only on passage from an odd position to an even one.

All left-continuous variables satisfy the left-continuity axiom:

$$\text{LEFT-CONT.} \qquad \Box(x^{-} = x)$$

An assertion that only refers to rigid or left-continuous variables is called a *left-continuous assertion*. If φ is a left-continuous assertion, then it satisfies the simpler induction axiom, which is the one holding for regular temporal logic.

$$(\varphi \Rightarrow \bigcirc\, \varphi) \;\rightarrow\; (\varphi \Rightarrow \Box\, \varphi)$$

We require that all system variables are left-continuous. This leads to a significant simplification in the verification rules, and brings TLR much closer to regular temporal logic.

According to the connection, established in the appendix between the discrete and dense semantics of TLR, the requirement of left-continuity leads to dense computations which are sequences of contiguous *left-open, right-closed* intervals. This type of dense semantics is closely related to the semantics considered in [Sta84], [Sta88], and [DK90].

10.1 Systems, Modules, and Their Computations

As before, a fair transition module has the form $M : \langle V, L, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}\rangle$. We require that all variables in V (and therefore in L) be left-continuous.

Let σ be a TLR-model $\sigma : s_0, s_1, \ldots$, and $\tau \in \mathcal{T}$ be a transition. As before, we say that τ is *taken* at position $j \geq 0$ of σ if $\langle s_j, s_{j+1}\rangle \vDash \rho_\tau$. That is $\rho_\tau(V, V')$ evaluates to T when we interpret each $x \in V$ as $s_j[x]$ and each $x' \in V'$ as $s_{j+1}[x]$. State s_{j+1} is called a τ-successor of s_j.

A *computation* of M is a TLR-model σ that satisfies the following requirements.

- *Left-Continuity:* For every odd position $j > 0$ and every system variable $x \in V$, s_j and s_{j-1} agree on the value of x, i.e. $s_j[x] = s_{j-1}[x]$. This is a direct consequence of the requirements that all system variables are left-continuous.
- *Initiation:* s_0 is initial, i.e., $s_0 \vDash \Theta$.
- *Consecution:* For each odd position $j > 0$, state s_{j+1} is a τ-successor of state s_j.
- *Justice:* For each $\tau \in \mathcal{J}$, there is no position $j \geq 0$ such that τ is continually enabled but not taken beyond j.
- *Compassion:* For each $\tau \in \mathcal{C}$, it is not the case that τ is enabled at infinitely many positions in σ but taken at only finitely many times.

Note that while consecution only requires that transitions be taken at odd positions, the requirements of justice and compassion can be satisfied by transitions that are taken at even positions. However, a transition τ can be taken at an even position j only if $\langle s_j, s_j\rangle \vDash \rho_\tau$. This is because, due to left-continuity, $s_{j+1}[V] = s_j[V]$ for all even j's.

It can be shown that the set of computations of a module is closed under stuttering. That is, if σ is a computation of module M, then so is any stuttering variant of σ. This property of sets of computations is also true in the regular framework.

An *observation* of a module M is any TLR-model, which is a stuttering variant of an L-variant of a computation of M, i.e., any model that can be obtained from a computation by arbitrarily changing the values of variables in L and then applying a stuttering transformation to the resulting model.

The set of observations of a module M is obviously closed under stuttering. This closure property is unique to the TLR framework and does not hold in the framework of sections 2–5. It is fair to say that the stuttering closure of the observation set of a module is the main advantage of TLR over the regular temporal framework.

The Existence of Clocks of Arbitrary Rate

Assume a fixed transition module M. Let τ be a transition with transition relation ρ_τ, such that $\rho_\tau(V, V')$ implies $V \neq V'$, i.e., $x \neq x'$ for at least one system variable $x \in V$.

Let u be a left-continuous integer variable, which is not a system variable. We claim that the following formula is valid.

$$\textsc{rate-doubling}: \quad \exists u : u = 0 \wedge \Box \begin{pmatrix} \vee\ u = u^+ \\ \vee\ En(\rho_\tau)\ \wedge\ V = V^+\ \wedge\ u = 0\ \wedge\ u^+ = 1 \\ \vee\ \rho_\tau(V, V^+)\ \wedge\ u = 1\ \wedge\ u^+ = 0 \end{pmatrix}$$

This axiom claims the existence of an integer variable u, which is initially 0, and such that u changes from 1 to 0, whenever transition τ is taken. The only other change allowed to u is that it may rise from 0 to 1, provided τ is currently enabled, and no system variable is changed in the same step.

Obviously, before τ can be taken it must be enabled and this gives u a chance to rise from 0 to 1, if it is not already 1. The consequences of these requirements is that u must change at least twice for any single activation of τ, and therefore the change rate of u is at least double than the activation frequency of τ. This ability to interject more rapid changes between any existing changes is a direct consequence of the stretchability of our models.

Consider, for example, a transition system, whose transitions include τ_I, the idling transition, and transition τ_1 with transition relation $\rho_{\tau_1} : x' = x + 1$. The following model is a computation of this transition system, where we list the interpretation of variable x.

$$\sigma_1 : \quad \langle x : 0\rangle,\ \langle x : 0\rangle,\ \langle x : 1\rangle, \langle x : 1\rangle,\ \langle x : 2\rangle,\ \langle x : 2\rangle,\ \cdots$$

To demonstrate axiom RATE-DOUBLING for transition ρ_1, we have to show a u-variant of a stretched version of σ_1, which satisfies the body of the axiom. Such a u-variant is given by the following model:

$$\begin{aligned} \sigma_2 : &\langle x : 0\,,\ u : 0\rangle,\ \langle x : 0\,,\ u : 0\rangle,\ \langle x : 0\,,\ u : 1\rangle,\ \langle x : 0\,,\ u : 1\rangle, \\ &\langle x : 1\,,\ u : 0\rangle,\ \langle x : 1\,,\ u : 0\rangle,\ \langle x : 1\,,\ u : 1\rangle,\ \langle x : 1\,,\ u : 1\rangle, \\ &\langle x : 2\,,\ u : 0\rangle,\ \langle x : 2\,,\ u : 0\rangle,\ \langle x : 2\,,\ u : 1\rangle,\ \langle x : 2\,,\ u : 1\rangle, \\ &\ldots \end{aligned}$$

Obviously, we can now find a variable z which changes at least twice as fast as u, and so on.

We refer to the body of the RATE-DOUBLING axiom as $clock_\tau(u)$.

Simulation and Refinement

The notions of simulation and refinements are the same as before. Module M_1 *simulates* module M_2 if every computation of M_1 is a computation of M_2. We say that M_1 refines M_2 if every computation of M_1 is an observation of M_2. Modules M_1 and M_2 are defined to be *equivalent* if each of them refines the other.

In Figure 19, we duplicate the two programs P_2 and P_4 that demonstrated the deficiency of regular temporal logic in handling stuttering robustness.
Our previous analysis of these two programs in section 5 established that program P_2 refines program P_4. Considering the other direction, we came up with a computation σ_5 of P_4, which is definitely not an observation of P_2 according

out y : **integer where** $y = 0$	**out** y : **integer where** $y = 0$
$\begin{bmatrix} \ell_0\text{: } \textbf{loop forever do} \\ \begin{bmatrix} \ell_1\text{: } \textbf{skip} \\ \ell_2\text{: } y := y + 2 \end{bmatrix} \end{bmatrix}$	$\begin{bmatrix} L_0\text{: } \textbf{loop forever do} \\ L_2\text{: } y := y + 2 \end{bmatrix}$
– Program P_2 –	– Program P_4 –

Fig. 19. Problematic programs P_2 and P_4.

to the definition of section 5. Reworking this example in the TLR framework, we come up with the following TLR-computation of P_4.

$$\begin{aligned} \sigma_6 : & \langle \Pi : 0 \,,\, y : 0 \rangle,\ \langle \Pi : 0 \,,\, y : 0 \rangle,\ \langle \Pi : 2 \,,\, y : 0 \rangle,\ \langle \Pi : 2 \,,\, y : 0 \rangle, \\ & \langle \Pi : 0 \,,\, y : 2 \rangle,\ \langle \Pi : 0 \,,\, y : 2 \rangle,\ \langle \Pi : 2 \,,\, y : 2 \rangle,\ \langle \Pi : 2 \,,\, y : 2 \rangle, \\ & \langle \Pi : 0 \,,\, y : 4 \rangle,\ \langle \Pi : 0 \,,\, y : 4 \rangle,\ \cdots \end{aligned}$$

In the TLR framework, σ_6 is an observation of program P_2. To see this, we observe that σ_6 is a stuttering variant of the following model.

$$\begin{aligned} \sigma_7 : & \langle \Pi : 0 \,,\, y : 0 \rangle^2,\ \langle \Pi : 0 \,,\, y : 0 \rangle^2,\ \langle \Pi : 2 \,,\, y : 0 \rangle^2, \\ & \langle \Pi : 0 \,,\, y : 2 \rangle^2,\ \langle \Pi : 0 \,,\, y : 2 \rangle^2,\ \langle \Pi : 2 \,,\, y : 2 \rangle^2, \\ & \langle \Pi : 0 \,,\, y : 4 \rangle^2,\ \langle \Pi : 0 \,,\, y : 4 \rangle^2,\ \cdots \end{aligned}$$

The superscript ^{2} following a state means that this state appears twice in succession in the sequence.

Model σ_7 is an L-variant (i.e., a π-variant) of a computation of P_2. To see this, we add to its states the interpretation given to π, the control variable of program P_2.

$$\begin{aligned} \sigma_8 : & \langle \pi : 0 \,,\, \Pi : 0 \,,\, y : 0 \rangle^2,\ \langle \pi : 1 \,,\, \Pi : 0 \,,\, y : 0 \rangle^2,\ \langle \pi : 2 \,,\, \Pi : 2 \,,\, y : 0 \rangle^2, \\ & \langle \pi : 0 \,,\, \Pi : 0 \,,\, y : 2 \rangle^2,\ \langle \pi : 1 \,,\, \Pi : 0 \,,\, y : 2 \rangle^2,\ \langle \pi : 2 \,,\, \Pi : 2 \,,\, y : 2 \rangle^2, \\ & \langle \pi : 0 \,,\, \Pi : 0 \,,\, y : 4 \rangle^2,\ \langle \pi : 1 \,,\, \Pi : 0 \,,\, y : 4 \rangle^2,\ \langle \pi : 2 \,,\, \Pi : 2 \,,\, y : 4 \rangle^2,\ \cdots \end{aligned}$$

It can be shown and will actually be proven, that every computation of P_4 is an observation of P_2. It follows that P_2 and P_4 refine each other and are, therefore, equivalent.

It only remains to present a set of TLR verification rules for proving properties and refinements of programs.

10.2 Property Verification in TLR

If we restrict our attention to simple invariance, response, and reactivity formulas that only refer to rigid and left-continuous variables, it can be shown that the rules proposed for verification of regular temporal properties are still sound and complete for proving TLR properties. This is stated by the following claim. In this claim, validity refers to TLR validity.

Claim 8 *Let p, q, and r be left-continuous assertions. Then,*

a. *Rule* INV *is sound and complete for proving the P-validity of the invariance formula* $\square\, p$.
b. *Rule* RESP *is sound and complete for proving the P-validity of the response formula* $p \Rightarrow \Diamond\, r$.
c. *Rule* REACT *is sound and complete for proving the P-validity of the reactivity formula* $p \wedge \square \Diamond\, q \;\Rightarrow\; \Diamond\, r$.

It follows that any proof of these properties, carried out within the framework of regular temporal logic, can be immediately transported to the TLR framework. This is not surprising, since it can be shown that all the formulas considered in claim 8 are insensitive to stuttering. Therefore, they are valid over a program P iff they are TLR-valid over P.

10.3 Simulation and Refinement Under TLR

Next, we consider methods for verifying simulation and refinement between programs under the TLR framework. Again, the key to comparison between program are the two temporal semantics formulas, $comp_M$ and $obs_M : (\exists L) comp_M$, intended to characterize the set of computations and set of observations of module M, respectively. The same two formulas apply in the TLR case, and they precisely specify the sets of computations and observations of module M.

An Extended Verification Rule for Refinement

Rules REF and its simpler version S-REF are sound for proving refinement also within the TLR framework. However, they are not always sufficient. In particular, they cannot be used to prove refinements that are based on stuttering transformations. We therefore introduce a more general rule that can be used in these cases.

Assume we wish to prove that module M refines module $M^{\mathcal{A}}$. Obviously, this is the case if the following implication is valid:

$$comp_M \quad \rightarrow \quad obs_{M^{\mathcal{A}}}$$

Let τ_h be one of the transitions in M and u a left-continuous variable which is not a system variable of either M or $M^{\mathcal{A}}$. Due to axiom RATE-DOUBLING, it is sufficient to prove the following implication, in order to prove that M refines $M^{\mathcal{A}}$.

$$clock_{\tau_h}(u) \;\wedge\; comp_M \quad \rightarrow \quad obs_{M^{\mathcal{A}}}$$

The question is how to use the conjunct $clock_{\tau_h}(u)$ appearing in the antecedent.

Assume that module M is associated with the components $\langle V, L, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C}\rangle$. We define a new transition module $\widehat{M} : \langle \widehat{V}, \widehat{L}, \widehat{\Theta}, \widehat{\mathcal{T}}, \widehat{\mathcal{J}}, \widehat{\mathcal{C}}\rangle$, to which we refer as the *augmented module*.

- $\widehat{V} = V \cup \{u\}$. Thus, u is added to the set of system variables.
- $\widehat{L} = L \cup \{u\}$. Variable u is considered a local variable of $\widehat{M}$.

- $\widehat{\Theta} = \Theta \ \wedge \ u = 0$. The initial value of u is 0.
- $\widehat{\mathcal{T}}$ includes a transition $\widehat{\tau}$ for every $\tau \in \mathcal{T}$. For each $\tau \neq \tau_h$, the transition relation $\rho_{\widehat{\tau}}$ is defined as:
  $$\rho_{\widehat{\tau}}: \quad \rho_\tau \ \wedge \ u = u'$$
  The transition relation for $\widehat{\tau_h}$ is given by:
  $$\rho_{\widehat{\tau_h}}: \quad \rho_\tau \ \wedge \ u = 1 \ \wedge \ u' = 0.$$
  In addition, $\widehat{\mathcal{T}}$ contains an extra transition *reset*, whose transition relation is given by:
  $$\rho_{reset}: \quad En(\tau_h) \ \wedge \ V = V' \ \wedge \ u = 0 \ \wedge \ u' = 1.$$
- $\widehat{\mathcal{J}} = \{\widehat{\tau} \mid \tau \in \mathcal{J}\}$. Thus, $\widehat{\mathcal{J}}$ is the set of all transitions $\widehat{\tau}$ corresponding to a just transition $\tau \in \mathcal{J}$.
- $\widehat{\mathcal{C}} = \{\widehat{\tau} \mid \tau \in \mathcal{C}\} \cup \{reset\}$. Thus, $\widehat{\mathcal{C}}$ consists of the set of all transitions $\widehat{\tau}$ corresponding to a compassionate transition $\tau \in \mathcal{J}$, plus the new transition *reset*.

The significance of the augmented fair transition module $\widehat{M}$ is made apparent by the following claim.

Claim 9
a. A model σ satisfies the conjunction $clock_{\tau_h}(u) \wedge comp_M$ iff σ is a computation of $\widehat{M}$.
b. Module M refines module $M^{\mathcal{A}}$ iff $\widehat{M}$ refines $M^{\mathcal{A}}$.

Claim 9 provides a new method for proving refinement. To prove that M refines $M^{\mathcal{A}}$, one may select some transition τ_h of M, construct the augmented module $\widehat{M}$, and prove that $\widehat{M}$ refines $M^{\mathcal{A}}$ using rules REF or S-REF.

Example Let us illustrate the application of this method for proving that program P_4 refines program P_2. As the key transition to be clocked, we take L_2. This leads to the following augmented transition module $\widehat{M_4}$.

- $\widehat{V}: \ \{\Pi, y, u\}$.
- $\widehat{L}: \ \{\Pi, u\}$.
- $\widehat{\Theta}: \ \Pi = 0 \ \wedge \ y = 0 \ \wedge \ u = 0$.
- $\widehat{\mathcal{T}}: \ \{\widehat{\tau}_I, \widehat{L}_0, \widehat{L}_2, reset\}$, with transition relations:
  $$\begin{array}{ll} \widehat{\rho}_I: & \Pi' = \Pi \ \wedge \ y' = y \ \wedge \ u' = u \\ \widehat{\rho}_0: & \Pi = 0 \ \wedge \ \Pi' = 2 \ \wedge \ y' = y \ \wedge \ u' = u \\ \widehat{\rho}_2: & \Pi = 2 \ \wedge \ \Pi' = 0 \ \wedge \ y' = y + 2 \ \wedge \ u = 1 \ \wedge \ u' = 0 \\ \rho_{reset}: & \Pi = 2 \ \wedge \ \Pi' = 2 \ \wedge \ y' = y \ \wedge \ u = 0 \ \wedge \ u' = 1 \end{array}$$
- $\widehat{\mathcal{J}}: \ \{\widehat{L}_0, \widehat{L}_2\}$.
- $\widehat{\mathcal{C}}: \ \{reset\}$.

To show that augmented module $\widehat{M_4}$ refines program P_2, we use a refinement substitution α given by
$$\pi[\alpha]: \quad \textbf{if } \Pi = 0 \textbf{ then } 0 \textbf{ else if } u = 0 \textbf{ then } 1 \textbf{ else } 2,$$
and the transition mapping κ given by:
$$\kappa(\widehat{\tau}_I) = \tau_I, \quad \kappa(\widehat{L}_0) = \ell_0, \quad \kappa(reset) = \ell_1, \quad \kappa(\widehat{L}_2) = \ell_2.$$

The proof is based on rule S-REF and uses the invariant

$$\varphi : \quad \Pi = 0 \ \lor \ \Pi = 2,$$

which can easily be established for program P_4, and consequently for the augmented module $\widehat{M}_4$.

It is straightforward to check that premises SF1, SF2, and EF3 hold for all relevant transitions.

This concludes the proof that program P_4 refines program P_2 in the TLR framework.

10.4 Conclusion

The paper presented an approach to verification of simulation and refinement between programs, based on two temporal logics. A major observation is that proving refinement can be done by methods that are very close to the methods developed for verifying properties of programs. This is due to the expressive ability of temporal logic to present in a single, not too complex, formula the entire set of computations of a given program.

In general, the task of proving refinement requires proving implications that involve quantified temporal logic, which can be quite challenging. However, in most cases, the quantification can be removed by finding an appropriate refinement substitution which is closely related to the known proof methods based on refinement mapping. Once the refinement substitution and transition correspondence between the two compared systems is established, the rest is just verification exercise of the standard type.

We observed that some programs produce computations that are similar, but can be precisely matched only by stretching one or the other. The behaviors of such programs differ only in the pace at which they perform certain activities. Since our view of reactive systems abstracts away precise measures of time, even when they are given in "steps" rather than in seconds, we wish to consider such programs as equivalent.

Motivated by such examples, we examined the possibility of a full fledged temporal logic that is insensitive towards stuttering. The logic TLR, presented in this paper, achieves this goal and can be used to verify the more general notion of refinement, which requires stretching of some segments of the computations to make them match. This logic deserves additional study, and can perhaps be further simplified.

We believe that the detailed proof rules developed here provide a sound base for disciplined and methodical verification of refinement between programs, and show that proving refinement can be embedded within the temporal methodology for reactive systems alongside the methods for verifying properties of programs. This extends the temporal framework and enables it to deal equally effectively with both property and actions style specifications, and provide firm formal support for reactive programs development.

Acknowledgement

We gratefully acknowledge the help of Luca de Alfaro, Nikolaj Bjørner, Eddie Chang, Michael Colon, and Hugh McGuire for their careful reading of the manuscript and thank them for many helpful suggestions.

A Dense Semantics

In this appendix, we present a dense semantics for TLR.

Recall that a state s is an interpretation of $\mathcal{V}$, assigning to each variable $x \in \mathcal{V}$ a value $s[x]$ over its domain D_x. We denote by Σ the set of all states.

Let $\sigma : \mathsf{R}^+ \mapsto \Sigma$ be a mapping, assigning to each positive real number $r \in \mathsf{R}^+$ a state $\sigma(r) \in \Sigma$. Let x be a variable in $\mathcal{V}$ ranging over domain D_x, and σ be a mapping from R^+ to Σ. Let $\sigma(r)$ denote the state assigned to $r \in \mathsf{R}^+$ by σ. The mapping σ induces a function $x_\sigma : \mathsf{R}^+ \mapsto D_x$ defined by

$$x_\sigma(r) = \sigma(r)[x],$$

i.e., the interpretation given to x by the state $\sigma(r)$.

A function x_σ is said to be *left constant* at $r \in \mathsf{R}^+$, if there exists a real number t, $0 < t < r$, such that $x_\sigma(u) = d$ for all u, $t < u < r$, and some element $d \in D_x$. The element d is called the *left limit* of x_σ at r, denoted $\lim\limits_{u \to r} x_\sigma(u)$.

A function x_σ is said to be *right constant* at $r \in \mathsf{R}^+$ if there exists a real number $t > r$, such that $x_\sigma(u) = d$ for all u, $r < u < t$, and some element $d \in D_x$. The element d is called the *right limit* of x_σ at r, denoted $\lim\limits_{r \leftarrow u} x_\sigma(u)$.

A function x_σ is said to have the *finite variability* property (be a *finitely varying* function) if it is

left and right constant at any $r > 0$.

Note that it is not required that the value of $x_\sigma(r)$ equals either its left limit or its right limit. If $x_\sigma(r)$ differs from any of its limits, we say that x_σ is *discontinuous* at r.

It can be shown that a function x_σ having the finite variability property also satisfies the requirement:

The function x_σ has only finitely many points of discontinuity in any finite interval $[a, b]$, $0 \leq a \leq b$.

This explains the name “finite variability” and is the characterization given in [BKP86] for this property.

A function x_σ is called *left continuous* [*right continuous*, respectively] if $x_\sigma(r) = \lim\limits_{u \to r} x_\sigma(u)$ [$x_\sigma(r) = \lim\limits_{r \leftarrow u} x_\sigma(u)$] for every $r > 0$. Note that a finitely varying function that is both left continuous and right continuous must be a constant function.

A (dense) TLR model is a mapping $\sigma : \mathsf{R}^+ \mapsto \Sigma$, such that, for each variable $x \in \mathcal{V}$, the function x_σ has the finite variability property.

This restriction on behaviors is intended to ensure that only finitely many transitions causing state changes are allowed within each finite time interval.

A model σ' is called an x-variant of the model σ if $\sigma'(r)$ agrees with $\sigma(r)$ on the interpretation of all variables in $\mathcal{V} - \{x\}$, for all $r \in \mathsf{R}^+$.

Given a model σ, we present inductive definitions for the value of a term t at a position $r \in \mathsf{R}^+$ in σ, denoted by $val(\sigma,\ r,\ t)$, and for the notion of a temporal formula p holding at a position $r \in \mathsf{R}^+$ in σ, denoted by $(\sigma, r) \models p$.

- For a variable $x \in \mathcal{V}$,
  $val(\sigma,\ r,\ x) \quad = \quad \sigma(r)[x]$
  That is, the value assigned to x by the state $\sigma(r)$.
- For a k-ary function f of $\mathcal{L}$ and terms $t_1, \ldots, t_k$,
  $val(\sigma,\ r,\ f(t_1, \ldots, t_k)) \quad = \quad f(val(\sigma,\ r,\ t_1) \ldots, val(\sigma,\ r,\ t_k))$.
- For a term t,
  $val(\sigma,\ r,\ t^+) \quad = \quad \lim\limits_{r \leftarrow u} val(\sigma,\ u,\ t)$
- For a term t,
  $val(\sigma,\ r,\ t^-) \quad = \quad \lim\limits_{u \rightarrow r} val(\sigma,\ u,\ t)$
- For a k-ary predicate p of $\mathcal{L}$ and terms $t_1, \ldots, t_k$,
  $(\sigma, r) \models p(t_1, \ldots, p_k) \iff p(val(\sigma,\ r,\ t_1) \ldots, val(\sigma,\ r,\ t_k)) = \mathrm{T}$
- $(\sigma, r) \models \neg p \iff (\sigma, r) \not\models p$
- $(\sigma, r) \models p \vee q \iff (\sigma, r) \models p$ or $(\sigma, r) \models q$
- $(\sigma, r) \models \bigcirc p \iff$ there exists $v > r$, such that $(\sigma, u) \models p$ for all u, $r < u < v$.
- $(\sigma, r) \models \ominus p \iff$ there exists a v, $0 < v < r$, such that $(\sigma, u) \models p$ for all u, $v < u < r$.
- $(\sigma, r) \models p \,\mathcal{U}\, q \iff$ for some $v \geq r, (\sigma, v) \models q$, and $(\sigma, u) \models p$, for all u, $r \leq u < v$.
- $(\sigma, r) \models \forall x.\ p \iff (\sigma', r) \models p$ for all σ', an x-variant of σ
- $(\sigma, r) \models \exists x.\ p \iff (\sigma', r) \models p$ for some σ', an x-variant of σ

It can be shown that the property of finite variability is upwards hereditary. That is, if the arguments of a function, a predicate, or an operator are finitely varying then so is the term or formula obtained by applying the operation, predicate, or operator to these arguments. Consequently, all the terms and formulas constructed and evaluated over a model are finitely varying. This justifies the application of the operators $\lim\limits_{u \rightarrow r}$ and $\lim\limits_{r \leftarrow u}$ to the semantic term $val(\sigma,\ u,\ t)$.

Unfortunately, the properties of left and right continuity are not hereditary, i.e., they are not preserved by all operators. Consider, for example, the interpretation of a boolean variable x, given by $x_\sigma(r):\ r > 1$, i.e., $x_\sigma(r)$ is true iff $r > 1$. This interpretation is left continuous. However, the formula $\neg x \wedge \bigcirc x$ (equivalently, $\neg x \wedge x^+$) is neither left nor right continuous at $r = 1$. This is because this formula is true precisely at $r = 1$ and false elsewhere.

The equivalence of the dense semantics defined here and the discrete semantics defined in section 7 is stated by the following claim.

Claim 10 *A* TLR *formula p is valid according to the dense semantics iff it is* *valid according to the discrete semantics.*

The proof of this theorem is based on a two ways reduction between dense models and discrete ones.

References

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. 3rd IEEE Symp. Logic in Comp. Sci.*, pages 165–175, 1988.

[BAMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Proc. 13th ACM Symp. Princ. of Prog. Lang.*, pages 173–183, 1986.

[dBdRR90] J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness.* Lecture Notes in Computer Science 430. Springer-Verlag, 1990.

[DK90] E. Diepstraten and R. Kuiper. Abadi, Lamport, and Stark: Towards a proof theory for stuttering, dense domains and refinmement mappings. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, Lecture Notes in Computer Science 430, pages 208–238. Springer-Verlag, 1990.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. Princ. of Prog. Lang.*, pages 163–173, 1980.

[Lam83a] L. Lamport. Specifying concurrent program modules. *ACM Trans. Prog. Lang. Sys.*, 5:190–222, 1983.

[Lam83b] L. Lamport. What good is temporal logic. In R.E.A. Mason, editor, *Proc. IFIP 9th World Congress*, pages 657–668. North-Holland, 1983.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, 1991. To appear in *Transactions on Programming Languages and Systems.*

[Lam92] L. Lamport. The existence of refinement mapping. TLA Note 92-03-19, March 1992.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, volume 193 of *Lect. Notes in Comp. Sci.*, pages 196–218. Springer-Verlag, 1985.

[MP83] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proc. 10th ACM Symp. Princ. of Prog. Lang.*, pages 141–154, 1983.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comp. Prog.*, 32:257–289, 1984.

[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.

[MP94] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems.* Springer-Verlag, New York, 1994. To appear.

[Pnu92] A. Pnueli. System specification and refinement in temporal logic. In R.K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lect. Notes in Comp. Sci.*, pages 1–38. Springer-Verlag, 1992.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:1–20, 81.

[Sta84] E.W. Stark. *Foundations of a Theory of Specifications for Distributed Systems*. PhD thesis, M.I.T, 1984.

[Sta88] E.W. Stark. Proving entailments between conceptual state specifications. *Theor. Comp. Sci.*, 56:135–154, 1988.