Verification of Parameterized Programs

Zohar Manna and Amir Pnueli

1 Introduction

In this chapter we present an approach to the verification of parameterized reactive programs, using temporal logic[1].

A parameterized program consists of several similar processes whose number is determined by an input parameter. A challenging problem is to provide methods for the *uniform* verification of such programs, i.e., proving correctness of the program for *any* number of processes. The ability to conduct a uniform verification of a parameterized program is one of the striking advantages of the deductive method for temporal verification over model-checking techniques such as [2] and [1].

Let M denote the input parameter which determines the number of processes in the considered system. We can use model-checking to verify the desired properties of the system for specific values of M, such as $M = 3, 4, 5$. Usually, the model checker's memory capacity is exceeded for values of M smaller than 100. Furthermore, in the general case, nothing can be concluded about the property holding for any value of M from the fact that it holds for some finite set of values. In comparison, the deductive method establishes in one fell swoop the validity of the property for any value of M.

The verification approach presented here extends the methodology introduced in [5] to the case of parameterized programs. Preliminary examples of verification of parameterized programs were presented in [4] and [7]. Verification of safety properties of parameterized programs is discussed in [9].

We start by introducing in Section 2 our general computational model for fair transition systems, the simple concurrent programming language SPL, and the specification language of temporal logic. We then introduce parameterized programs (Section 3), and show how to verify their safety properties (Section 4) and their response properties (Section 5). In Sec-

[1] This research was supported in part by the National Science Foundation under grant CCR-92-23226, by the Defense Advanced Research Projects Agency under contracts NAG2-892, by the United States Air Force Office of Scientific Research under contract F49620-93-1-0139, and by the European Community ESPRIT Basic Research Action Project 6021 (REACT).

tion 6, we present verification diagrams and illustrate their use for presenting proofs of response properties. Finally, in Section 7, we present a set of examples of parameterized programs that communicate by "add-and-store" statements.

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 finite 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 said to be *enabled* on s iff $\tau(s) \neq \emptyset$. Otherwise τ is *disabled* on 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 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.

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

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

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

The enabledness of a transition τ can be expressed by the formula

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

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

We require that every state $s \in \Sigma$ has at least one transition enabled on it. This is often ensured by including in $\mathcal{T}$ the *idling* transition τ_I (also called the *stuttering* transition), whose transition relation is $\rho_I : (V = V')$. Thus, s' is a τ_I-successor of s iff s 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. A program is constructed out of statements which may be labeled. Labels serve as unique names for the statements and also as possible sites of control.

The following is a list of some SPL statements together with an explanation of their intended meaning. We present only the statements that are used in this chapter; the reader is referred to [6] for a description of the full language.

- *Assignment*: For a list of variables $u_1, \ldots, u_k$ and a list of expressions of corresponding types $e_1, \ldots, e_k$,

  $$(u_1, \ldots, u_k) \ := \ (e_1, \ldots, e_k)$$

  is an *assignment* statement. For the case that $k = 1$, we write simply $u := e$.

- *Await*: For a boolean expression c,

  await c

  is an *await* statement. We refer to condition c as the *guard* of the statement. Execution of **await** c changes no data variables. Its sole purpose is to wait until c becomes true, at which point it terminates.
- *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.
- *Conditional*: For statement S and a boolean expression c,

  if c **then** S

  is a (one-branch) *conditional* statement. Its execution begins by evaluating c. If c evaluates to F, execution of the statement terminates. Otherwise, subsequent steps proceed to execute S.
- *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.
- *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.
- *Selection*: For statements $S_1, \ldots, S_k$,

  S_1 **or** $\cdots$ **or** S_k

  is a *selection* statement. Its intended meaning is a non-deterministic selection of a statement S_i and its execution. The first step in the execution of the selection statement selects a statement S_i, $i = 1, \ldots, k$, that is currently enabled and performs the first step in the execution of S_i. Subsequent steps proceed to execute the rest of the selected substatement, ignoring the other S_j's. If more than one of $S_1, \ldots, S_k$ is enabled, the selection is non-deterministic.

- *Cooperation*: For statements $S_1, \ldots, S_k$,

  ℓ: $[\ell_1\!: S_1;\ \widehat{\ell}_1\!:\ ] \parallel \cdots \parallel [\ell_k\!: S_k;\ \widehat{\ell}_k\!:\ ];\ \widehat{\ell}\!:$

  is a *cooperation* statement. Its intended meaning is the parallel execution of $S_1, \ldots, S_k$. The first step in the execution of a cooperation statement is referred to as the *entry* step. It can be conceived as setting the stage for the parallel execution of $S_1, \ldots, S_k$ while moving from ℓ to $\ell_1, \ldots, \ell_k$. Subsequent steps proceed to perform steps from $S_1, \ldots, S_k$. When all $S_1, \ldots, S_k$ have terminated, there is an additional *exit* step that closes the parallel execution, and moves from $\widehat{\ell}_1, \ldots, \widehat{\ell}_k$ to $\widehat{\ell}$. Each parallel component is assigned an *entry label* ℓ_i and an *exit label* $\widehat{\ell}_i$, which is the location of control after execution of S_i terminates. Label $\widehat{\ell}_i$ can be viewed as labeling an empty statement following S_i. We refer to component $[\ell_i\!: S_i;\ \widehat{\ell}_i\!:\ ]$ as a *process* of the cooperation statement, and say that statements $S_1, \ldots, S_k$ are parallel to one another.

  It is important to note that in the combination

  $\big([\ell_1\!: S_1;\ \widehat{\ell}_1\!:\ ] \parallel [\ell_2\!: S_2;\ \widehat{\ell}_2\!:\ ]\big);\ S_3,$

  execution of S_3 cannot start until both S_1 and S_2 have terminated.

  Execution of the cooperation statement, once it has been initialized, proceeds by *interleaving*. That is, transitions from the various processes $S_1, \ldots, S_k$ are nondeterministically chosen and executed, one at a time. The two fairness requirements, justice and compassion, ensure that a computation does not consistently ignore one of the processes forever.

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, where coordination between the processes is required. It is not required that this statement terminate.
- *Critical*: The statement

  critical

  represents the critical activity in programs for mutual exclusion. It is required that this statement terminates.

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

Programs

A program P has the form

$$P :: \left[\text{declaration};\ \left[P_1 :: [\ell_1\text{: } S_1;\ \ \widehat{\ell_1}\ :] \parallel \cdots \parallel P_m :: [\ell_m\text{: } S_m;\ \ \widehat{\ell_m}\ :]\right]\right],$$

where $P_1 :: [\ell_1\text{: } S_1;\ \widehat{\ell_1}\ :], \ldots, P_m :: [\ell_m\text{: } S_m;\ \widehat{\ell_m}\ :]$ are *named processes*. The names of the program and of the processes are optional, and may be omitted. The *body* $[\ell_i\text{: } S_i;\ \ \widehat{\ell_i}\ :]$ of process P_i consists of a (possibly labeled) statement $\ell_i : S_i$ 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 (Program ANY-Y) Fig. 1 presents a simple program consisting of two processes communicating by the shared variable x, initially set to 0. Process P_1 keeps incrementing variable y as long as $x = 0$. Process P_2 has only one statement, which sets x to 1. Obviously, once x is set to 1, process P_2 terminates and some time later so does P_1, as soon as it observes that $x \neq 0$. ▟

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

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

FIG. 1. Program ANY-Y: a simple concurrent program.

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

$$\Big[\text{declaration; } \big[P_1 :: [\ell_1 \colon S_1;\ \widehat{\ell}_1 :] \parallel \cdots \parallel P_m :: [\ell_m \colon S_m;\ \widehat{\ell}_m :]\big]\Big].$$

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.

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 a single *control variable* π. The data variables Y range over their respectively declared data domains. The control variable π ranges over subsets of L_P, i.e., sets of locations. The value of π in a state denotes all the locations of the program in which control currently resides.

For example, the system variables for program ANY-Y are $V : \{\pi, x, y\}$, where x and y range over the integers, and π ranges over subsets of the location set $\{\ell_0, \ell_1, \ell_2, m_0, m_1\}$.

As states we take all possible interpretations that assign to the system variables values over their respective domains. For example, the initial state of program ANY-Y is

$$\langle \pi : \{\ell_0, m_0\},\ x : 0,\ y : 0\rangle.$$

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

$$at\_\ell : \quad \pi \cap L_i = \{\ell\}.$$

Variable π satisfying at_ℓ implies that $\ell \in \pi$ and that no other location $\tilde{\ell} \in L_i$, $\tilde{\ell} \neq \ell$, satisfies $\tilde{\ell} \in \pi$. For example, for program ANY-Y, $\pi = \{\ell_0, m_0\}$ 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

$$\Theta\colon \quad \pi = \{\ell_1, \ldots, \ell_m\} \ \wedge\ \varphi.$$

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

For example, the initial condition for program ANY-Y is given by

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

Transitions

To facilitate the expression of transition relations, we introduce several abbreviations. Let L and $\widehat{L}$ be two sets of locations. The abbreviation

$$move(L, \widehat{L})\colon \quad \bigwedge_{[\ell] \in L} at\_\ell \ \wedge\ \pi' = (\pi - L) \cup \widehat{L}$$

expresses a simultaneous move of control from all the locations in L to the new locations in $\widehat{L}$. For the simple case that $L = \{\ell\}$ and $\widehat{L} = \{\widehat{\ell}\}$, we write $move(\{\ell\}, \{\widehat{\ell}\})$ simply as $move(\ell, \widehat{\ell})$.

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

$$\rho_I: \ pres(V).$$

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

- *Assignment*: Consider the statement

  $$\ell\colon\ (u_1, \ldots, u_k) := (e_1, \ldots, e_k); \quad \widehat{\ell}:$$

  where $\widehat{\ell}$ is the location to which control moves after the 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

  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \wedge y_1' = e_1 \wedge \cdots \wedge y_k' = e_k \wedge pres(Y - \{y_1, \ldots, y_k\}).$$

  The last conjunct claims that all data variables Y, excluding $y_1, \ldots, y_k$, retain their values over the transition τ_ℓ.

- *Await*: With the statement ℓ: **await** c; $\widehat{\ell}$:, we associate a transition τ_ℓ, with the transition relation

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

  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 $\widehat{\ell}$.

- *Request*: With the statement ℓ: **request**(r); $\widehat{\ell}$: , we associate a transition τ_ℓ, with the transition relation

  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ r > 0 \ \wedge \ r' = r - 1 \ \wedge \ pres(Y - \{r\}).$$

  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 ℓ: **release**(r); $\widehat{\ell}$: , 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.

- *Conditional*: With the statement ℓ: [**if** c **then** $\widetilde{\ell}: S$] ; $\widehat{\ell}$:, we associate a transition τ_ℓ, with the transition relation

  $$\rho_\ell: \quad \begin{pmatrix} c \ \wedge \ move(\ell, \widetilde{\ell}) \\ \vee \\ \neg c \ \wedge \ move(\ell, \widehat{\ell}) \end{pmatrix} \ \wedge \ pres(Y).$$

  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 $\widehat{\ell}$.

- *While*: With the statement ℓ: $\left[\textbf{while}\ c\ \textbf{do}\ [\widetilde{\ell}\colon \widetilde{S}]\right]$; $\widehat{\ell}$:, we associate a transition τ_ℓ, with the transition relation
$$\rho_\ell\colon\ \left(\begin{array}{c} c\ \wedge\ move(\ell,\widetilde{\ell}) \\ \vee \\ \neg c\ \wedge\ move(\ell,\widehat{\ell}) \end{array}\right)\ \wedge\ pres(Y).$$
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.
- *Cooperation* Excluding the body of the program, we associate with each *cooperation* statement,
$$\ell\colon\ \left[[\ell_1\colon S_1;\ \widehat{\ell}_1]\ \|\ \cdots\ \|\ [\ell_k\colon S_k;\ \widehat{\ell}_k]\right];\quad \widehat{\ell}\colon$$
an *entry transition* τ_ℓ^{E} and an *exit transition* τ_ℓ^{X}. The corresponding transition relations are given, respectively, by
$$\rho_\ell^{\mathrm{E}}\colon\quad move\big(\{\ell\},\ \{\ell_1,\ldots,\ell_k\}\big)\ \wedge\ pres(Y)$$
$$\rho_\ell^{\mathrm{X}}\colon\quad move\big(\{\widehat{\ell}_1,\ldots,\widehat{\ell}_k\},\ \{\widehat{\ell}\}\big)\ \wedge\ pres(Y).$$
The entry transition begins execution of the cooperation statement by placing in π the set of locations that are just in front of the parallel statements $S_1,\ldots,S_k$. The exit transition can be taken only if all the parallel statements have terminated, which is detectable by observing that π satisfies
$$at\_\widehat{\ell}_1\ \wedge\ \cdots\ \wedge\ at\_\widehat{\ell}_k.$$
Note that the absence of entry or exit transitions associated with the body of the program is consistent with the fact that, according to the initial condition, execution starts with π set to the entry locations of the top-level processes.
- *Noncritical*: With the statement ℓ: **noncritical**; $\widehat{\ell}$:, we associate a transition τ_ℓ, with transition relation
$$\rho_\ell\colon\quad move(\ell,\widehat{\ell})\ \wedge\ pres(Y).$$
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
$$\rho_\ell\colon\quad move(\ell,\widehat{\ell})\ \wedge\ pres(Y).$$
The only observable action of the critical statement is to terminate.

The discerning reader may have observed that no transitions are directly associated with the concatenation statement. This is because each transition occurring in the execution of a concatenation statement can be attributed to one of its substatements.

When there is no danger of confusion, we will refer to the transition τ_ℓ simply as ℓ.

For example, the fair transition system corresponding to program ANY-Y has the transitions

$$\tau_I,\ \tau_{\ell_0},\ \tau_{\ell_1},\ \tau_{m_0},$$

which can also be presented as $\tau_I, \ell_0, \ell_1, m_0$. Transition τ_{ℓ_1}, for example, has the transition relation

$$\rho_{\ell_1}: \quad move(\ell_1, \ell_0) \ \wedge\ y' = y + 1 \ \wedge\ x' = x.$$

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 chapter. The programs presented in [6] 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 Requirement Specification Language: Temporal Logic

As a requirement specification language for reactive systems 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), and temporal operators. There are two classes of temporal operators, future and past; in this presentation we refer only to the future operators. The basic future temporal operators are:

$$\bigcirc \text{ – Next} \qquad\qquad \mathcal{U} \text{ – Until}$$

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state s_j provides an interpretation for the vocabulary of p, i.e., the variables occurring in p.

For a given state s_j and 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.
- $(\sigma, j) \vDash \neg p \iff (\sigma, j) \nvDash p$
- $(\sigma, j) \vDash p \vee q \iff (\sigma, j) \vDash p$ 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 $j \leq i < k$

Additional temporal operators can be defined as follows:

$\diamondsuit\, p = \text{T}\,\mathcal{U}\, p$ – Eventually
$\square\, p = \neg \diamondsuit \neg p$ – Henceforth
$p\,\mathcal{W}\, q = \square\, p \vee (p\,\mathcal{U}\, q)$ – Wait-for (Unless, Weak Until)

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

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

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 $(\sigma, j) \vDash p$, we say that j is a *p-position* (in σ).

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

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.

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

The formulas p and q are defined to be *congruent*, denoted $p \approx q$, if the formula $\square(p \leftrightarrow q)$ is valid, i.e., $(\sigma, j) \vDash p$ iff $(\sigma, j) \vDash q$, 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 Specification 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 two most important classes: *safety* and *response*.

Safety Properties

Safety properties are those that can be expressed by a formula generated from state formulas, the boolean operators $\vee$ and $\wedge$, and the temporal operators $\square$ and $\mathcal{W}$. We refer to a formula of this form as a *canonical safety formula*.

The only safety properties we consider in this presentation are those that can be expressed by the *invariance* formula $\square\,\varphi$, where φ is a state formula.

The following are several safety formulas that are valid over program ANY-Y and therefore specify properties of this program:

- $\square(x = 0 \;\vee\; x = 1)$
  This formula claims that x has the values 0 or 1 in all states appearing in computations of ANY-Y.
- $\square(x = 1 \leftrightarrow at\_m_1)$
  This formula claims that x equals 1 if and only if control of process P_2 is at location m_1.
- $at\_\ell_1 \;\Rightarrow\; at\_m_1$
  This formula claims that P_1 can be at location ℓ_1 only if control of process P_2 is at location m_1.

Response Properties

We also consider response properties that can be expressed by a formula

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

for state formulas p and q.

For example, the response formula

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

states the property of *termination* of program ANY-Y. This property ensures that the program eventually reaches a state in which P_1 is at ℓ_2 while P_2 is at m_1.

Verification Conditions

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

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

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

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

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

In the context of program ANY-Y, consider for example the verification condition of transition ℓ_1 with respect to assertions $p = q : \; y \geq 0$.

$$\{y \geq 0\}\; \ell_1 \; \{y \geq 0\}.$$

Expanding the definition of the verification condition, this yields

$$\underbrace{\cdots \wedge\; y' = y+1 \;\wedge \cdots}_{\rho_{\ell_1}} \quad \wedge \quad \underbrace{y \geq 0}_{p} \quad \rightarrow \quad \underbrace{y' \geq 0}_{q'}.$$

which is state valid. This shows that every ℓ_1-successor of a state satisfying $y \geq 0$ also satisfies $y \geq 0$.

3 Parameterized Programs

Many distributed programs, in particular those that control communication and synchronization of networks, have as their body a parallel composition of many identical processes.

For example, we may consider the simple case of many processes coordinating mutual exclusion by a single semaphore. In this case, the body S of each process may have the form

$$\ell_0\text{: } \textbf{loop forever do} \begin{bmatrix} \ell_1\text{: } \textbf{noncritical} \\ \ell_2\text{: } \textbf{request } y \\ \ell_3\text{: } \textbf{critical} \\ \ell_4\text{: } \textbf{release } y \end{bmatrix}.$$

The complete program, for the special case of three processes, is given by

$$
\begin{array}{l}
\textbf{in} \quad M\colon \textbf{integer where } M \geq 1 \\
\qquad\; x \;:\textbf{array } [1..M] \textbf{ of integer} \\
\textbf{out } z \;:\textbf{integer where } z = 0 \\
\displaystyle\mathop{\Big\|}_{j=1}^{M} \; P[j] :: \left[\begin{array}{l} \textbf{local } y\colon \textbf{ integer} \\ \ell_0\colon\; y := x[j] \\ \ell_1\colon\; z := z + y \cdot y \\ \ell_2\colon \end{array}\right]
\end{array}
$$

FIG. 2. Program PAR-SUM (parallel sum of squares).

$$P^3 :: \quad \Big[\textbf{local } y\colon \textbf{ integer where } y = 1;\; \big[S \,\|\, S \,\|\, S\big]\Big].$$

Some renaming of locations will be needed to distinguish between the three copies of S.

However, establishing the correctness of this three-process program, i.e., showing that mutual exclusion is maintained, does not formally guarantee anything about the following similar program, which has four copies of the statement S as its processes:

$$P^4 :: \quad \Big[\textbf{local } y\colon \textbf{ integer where } y = 1;\; \big[S \,\|\, S \,\|\, S \,\|\, S\big]\Big].$$

In some cases, it is possible to treat the whole (infinite) family of programs $P^1, P^2, P^3, P^4, \ldots$ in a *uniform* way, providing a single proof that guarantees correctness of P^n for any $n \geq 1$. The key to such a uniform treatment is *parameterization*, i.e., presenting a single syntactic object that actually represents a family of objects. There are three areas that we have to parameterize: the programs, their specifications, and the corresponding verification process.

3.1 Parameterizing the Syntax of Programs

Programs may be parameterized by using compound statements of variable size. Here we only consider a variable-size *cooperation* statement, which has the following form:

$$\mathop{\Big\|}_{j=1}^{M} S[j].$$

The *parameterized* statement $S[j]$ may contain explicit occurrences of the parameter j as a variable in expressions and as a subscript in array references. In addition, the parameter j is considered to implicitly subscript all the variables that are declared local to $S[j]$ and all the labels occurring within $S[j]$.

Example Consider program PAR-SUM of Fig. 2 that computes in z the sum of the squares of the elements of the array $x[1..M]$.

$$
\begin{array}{l}
\textbf{in} \quad M\colon \textbf{integer where } M \geq 1 \\
\qquad\; x\ : \textbf{array } [1..M] \textbf{ of integer} \\
\textbf{out}\ z\ : \textbf{integer where } z = 0
\end{array}
$$

$$
\mathop{\Big\|}_{j=1}^{M} P[j] :: \left[\begin{array}{l}
\textbf{local } y[j]\colon \textbf{ integer} \\
\ell_0[j]\colon\ y[j] := x[j] \\
\ell_1[j]\colon\ z := z + y[j] \cdot y[j] \\
\ell_2[j]\colon
\end{array}\right]
$$

FIG. 3. Program PAR-SUM — explicitly subscripted version.

The body of this program is a variable-size *cooperation* statement. The cooperation statement consists of a parallel composition of processes $P[1], \ldots, P[M]$. As we see in this program, the parameter j appears explicitly in the reference to $x[j]$. It is also assumed to implicitly subscript all the labels appearing in $P[j]$, as well as the variable y which is locally declared in $P[j]$. In Fig. 3 we present the same program with the subscripting by j made explicit.

Usually, we prefer to present program PAR-SUM as in Fig. 2, but to reason about its behavior in terms of the fully subscripted version as in Fig. 3. ▟

3.2 Parameterized Transition Systems

Several extensions are necessary for the convenient representation of parameterized programs as transition systems. The main extension is the need to replace the notion of *finiteness* by that of *finite representability*.

For example, all the transition systems we have considered so far had a fixed number of transitions. This is no longer true in the case of parameterized programs. Consider, for example, statement ℓ_0 in the program of Fig. 2. This statement copies the value of $x[j]$ to the variable y, which is local to $P[j]$. Associating transitions with statements, we must have a separate transition $\tau_0[j]$ for each $j = 1, \ldots, M$. Since M is not fixed, there are unboundedly many transitions associated with the statement ℓ_0. On the other hand, we can represent all these transitions by a single *transition relation scheme* that uses j as a parameter. This transition scheme is given by

$$\rho_0[j]\colon \quad \mathit{move}\big(\ell_0[j],\ \ell_1[j]\big)\ \wedge\ y'[j] = x[j].$$

Consider a fully subscripted program, such as the one presented in Fig. 3; it is quite straightforward to define the transition system associated with it. Let us point out the main constituents of this transition system.

- *System Variables*

  These consist of

$\pi \quad M \quad x \quad z \quad y.$

The variable π ranges over sets of locations, some of which may be subscripted, e.g., of the form $\ell_1[j]$ for various j's. Arrays, whether explicit (e.g., x) or implicit (e.g., y in Fig. 2), are treated as variables that range over functions. Thus, x and y are considered as functions from the (subscript) domain $[1..M]$ to the integers.

- *Initial Condition*

  $\Theta\colon \quad \pi = \{\ell_0[1], \ldots, \ell_0[M]\} \ \wedge \ M \geq 1 \ \wedge \ z = 0.$
- *Transitions*

  The transitions of the program consist of

  $\ell_0[j]$: place $x[j]$ in $y[j]$.

  $\ell_1[j]$: add $y[j] \cdot y[j]$ to z.

We present below a computation of program PAR-SUM for the case that $M = 2$ and $x = \langle 1, 2\rangle$. For each state, we list the current values of π, z, and y.

$$\big\langle \{\ell_0[1],\ \ell_0[2]\},\ 0,\ \langle -,\ -\rangle\big\rangle \xrightarrow{\ell_0[2]}$$
$$\big\langle \{\ell_0[1],\ \ell_1[2]\},\ 0,\ \langle -,\ 2\rangle\big\rangle \xrightarrow{\ell_1[2]} \big\langle \{\ell_0[1],\ \ell_2[2]\},\ 4,\ \langle -,\ 2\rangle\big\rangle \xrightarrow{\ell_0[1]}$$
$$\big\langle \{\ell_1[1],\ \ell_2[2]\},\ 4,\ \langle 1,\ 2\rangle\big\rangle \xrightarrow{\ell_1[1]} \big\langle \{\ell_2[1],\ \ell_2[2]\},\ 5,\ \langle 1,\ 2\rangle\big\rangle$$

To retrieve the value of the kth element of any array y, we use the notation $y[k]$ that can be viewed as applying the function y to the argument k. To represent the modification of an array x, we use the notation

$$update(x,\ k,\ e).$$

The value of this expression is an array that agrees with x on all subscript values $i \neq k$, and has as its kth element the value of e.

The main properties of the *update* function are

$$update(x,\ k,\ e)[k] \ = \ e$$
$$j \neq k \ \rightarrow \ update(x,\ k,\ e)[j] = x[j].$$

Thus, the proper representation of the transition relation for $\ell_0[j]$ is

$$\rho_0[j]\colon \quad move\big(\ell_0[j],\ \ell_1[j]\big) \ \wedge \ y' = update\big(y,\ j,\ x[j]\big).$$

When there is no danger of confusion, we will abbreviate equalities of the form $y' = update(y, k, e)$, to $y'[k] := e$. Note that even if the transition modifies k, the value to be used is that of k rather than k'.

3.3 Examples

We present two examples that illustrate the structure of parameterized programs. In later discussions, we will specify and verify these two examples.

Example (multiple mutual exclusion by semaphores)

To illustrate the succinct way in which a parameterized program represents an infinite family of programs, consider the parameterized program MPX-SEM (Fig. 4) that achieves mutual exclusion by semaphores.

In this program we use incrementation modulo M, defined by

$$
\begin{array}{l}
\textbf{in} \quad M: \textbf{integer where } M \geq 2 \\
\textbf{local } y \ : \textbf{array } [1..M] \textbf{ of integer} \\
\qquad\qquad\qquad \textbf{where } y[1] = 1,\ y[j] = 0 \text{ for } 2 \leq j \leq M \\
\\
\overset{M}{\underset{j=1}{\Big|\Big|}} \ P[j] :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\qquad \left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } \textbf{request } y[j] \\
\ell_3\text{: } \textbf{critical} \\
\ell_4\text{: } \textbf{release } y[j \oplus_M 1]
\end{array}\right]
\end{array}\right]
\end{array}
$$

FIG. 4. Program MPX-SEM (multiple mutual exclusion by semaphores).

$$
\begin{array}{l}
\textbf{local } y\text{: } \textbf{array } [1..2] \textbf{ of integer where } y[1] = 1,\ y[2] = 0 \\
\\
P[1] :: \left[\begin{array}{l}
\ell_0[1]\text{: } \textbf{loop forever do} \\
\qquad \left[\begin{array}{l}
\ell_1[1]\text{: } \textbf{noncritical} \\
\ell_2[1]\text{: } \textbf{request } y[1] \\
\ell_3[1]\text{: } \textbf{critical} \\
\ell_4[1]\text{: } \textbf{release } y[2]
\end{array}\right]
\end{array}\right] \\
\qquad || \\
P[2] :: \left[\begin{array}{l}
\ell_0[2]\text{: } \textbf{loop forever do} \\
\qquad \left[\begin{array}{l}
\ell_1[2]\text{: } \textbf{noncritical} \\
\ell_2[2]\text{: } \textbf{request } y[2] \\
\ell_3[2]\text{: } \textbf{critical} \\
\ell_4[2]\text{: } \textbf{release } y[1]
\end{array}\right]
\end{array}\right]
\end{array}
$$

FIG. 5. Program MPX-SEM — elaboration for $M = 2$.

$$
j \oplus_M 1 \ = \ (j \ mod \ M) + 1 \ = \ \begin{cases} j+1 & \text{if} \quad j < M \\ 1 & \text{if} \quad j = M \end{cases}
$$

In Fig. 5 we present one member of the family, corresponding to $M = 2$, in which we explicitly represent the dependence of the locations on the process' subscript. ▪

Example (finding the maximum of an array)

As another illustrative example, consider program MAX-ARRAY given in Fig. 6. This program places in the output variable z the maximum ele-

$$
\begin{array}{l}
\quad\quad\quad \textbf{in} \quad\ M\colon \textbf{integer where } M \geq 1 \\
\quad\quad\quad \phantom{\textbf{local}}\ x\ : \textbf{array } [1..M] \textbf{ of integer} \\
\quad\quad\quad \textbf{local } y\ : \textbf{array } [1..M] \textbf{ of boolean where } y = \mathrm{T} \\
\quad\quad\quad \textbf{out} \quad z\ : \textbf{integer} \\
\ell_0\colon \mathop{\Big|\Big|}_{i,j \in [1..M]} P[i,j] :: \begin{bmatrix} \ell_1\colon \textbf{ if } x[i] < x[j] \textbf{ then } \ell_2\colon\ y[i] := \mathrm{F} \\ \ell_3\colon \end{bmatrix} \\
\ell_4\colon \mathop{\Big|\Big|}_{i=1}^{M} Q[i] :: \begin{bmatrix} \ell_5\colon \textbf{ if } y[i] \textbf{ then } \ell_6\colon\ z := x[i] \\ \ell_7\colon \end{bmatrix} \\
\ell_8\colon
\end{array}
$$

FIG. 6. Program MAX-ARRAY (finding maximum of an array).

ment of the integer array x. A unique feature of this program is that the maximum value is computed in a highly parallel fashion.

The initial value of array y is specified to be $y[j] = \mathrm{T}$ for all $j = 1, \ldots, M$.

The first statement spawns into M^2 processes. Each process $P[i,j]$ sets the boolean variable $y[i]$ to F if the array element $x[i]$ is smaller than the element $x[j]$. After all the processes in the cooperation statement ℓ_0 have terminated, we are guaranteed that, for every i such that $x[i]$ is not maximal in x, $y[i] = \mathrm{F}$. It follows that $y[i] = \mathrm{T}$ iff $x[i]$ is maximal.

The parallel processes spawned in ℓ_4 set z to $x[i]$ if $x[i]$ is maximal. Note that, since there may be more than one i such that $x[i]$ is maximal, more than one process may assign a value to z, but all the assigned values are the same.

The statement comprising the body of the MAX-ARRAY program may be inserted anywhere in a larger program, where we have an array $x[1..M]$ and need to efficiently compute its maximum[2]. The variable M may assume different values in different entries to this statement. On each entry, the appropriate number of processes is spawned. ∎

3.4 Specifying Parameterized Programs

To specify and reason about parameterized programs, we introduce some conventions and special notation.

Subscripted Locations

An important requirement for the specification of parameterized programs is the ability to refer to parameterized locations. We use integer variables

[2]Program MAX-ARRAY is very efficient in terms of time, but may be considered wasteful in terms of space requirements.

to subscript locations, and allow the general integer operations, predicates, and quantifiers to be applied to these variables.

Thus, to specify the property of mutual exclusion for program MPX-SEM (Fig. 4) for an arbitrary M, $M \geq 2$, we may use the formula

$$\forall i, j \in [1..M]\colon\ i \neq j\colon\quad \Box\, \underbrace{\neg\big(at\_\ell_3[i]\ \wedge\ at\_\ell_3[j]\big)}_{\psi[i,j]}\ .$$

This formula states that for every two distinct processes $P[i]$ and $P[j]$, $i, j \in [1..M]$, $i \neq j$, it is never the case that control is at $\ell_3[i]$ and $\ell_3[j]$ at the same time.

Notations

Several notational conventions are useful for reasoning about parameterized programs. For a location ℓ_i in a parameterized process we define the set L_i by

$$L_i\ =\ \big\{j \mid \ell_i[j] \in \pi\big\},$$

that is, the set of indices of the processes (a subset of $\{1..M\}$) that currently reside at location ℓ_i. We define a corresponding integer

$$N_i\ =\ |L_i|,$$

that represents the *number* of processes currently residing at location ℓ_i.

This notation is easily extended to refer to *location sets* rather than to individual locations. Thus,

$$\begin{aligned}
L_{i_1,i_2,\ldots,i_k} &= L_{i_1}\ \cup\ L_{i_2}\ \cup\ \ldots\ \cup\ L_{i_k}\\
L_{i..j} &= L_i\ \cup\ L_{i+1}\ \cup\ \ldots\ \cup\ L_j\\
N_{i_1,i_2,\ldots,i_k} &= |L_{i_1,i_2,\ldots,i_k}|\\
N_{i..j} &= |L_{i..j}|.
\end{aligned}$$

4 Verifying Invariance Properties

Now that we have discussed the transition semantics of parameterized programs and introduced a sufficiently expressive assertion language, we can proceed with the verification of properties of parameterized programs, starting with *state-invariance* properties.

4.1 A Rule for State Invariance

For an assertion (state formula) p, the formula $\Box\, p$ is called a (*state*) *invariance formula*. If $\Box\, p$ is valid over program P, we say that p is an invariant of P, implying that all P-accessible states satisfy p.

Rule INV, presented in Fig. 7, is a complete rule for proving that assertion p is an invariant of program P.

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

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. 7. Rule INV (invariance).

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

$$p: \quad y \geq 0$$

is an invariant of program ANY-Y. We take $\varphi = p : y \geq 0$ in the rule.

Premise I1 requires showing

$$\underbrace{\cdots \wedge\ y = 0\ \wedge \cdots}_{\Theta} \quad \rightarrow \quad \underbrace{y \geq 0}_{\varphi}$$

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 3 transitions of program ANY-Y, which are $\{\tau_I, \ell_0, \ell_1, m_0\}$.

We find it sufficient to check two kinds of transitions:

- The verification condition for transition ℓ_1 is:
$$\underbrace{\cdots \wedge\ y' = y + 1\ \wedge \cdots}_{\rho_{\ell_2}} \wedge \underbrace{y \geq 0}_{\varphi} \quad \rightarrow \quad \underbrace{y' \geq 0}_{\varphi'}$$
- All other transitions $\tau \in \{\tau_I, \ell_0, m_0\}$ preserve the value of y. Therefore, their verification condition is:
$$\underbrace{\cdots \wedge\ y' = y\ \wedge \cdots}_{\rho_\tau} \wedge \underbrace{y \geq 0}_{\varphi} \quad \rightarrow \quad \underbrace{y' \geq 0}_{\varphi'}$$

Obviously, all these implications are state-valid. ▟

An assertion φ that satisfies premises I1 and I3 is called *inductive*. If φ is an inductive assertion, then we can apply rule INV, taking $p = \varphi$ and conclude that any assertion which is inductive over P is P-invariant. Rule INV can be viewed as suggesting a proof strategy that can be summarized as follows:

To prove $\Box\, p$, find an inductive assertion φ that is stronger than p.

We refer to this as the *strengthening strategy*.

4.2 Application to Parameterized Programs

The transitions of parameterized programs can be characterized by a fixed number of transition relation schemes. Therefore, premise I3 has to be established for each of the transition schemes by a proof that treats the parameter j as any other logical variable. We say that premise I3 is established by a *uniform proof* (uniform in j).

We present several examples of uniform proofs of invariance properties of parameterized programs.

Example (multiple mutual exclusion by semaphores)

The invariance property of program MPX-SEM (Fig. 4) is mutual exclusion, which is specified by

$$p: \quad \forall i, j \in [1..M]\colon\ i \neq j\colon\ \ \Box\, \underbrace{\neg\big(at\_\ell_3[i]\ \wedge\ at\_\ell_3[j]\big)}_{\psi[i,j]}\ .$$

To prove this statement, we have to establish, for arbitrary i and j, where $i, j \in [1..M]$ and $i \neq j$, the invariance of the assertion $\psi[i,j]$.

To establish the invariance of $\psi[i,j]$, we have to find an assertion φ stronger than p, i.e., implying p, which is *inductive*, i.e., satisfies premises I1 and I2. Such an assertion can be given by the conjunction $\varphi_1 \wedge \varphi_2$ of the following two assertions:

$$\varphi_1\colon \quad \forall j\colon\ y[j] \geq 0$$

$$\varphi_2\colon \quad N_{3,4} + \sum_{j=1}^{M} y[j] \ =\ 1.$$

Recall that $N_{3,4}$ is the number of processes which currently execute at locations $\ell_{3,4}$.

To show that the conjunction $\varphi_1 \wedge \varphi_2$ is inductive, i.e., satisfies premises I1 and I3, it is sufficient to show that each of φ_1, φ_2 satisfies these two premises separately.

There are two transitions that may affect φ_1 or φ_2. They are $\ell_2[i]$ and $\ell_4[i]$. We observe that $\rho_2[i]$ implies

$$at\_\ell_2[i]\ \wedge\ at'\_\ell_3[i]\ \wedge\ y[i] > 0\ \wedge\ y'[i] = y[i] - 1\ \wedge\ \forall j\colon\ j \neq i\colon\ y'[j] = y[j]$$

and $\rho_4[i]$ implies

$$at\_\ell_4[i] \wedge at'\_\ell_0[i] \wedge y'[i \oplus_M 1] = y[i \oplus_M 1] + 1\ \wedge \forall j (j \neq i \oplus_M 1)\colon\ y'[j] = y[j].$$

- φ_1 *is Inductive*

Clearly $\Theta \rightarrow \varphi_1$, that is

$$\underbrace{M \geq 2 \ \wedge \ \pi = \{\ell_0[1], \ldots, \ell_0[M]\} \ \wedge \ y[1] = 1 \ \wedge \ y[2] = \cdots = y[M] = 0}_{\Theta} \ \rightarrow \underbrace{\forall j\colon \ y[j] \geq 0}_{\varphi_1} .$$

It is easy to establish, for every $i \in [1..M]$,

$$\ell_2[i]\colon \ \underbrace{\cdots \ \wedge \ y[i] > 0 \ \wedge \ y'[i] = y[i] - 1 \ \wedge \ \forall j\colon \ j \neq i\colon \ y'[j] = y[j]}_{\rho_2[i]} \ \wedge \ \underbrace{\forall j\colon \ y[j] \geq 0}_{\varphi_1} \ \rightarrow \ \underbrace{\forall j\colon \ y'[j] \geq 0}_{\varphi_1'}$$

$$\ell_4[i]\colon \ \underbrace{\cdots \ \wedge \ y'[i \oplus_M 1] = y[i \oplus_M 1] + 1 \ \wedge \ \forall j\colon \ j \neq i \oplus_M 1\colon \ y'[j] = y[j]}_{\rho_4[i]} \ \wedge \ \underbrace{\forall j\colon \ y[j] \geq 0}_{\varphi_1} \ \rightarrow \ \underbrace{\forall j\colon \ y'[j] \geq 0}_{\varphi_1'} .$$

This establishes the preservation of φ_1 over all transitions. Consequently, φ_1 is inductive.

- φ_2 *is Inductive*

Clearly $\Theta \rightarrow \varphi_2$, that is

$$\underbrace{M \geq 2 \ \wedge \ \pi = \{\ell_0[1], \ldots, \ell_0[M]\} \ \wedge \ y[1] = 1 \ \wedge \ y[2] = \cdots = y[M] = 0}_{\Theta} \ \rightarrow \underbrace{N_{3,4} + \sum_{j=1}^{M} y[j] \ = \ 1}_{\varphi_2} .$$

The implication holds since $\pi = \{\ell_0[1], \ldots, \ell_0[M]\}$ implies $N_{3,4} = 0$ and $y[1] = 1 \wedge y[2] = \cdots = y[M] = 0$ implies $\sum_{j=1}^{M} y[j] = 1$.

The effects of $\ell_2[i]$ and $\ell_4[i]$ on φ_2 can be expressed respectively by

$$\begin{array}{llll} \ell_2[i]\colon & N_{3,4}' = N_{3,4} + 1 & \wedge & \displaystyle\sum_{j=1}^{M} y'[j] = \left(\sum_{j=1}^{M} y[j]\right) - 1 \\ \ell_4[i]\colon & N_{3,4}' = N_{3,4} - 1 & \wedge & \displaystyle\sum_{j=1}^{M} y'[j] = \left(\sum_{j=1}^{M} y[j]\right) + 1. \end{array}$$

Consequently, φ_2 is also inductive.

Clearly, $\varphi_1 \wedge \varphi_2$ implies

$$N_3 \;\leq\; N_{3,4} \;=\; 1 - \sum_{j=1}^{M} y[j] \;\leq\; 1,$$

showing that there can be at most one process at ℓ_3. Thus, $\varphi_1 \wedge \varphi_2$ implies p which claims mutual exclusion, confirming premise I2.

This establishes mutual exclusion for program MPX-SEM for an arbitrary parameter M, $M \geq 2$. ▟

Example (finding the maximum of an array)

Consider program MAX-ARRAY of Fig. 6. The initial condition of the program is given by

$$\Theta\colon \quad \pi = \{\ell_0\} \;\wedge\; \bigwedge_{i=1}^{M} \big(y[i] = \mathrm{T}\big).$$

We wish to prove the partial correctness of program MAX-ARRAY, which is given by the invariance of the assertion

$$\psi\colon \quad at\_\ell_8 \;\rightarrow\; maximal(z, x),$$

where $maximal(z, x)$ states that z is the maximal element of the array $x[1..M]$ and is an abbreviation for the formula:

$$maximal(z,\ x)\colon \quad \exists k \in [1..M]\colon\ z = x[k] \quad \wedge \quad \forall j \in [1..M]\colon\ x[j] \leq z.$$

We base our proof on the property that, in any array, one of the elements is maximal. Let $a \in [1..M]$ be the index of an element which is maximal in x. Thus,

$$\forall j \in [1..M]\colon\ x[j] \leq x[a].$$

We formulate several assertions whose conjunction can be shown to be inductive, and hence invariant:

$$\begin{array}{ll}
\varphi_1[i,j]\colon & at_\ell_2[i,j] \;\rightarrow\; x[i] < x[j] \\
\varphi_2[i]\colon & at_\ell_6[i] \;\rightarrow\; y[i] \\
\varphi_3\colon & y[a] \\
\varphi_4[i]\colon & x[i] = x[a] \;\vee\; at_\ell_0 \;\vee\; at_\ell_{1,2}[i,a] \;\vee\; \neg y[i] \\
\varphi_5\colon & N_{0..4} > 0 \;\vee\; at_\ell_{5,6}[a] \;\vee\; z = x[a]
\end{array}$$

where $N_{0..4}$ is the number of processes which currently reside at locations $\ell_0, \ldots, \ell_4$. The indices i and j range over $1, \ldots, M$.

Assertion $\varphi_1[i,j]$ claims that process $P[i,j]$ can reach location ℓ_2 only if $x[i] < x[j]$.

Assertion $\varphi_2[i]$ claims that process $Q[i]$ can reach location ℓ_6 only if $y[i]$ is true.

Assertion φ_3 claims that $y[a]$ remains true throughout the computation.

Assertion $\varphi_4[i]$ claims that if $x[i] \neq x[a]$ (and hence $x[i] < x[a]$), then either the program is still at ℓ_0, or process $P[i,a]$ is still comparing $x[i]$ with $x[a]$ at $\ell_1[i,a]$, or is at the assignment $\ell_2[i,a]$ that will eventually set $y[i]$ to F, or $y[i]$ has already been set to F. For the program to function correctly we must ensure that, by the time execution reaches ℓ_4, $y[i]$ is false for all i such that $x[i] \neq x[a]$ (and therefore $x[i] < x[a]$). Assertion $\varphi_4[i]$ guarantees this by stating that if $x[i] \neq x[a]$ and $y[i]$ is true then the execution of process $P[i,a]$ has not terminated yet. Note that process $P[i,a]$ is expected to set $y[i]$ to false.

Assertion φ_5 claims that either process $Q[a]$, which will eventually set z to $x[a]$, has not terminated yet, or z has already been set to $x[a]$ (and will retain this value).

To conclude the partial correctness of the program, we observe that at_ℓ_8 contradicts each of the disjuncts $N_{0..4} > 0$ and $at\_\ell_{5,6}[a]$, appearing in φ_5, leading to

$$at\_\ell_8 \;\rightarrow\; z = x[a],$$

which by the assumed maximality of $x[a]$ implies the invariance of

$$at\_\ell_8 \;\rightarrow\; maximal(z,\; x).$$ ▟

5 Verifying Response Properties

In this and the succeeding sections, we present rules for proving response properties of parameterized programs. The main rule for verifying response properties relies on the notion of a *well-founded domain* for measuring progress towards the goal.

5.1 Well-founded Domains

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

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

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

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

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

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

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

The Lexicographic Product

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

$\mathcal{A}$ is defined as $\mathcal{A}_1 \times \mathcal{A}_2$, i.e., the set of all pairs (a_1, a_2), such that $a_1 \in \mathcal{A}_1$ and $a_2 \in \mathcal{A}_2$.

$\succ$ is an order defined to hold between $(a_1, a_2) \in \mathcal{A}$ and $(a'_1, a'_2) \in \mathcal{A}$ iff

$$a_1 \succ_1 b_1 \quad \text{or} \quad a_1 = b_1 \ \wedge \ a_2 \succ_2 b_2.$$

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

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

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

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

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

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

According to this definition

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

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

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

For example, for $k = 3$

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

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

It is possible to make lexicographic comparisons between tuples of integers of different lengths. The convention is that the relation holding between $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_k)$ for $i < k$ is determined by lexicographically comparing $(a_1, \ldots, a_i, 0, \ldots, 0)$ to $(b_1, \ldots, b_i, b_{i+1}, \ldots, b_k)$. That is, we pad the shorter tuple by zeros on the right until it assumes the length of the longer tuple.

According to this definition, $(0, 2) \succ 0$, since $(0, 2) \succ (0, 0)$. In a similar way, $1 \succ (0, 2)$.

For assertions p, q, and $\varphi_1, \ldots, \varphi_m$,
a well-founded domain $(\mathcal{A}, \succ)$,
ranking functions $\delta_1, \ldots, \delta_m \colon \Sigma \mapsto \mathcal{A}$, and
helpful sets $h_1, \ldots, h_m \subseteq \mathcal{J}$

$$\text{JH1.} \quad p \quad \rightarrow \quad q \ \lor \ \bigvee_{j=1}^{m} \varphi_j$$

The following three premises hold for $i = 1, \ldots, m$

$$\text{JH2.} \quad \rho_\tau \ \land \ \varphi_i \quad \rightarrow \quad \begin{bmatrix} q' \ \lor \ \bigvee_{j=1}^{m} (\varphi'_j \ \land \ \delta_i \succ \delta'_j) \\ \lor \ \begin{pmatrix} \varphi'_i \ \land \ \delta_i = \delta'_i \ \land \\ \mathcal{E}(h_i) \subseteq \mathcal{E}'(h_i) \end{pmatrix} \end{bmatrix}$$

for every $\tau \in \mathcal{T}$

$$\text{JH3.} \quad \rho_\tau \ \land \ \tau \in h_i \ \land \ \varphi_i \quad \rightarrow \quad q' \ \lor \ \bigvee_{j=1}^{m} (\varphi'_j \ \land \ \delta_i \succ \delta'_j)$$

for every $\tau \in \mathcal{T}$

$$\text{JH4.} \quad \varphi_i \quad \rightarrow \quad \exists \tau \in h_i \colon \ En(\tau)$$

$$p \quad \Rightarrow \quad \Diamond\, q$$

FIG. 8. Rule JH-WELL (well-founded response with helpful sets).

5.2 The Rule

In Fig. 8, we present rule JH-WELL for proving response properties of parameterized programs. The rule uses a fixed number of assertions $\varphi_1, \ldots, \varphi_m$, and associated *sets* of helpful transitions $h_1, \ldots, h_m$, $h_i \subseteq \mathcal{J}$ for $i = 1, \ldots, m$. We refer to h_i as the *helpful set* for assertion φ_i, $i = 1, \ldots, m$. The notation $\mathcal{E}(h_i)$ denotes the set of transitions belonging to h_i that are enabled. Both h_i and $\mathcal{E}(h_i)$ may depend on the state. The inclusion $\mathcal{E}(h_i) \subseteq \mathcal{E}'(h_i)$ is an abbreviation for the formula

$$\forall \tau \in h_i \colon \quad En(\tau) \rightarrow En'(\tau),$$

implying that any helpful transition that was enabled before the transition is still enabled after the transition.

The rule uses a well-founded domain $(\mathcal{A}, \succ)$, and ranking functions $\delta_i \colon \Sigma \mapsto \mathcal{A}$, mapping states of the system to elements of $\mathcal{A}$. The ranking functions measure progress of the computation towards the goal q.

Premise JH1 requires that every p-position either satisfies q or satisfies one of $\varphi_1, \ldots, \varphi_m$.

Premise JH2 requires that the application of an arbitrary transition τ to a φ_i-position k leads to a successor position $k' = k+1$ satisfying one of the following:

- k' satisfies q, or
- k' satisfies some φ_j with a rank $\delta_j(k')$ smaller than $\delta_i(k)$, or
- k' satisfies φ_i with a rank $\delta_i(k')$ equal to $\delta_i(k)$, and with a set of enabled helpful transitions $\mathcal{E}'(h_i)$ containing $\mathcal{E}(h_i)$.

Note that in the case of no observable progress, described by the third clause above, we require the persistence of enabled helpful transitions. That is, any helpful transition that is enabled at k is also enabled at k'. Position k' may have some additional enabled helpful transitions.

Premise JH3 requires that the application of a helpful transition $\tau \in h_i$, to a φ_i-position k, leads to a successor position k' which either satisfies q or satisfies some φ_j with a rank lower than that of k.

Since premise JH3 covers the cases of $\tau \in h_i$, it is sufficient to establish premise JH2 only for $\tau \notin h_i$.

Premise JH4 requires that every φ_i-position has at least one helpful transition which is enabled.

Justification To justify the rule, assume a computation such that p holds at position k and no later position $i \geq k$ satisfies q. By JH1 some φ_j must hold at position k. Let φ_{i_1} be the formula holding at k, and denote the rank δ_{i_1} at k by u_1. By JH4, there exists some helpful transition $\tau_{i_1} \in h_{i_1}$ which is enabled at position k.

Consider the transition τ taken at position k, leading into position $k+1$. By JH2, either position $k+1$ has a lower rank $u_2 \prec u_1$, or it has an equal rank $u_2 = u_1$, but then τ_{i_1} is still helpful and enabled at $k+1$. In the case of equal ranks, we can continue the argument from $k+1$ to $k+2$, $k+3$, etc. However, we cannot have all positions $i > k$ with the same rank. To see this, assume that all positions beyond k do have the same rank. By JH2, this implies that τ_{i_1} is continuously helpful and enabled. By JH3, τ_{i_1} is not taken beyond k because taking it would have led to a state with a rank lower than u_1. Thus, our assumption that all positions beyond k have the same rank leads to the situation that τ_{i_1} is continuously enabled and not taken, violating the justice requirement for τ_{i_1}.

Thus, eventually, we must reach a position k_2, $k_2 > k$, with rank u_2, where $u_2 \prec u_1$. In a similar way we can establish the existence of a position $k_3 > k_2$, with rank u_3, where $u_3 \prec u_2$. Continuing in this manner, we construct an infinitely descending sequence $u_1 \succ u_2 \succ u_3 \succ \cdots$ of elements of $\mathcal{A}$. This is impossible, due to the well-foundedness of $\succ$ on $\mathcal{A}$.

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

Example Consider the parameterized program TRIVIAL of Fig. 9. This program satisfies the response property

$$\underbrace{N_0 > 0}_{p} \quad \Rightarrow \quad \Diamond \underbrace{N_1 > 0}_{q},$$

which states that if some process is at ℓ_0, then eventually some process will be at ℓ_1.

$$\boxed{\begin{array}{l} \quad\ \ \textbf{in} \quad M \quad : \textbf{integer where } M > 0 \\[2ex] \displaystyle\mathop{\Big\|}_{i=1}^{M} P[i] :: \quad \big[\ell_0\text{: } \textbf{skip};\ \ell_1\text{: } \textbf{skip}; \ell_2\text{: } \ \big] \end{array}}$$

FIG. 9. Program TRIVIAL.

To prove this property, we use a simple case of the rule for $m = 1$. As intermediate assertion we take

$$\varphi = p\text{:} \quad N_0 > 0.$$

As ranking function we take $\delta = 1$. The helpful set is

$$h\text{:} \quad \big\{\ell_0[1],\ \ldots,\ \ell_0[M]\big\}.$$

For this simple example, the goal q: $N_1 > 0$ is achieved after taking a single helpful transition. Therefore, we do not have to consider helpful steps that decrease the ranking function δ, which can be chosen to be a constant.

We consider the premises of rule JH-WELL.

- Premise JH1

  This premise requires showing

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

  Since $\varphi = p$, this implication is trivially valid.

- Premise JH2

  This premise requires showing for each $\tau \in \mathcal{T}$

  $$\rho_\tau \quad \wedge \quad \underbrace{N_0 > 0}_{\varphi} \quad \rightarrow \quad \begin{bmatrix} \cdots \ \vee \ \cdots \\ \quad \vee \\ \underbrace{N_0' > 0}_{\varphi'} \ \wedge \ \underbrace{1 = 1}_{\delta = \delta'} \ \wedge \ \mathcal{E}(h) \subseteq \mathcal{E}'(h) \end{bmatrix}.$$

  As previously observed, it is sufficient to consider transitions $\tau \notin h$, i.e., $\tau \neq \ell_0[k]$ for any $k \in [1..m]$. These transitions do not change the set of j's for which $at\_\ell_0[j]$ holds. Consequently, these transitions preserve the value of N_0 and the set $\mathcal{E}(h)$.

- Premise JH3

  This premise requires showing for each $\tau \in \mathcal{T}$

$$\rho_\tau \ \wedge \ \tau \in h \ \wedge \ N_0 > 0 \quad \rightarrow \quad \underbrace{N_1' > 0}_{q'} \ \vee \ \cdots,$$

that is, we have to show that for each $i \in [1..M]$

$$\rho_{\ell_0[i]} \ \wedge \ N_0 > 0 \quad \rightarrow \quad N_1' > 0 \ \vee \ \cdots.$$

Clearly, $\rho_{\ell_0[i]}$ implies $at'\_\ell_1[i]$ from which $N_1' > 0$ follows.

- Premise JH4

  This premise requires showing

  $$\underbrace{N_0 > 0}_{\varphi} \quad \rightarrow \quad \exists i \in [1..M]\colon \ \underbrace{En(\ell_0[i])}_{En(h)}.$$

  By $N_0 > 0$, there is at least some i for which $at\_\ell_0[i]$ is true, implying that $\ell_0[i]$ is enabled. ▟

A Notation for Helpful Sets

Let $\underset{p \in R}{\Big\|} S[p]$ be a parallel statement, where the parameter p ranges over some set R (e.g., $[1..M]$). Let ℓ_k be a location within statement S. We write T_k to denote the set of transitions of the form $L_k[p]$, i.e.,

$$T_k\colon \quad \{\ell_k[p] \mid p \in R\}$$

For example, in program MAX-ARRAY of Fig. 10 (also Fig. 6), T_2 denotes the set of transitions $\{\ell_2[i,j] \mid i,j \in [1..M]\}$, while T_5 denotes the set $\{\ell_5[i] \mid i \in [1..M]\}$.

This notation is extended to denote sets of transitions over a set of labels such as $T_{2,3}$ and sets of transitions over ranges of labels such as $T_{5..7}$. For example, $T_{0..7}$ denotes the set of all just transitions of program MAX-ARRAY.

Example (program MAX-ARRAY)

As a second example, reconsider program MAX-ARRAY (Fig. 10) for parallel computation of the maximum of an integer array.

In Section 4, dealing with safety, we proved the partial correctness of MAX-ARRAY, expressed by

$$at\_\ell_8 \quad \Rightarrow \quad maximal(z,\ x).$$

To complete the verification of this program, we have to prove its termination, expressed by the response property

$$\underbrace{at\_\ell_0}_{p} \quad \Rightarrow \quad \Diamond \underbrace{at\_\ell_8}_{q}.$$

Intuitively, termination is obvious, since each process is a sequential program with no loops. To prove it formally, using rule JH-WELL, we identify a well-founded domain

$$(\mathcal{A}, \succ)\colon \quad ([1..8],\ >) \ \times \ ([0..M^2],\ >)$$

The parameterized intermediate assertions, helpful transitions, and the ranking functions are given in the table of Fig. 11.

Note that we refer to the exit transitions from the cooperation statements ℓ_0 and ℓ_4 by the names ℓ_3 and ℓ_7, respectively. These are *single*

$$
\begin{array}{ll}
\textbf{in} & M\colon \textbf{integer where } M \geq 1 \\
 & x\ : \textbf{array } [1..M] \textbf{ of integer} \\
\textbf{local } y & : \textbf{array } [1..M] \textbf{ of boolean where } y = \text{T} \\
\textbf{out}\ \ z & : \textbf{integer}
\end{array}
$$

$$
\begin{array}{l}
\ell_0\colon \displaystyle\mathop{\Big\|}_{i,j \in [1..M]} P[i,j] :: \left[\begin{array}{l} \ell_1\colon \textbf{ if } x[i] < x[j] \textbf{ then } \ell_2\colon\ y[i] := \text{F} \\ \ell_3\colon \end{array}\right] \\
\ell_4\colon \displaystyle\mathop{\Big\|}_{i=1}^{M} Q[i] :: \left[\begin{array}{l} \ell_5\colon \textbf{ if } y[i] \textbf{ then } \ell_6\colon\ z := x[i] \\ \ell_7\colon \end{array}\right] \\
\ell_8\colon
\end{array}
$$

FIG. 10. Program MAX-ARRAY (finding the maximum of an array).

$\varphi_8\colon\ at\_\ell_0$	$h_8\colon\ \{\ell_0\}$	$\delta_8\colon\ 8$
$\varphi_7\colon\ N_1 > 0 \wedge N_{1..3} = M^2$	$h_7\colon\ T_1$	$\delta_7\colon\ (7, N_1)$
$\varphi_6\colon\ N_2 > 0 \wedge N_{2,3} = M^2$	$h_6\colon\ T_2$	$\delta_6\colon\ (6, N_2)$
$\varphi_5\colon\ N_3 = M^2$	$h_5\colon\ \{\ell_3\}$	$\delta_5\colon\ 5$
$\varphi_4\colon\ at\_\ell_4$	$h_4\colon\ \{\ell_4\}$	$\delta_4\colon\ 4$
$\varphi_3\colon\ N_5 > 0 \wedge N_{5..7} = M$	$h_3\colon\ T_5$	$\delta_3\colon\ (3, N_5)$
$\varphi_2\colon\ N_6 > 0 \wedge N_{6,7} = M$	$h_2\colon\ T_6$	$\delta_2\colon\ (2, N_6)$
$\varphi_1\colon\ N_7 = M$	$h_1\colon\ \{\ell_7\}$	$\delta_1\colon\ 1$

FIG. 11. Assertions, transitions and rankings for program MAX-ARRAY.

transitions that are enabled only when all processes within the cooperation statement have reached their terminal locations ℓ_3 and ℓ_7.

As we see, the table refers to eight parameterized assertions and helpful transition sets. Assertions φ_6 and φ_7 are associated with helpful sets of size M^2. Assertions φ_2 and φ_3 are associated with helpful sets of size M. The remaining assertions are associated with helpful sets of size 1. Note that the ranking of the assertions with singleton helpful sets consist of a single positive integer $\delta \in \{1, 4, 5, 8\}$ which should be interpreted as abbreviation for the pair $(\delta, 0)$.

The proof uses the following invariant which can be established by rule INV:

$$\psi: \quad \begin{pmatrix} (N_{0,4,8} = 1 & \wedge & N_{1..3,5..7} = 0) & \vee \\ (N_{1..3} = M^2 & \wedge & N_{0,4..8} = 0\ \ ) & \vee \\ (N_{5..7} = M & \wedge & N_{0..4,8} = 0\ \ ) & \end{pmatrix}$$

It is not difficult to check the premises of rule JH-WELL and verify that they are all satisfied.

For example, let us check premise JH3 for assertion φ_2. We show that the following implication is state valid:

$$\rho_{\ell_6[i]} \wedge \underbrace{N_6 > 0 \wedge N_{6,7} = M}_{\varphi_2} \rightarrow \cdots \vee \underbrace{N_7' = M \wedge (2, N_6) \succ (1, 0)}_{\varphi_1' \wedge \delta_2 \succ \delta_1'}$$
$$\vee \underbrace{N_6' > 0 \wedge N_{6,7}' = M \wedge (2, N_6) \succ (2, N_6')}_{\varphi_2' \wedge \delta_2 \succ \delta_2'}.$$

Assertion φ_2: $N_6 > 0 \wedge N_{6,7} = M$ implies $N_7 \leq M - 1$. To show the validity of the implication, we consider two cases. If $N_7 = M - 1$, then transition $\ell_6[i]$ leads to a successor state in which $N_7 = M$, establishing φ_1'. If $N_7 < M - 1$, there exists another index $k \neq i$ such that $at\_\ell_6[k]$ holds at the state before the transition and therefore also at the successor state. This establishes φ_2' and a rank decrease at the successor state. ■

6 Representation by Verification Diagrams

The main task of constructing a proof by rule JH-WELL involves the identification of the constructs required by the rule, which consist of the auxiliary assertions, their associated helpful sets, and ranking functions. These constructs can be conveniently and effectively presented by a diagram that summarizes the assertions with their associated helpful transitions and ranking functions, and displays the possible transitions between the assertions.

We define a *verification diagram* to be a directed labeled graph constructed as follows:

- *Nodes* in the graph are labeled by assertions and their corresponding ranking functions. We will often refer to the node by the assertion labeling it.
- *Edges* in the graph represent transitions between assertions. Each edge connects one assertion to another and is labeled by an expression denoting a set of transitions in the program. We distinguish two types of edges. Double-lined edges represent helpful transitions, while single-lined edges represent unhelpful transitions.
- One of the nodes is designated as a *terminal node* ("goal" node). In the graphical representation, this node is distinguished by having a boldface boundary. No edges depart from a terminal node.

It is required that every node except the goal node has a double edge departing from it. It is also required that, if a node has a departing edge labeled by set expression S_1 and another departing edge labeled by S_2, then either $S_1 = S_2$ or $S_1 \cap S_2 = \emptyset$. Furthermore, the same set expression S cannot label both a single and a double edge departing from the same node.

Verification and Enabling Requirements for Verification Diagrams

Consider a nonterminal node labeled by assertion φ and ranking function δ, and S, a set-valued expression specifying a set of transitions. Let $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the successors of φ by edges labeled with S, and $\delta_1, \ldots, \delta_k$ be the ranking functions of these successors.

- If S labels a single (-lined) edge, then the following verification condition is implied for every $\tau \in S$:

$$\rho_\tau \ \wedge \ \varphi \quad \rightarrow \quad \left(\begin{array}{l} (\varphi \ \wedge \ \delta = \delta' \ \wedge \ \mathcal{E}(h) \subseteq \mathcal{E}'(h)) \ \vee \ (\varphi \ \wedge \ \delta \succ \delta') \\ \quad \vee \\ (\varphi_1 \ \wedge \ \delta \succ \delta'_1) \ \vee \ \cdots \ \vee \ (\varphi_k \ \wedge \ \delta \succ \delta'_k) \end{array} \right),$$

  where h is the helpful set for assertion φ, i.e., the set labeling double edges departing from φ's node. If $k = 0$, i.e., S denotes the set of transitions which do not label any edge departing from φ, then the second disjunct is omitted.

- If $S = h$ labels a double edge identifying S as the helpful set for φ, then the following verification condition is implied for every $\tau \in \mathcal{T}$:
  $\rho_\tau \ \wedge \ \tau \in h \ \wedge \ \varphi \quad \rightarrow \quad (\varphi_1 \ \wedge \ \delta \succ \delta'_1) \ \vee \ \cdots \ \vee \ (\varphi_k \ \wedge \ \delta \succ \delta'_k)$.
- If h labels a double edge departing from φ, we require
  $\varphi \quad \rightarrow \quad \exists \tau \in h\colon \ En(\tau)$.

Note that in the case of an unhelpful transition, we allow a τ-successor with a rank equal to that of φ, provided it satisfies the same assertion φ. To make the treatment uniform, the preceding definitions require a ranking δ_0 for the terminal node and that transitions entering this node cause a rank decrease.

Valid Verification Diagrams

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

The consequences of having a valid verification diagram are stated in the following claim:

> **Claim R** A P-valid verification diagram establishes that the response formula
> $$\bigvee_{j=0}^{m} \varphi_j \quad \Rightarrow \quad \Diamond \varphi_0$$

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

$$\text{(R1)} \quad p \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \text{(R2)} \quad \varphi_0 \;\rightarrow\; q,$$

then we can conclude the validity of

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

The type of verification diagram defined here is specialized for the presentation of proofs by rule JH-WELL. We refer the reader to [8] for a general description of other types of verification diagrams.

Example (program TRIVIAL)

In Fig. 12 we present a verification diagram establishing the response property

$$\underbrace{N_0 > 0}_{p} \;\Rightarrow\; \Diamond \underbrace{N_1 > 0}_{q},$$

for program TRIVIAL of Fig. 9.

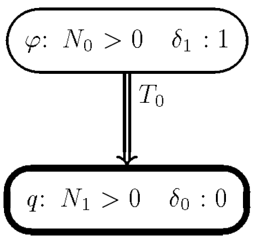

FIG. 12. A verification diagram for program TRIVIAL.

Example (program MAX-ARRAY)

We return to the example of program MAX-ARRAY (Fig. 10), for which we wish to prove termination, specified by

$$\underbrace{at\_\ell_0}_{p} \;\Rightarrow\; \Diamond \underbrace{at\_\ell_8}_{q}.$$

We present two proofs of this property using rule JH-WELL. In Fig. 13 we present a verification diagram which represents in a graphical form the previous proof of this property that appeared in the table of Fig. 11.

The second proof illustrates the power of rule JH-WELL, by using only a single intermediate assertion and single ranking function and helpful set:

$$\varphi: \quad \left(\begin{array}{lll} at\_\ell_0 \;\wedge\; N_{1..7} = 0 & \vee \quad N_{1..3} = M^2 \;\wedge\; N_{0,4..7} = 0 & \vee \\ at\_\ell_4 \;\wedge\; N_{0..3,5..7} = 0 & \vee \qquad N_{0..4} = 0 \;\wedge\; N_{5..7} = M & \end{array} \right)$$

$$\delta: \quad (N_0,\; N_1,\; N_2,\; N_3,\; N_4,\; N_5,\; N_6,\; N_7)$$

$$h: \quad T_{0..7}$$

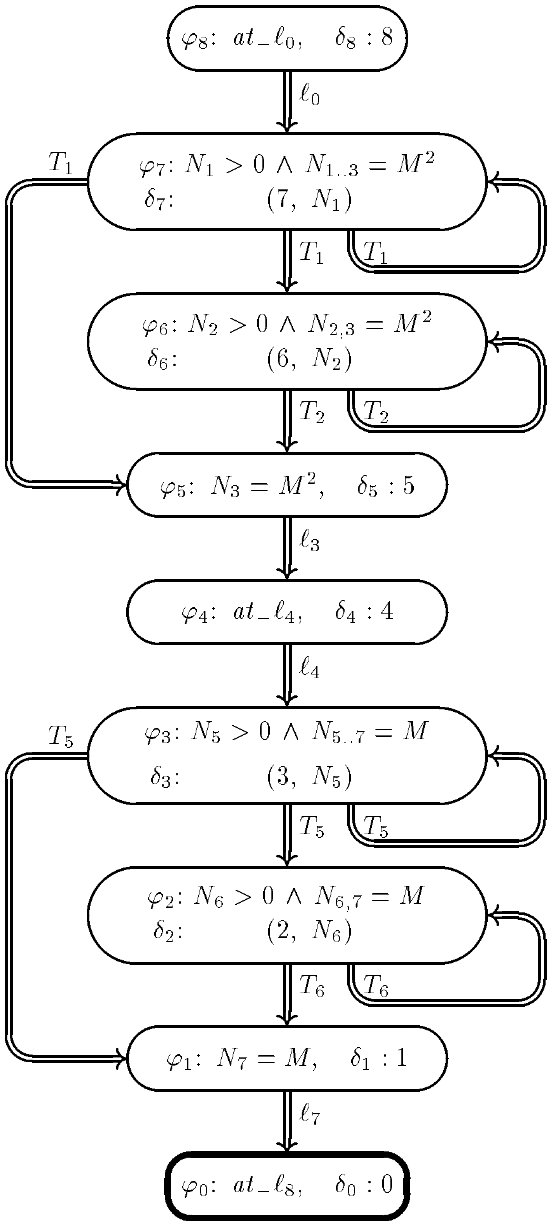

FIG. 13. A verification diagram for program MAX-ARRAY.

Thus, the well-founded domain used is $\mathbb{N}^8$, consisting of 8-tuples of natural numbers. It is not difficult to see that each of the transitions in $T_{0..7}$ causes δ to decrease, and that at leat one of these transitions is enabled on every φ-state. ▪

6.1 Structured Verification Diagrams

We introduce some encapsulation conventions, inspired by the Statecharts formalism of Harel [3]. These conventions lead to more structured and hierarchical diagrams and improve the readability and manageability of large complex diagrams.

The basic construct of encapsulation is the introduction of a *compound node* containing internal nodes. We refer to the contained nodes as the *descendants* of the compound node. Nodes that are not compound are called *basic nodes*.

- *Departing edges*
  An edge departing from a compound node is interpreted as though it departed from each of its descendants. We thus have the graphical equivalence presented in Fig. 14.

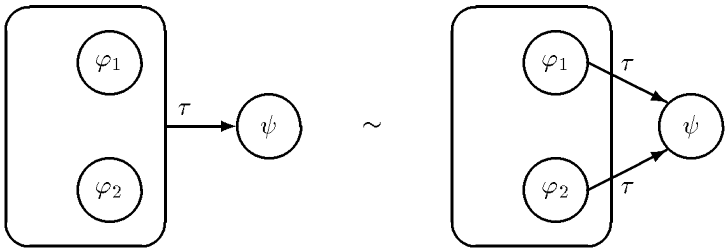

FIG. 14. Departing edges.

- *Arriving edges*
  In a similar way, an edge arriving at a compound node is interpreted as though it arrived at each of its descendants. Thus, we have the graphical equivalence presented in Fig. 15.
- *Common factors*
  An assertion φ labeling a compound node is interpreted as though it were a conjunct added to each of its descendants. This is described by the graphical equivalence presented in Fig. 16.

6.2 Distributing the Ranking Functions

To make verification diagrams more readable, we introduce an additional encapsulation convention. The previous convention allowed a compound

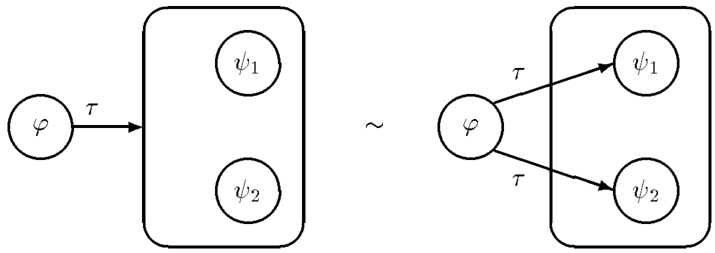

FIG. 15. Arriving edges.

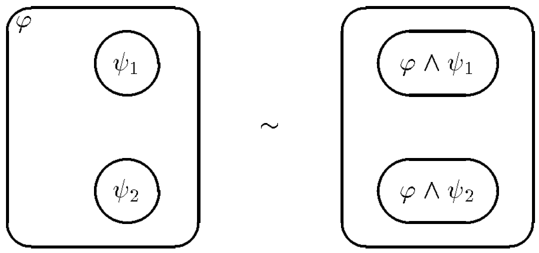

FIG. 16. Common factors.

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

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

$$\delta\colon\ f,$$

where f is some ranking function mapping states into a well-founded domain $\mathcal{A}$.

Consider a basic node n_i labeled by assertion φ_i and local ranking function f_b. Assume that node n_i is contained in a nested sequence of compound nodes that are labeled by ranking labels $\delta\colon f_1, \ldots, \delta\colon f_m$, as we go from the outermost compound node towards n_i. This situation is depicted in Fig. 17. Then the full ranking function associated with the node φ_i is given by the tuple

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

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

Another part of the rank distribution convention allows us to omit the local rank labeling a node φ_i altogether. This is interpreted as though the

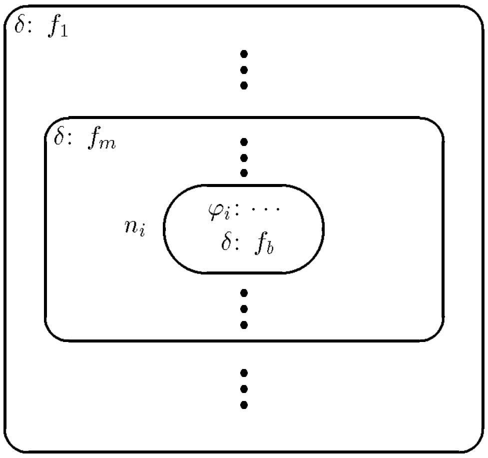

FIG. 17. Encapsulated sequence of nodes.

node was labeled with the ranking function δ: i, where i is the index of the node (and the assertion labeling it).

Example (mutual exclusion by turn setting)

Consider program TURN of Fig. 18 which manages mutual exclusion between an arbitrary number of processes by turn setting. The program uses a shared variable t, which is initially 0. Each process $P[i]$, interested in entering its critical section, loops in $\ell_{2..5}$ trying to change t to the identity number of $P[i]$, namely i. The grouped statement at ℓ_3 is such that it sets t to i only if it finds $t = 0$. If $P[i]$ finds $t \neq 0$ it does not modify t but still moves to ℓ_4. The angular brackets around the statement imply that the complete statement is executed as one transition in a single step. At ℓ_4 the process checks whether its execution of ℓ_3 succeeded in setting t to i. Finding a value $t = i$ at ℓ_4, $P[i]$ enters its critical section ℓ_5. Otherwise it returns to ℓ_2 and then to ℓ_3 to try again.

Mutual exclusion is ensured by the invariant

$$\chi: \quad i \in L_{5,6} \quad \rightarrow \quad t = i.$$

We wish to show a weak-accessibility property, formulated by

$$\psi: \quad \underbrace{N_2 > 0 \ \wedge \ t = 0}_{p} \quad \Rightarrow \quad \Diamond \underbrace{N_5 > 0}_{q} \ .$$

This formula states that if some process is at ℓ_2 with $t = 0$, then eventually some process will get to ℓ_5. We refer to it as *weak accessibility* (sometimes also as *communal accessibility*), since it does not guarantee that the same process detected at ℓ_2 with $t = 0$ will be the one to enter ℓ_5. The stronger property of accessibility, claiming that any individual process visiting ℓ_2 with $t = 0$ will eventually reach ℓ_5, which can be expressed by the formula

$$
\begin{array}{l}
\textbf{in} \quad\ M\text{: } \textbf{integer where } M > 1 \\
\textbf{local } t \ \ \text{: } \textbf{integer where } t = 0
\end{array}
$$

$$
\mathop{\Big\|}_{i=1}^{M} P[i] :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\qquad \left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } \textbf{while } t \neq i \textbf{ do} \\
\qquad \left[\begin{array}{l}
\ell_3\text{: } \langle \textbf{if } t = 0 \textbf{ then } t := i \rangle \\
\ell_4\text{: } \textbf{if } t = i \textbf{ then} \\
\qquad \ell_5\text{: } \textbf{critical}
\end{array}\right] \\
\ell_6\text{: } t := 0
\end{array}\right]
\end{array}\right]
$$

FIG. 18. Program TURN (mutual exclusion by turn setting).

$$at\_\ell_2[i] \ \wedge \ t = 0 \quad \Rightarrow \quad \diamondsuit \ at\_\ell_5[i],$$

is not valid for this program.

To verify the weak-accessibility property expressed by ψ, we use the following in rule JH-WELL:

$$
\begin{array}{llllll}
\varphi_3\text{:} & N_2 > 0 \ \wedge \ N_3 = 0 \ \wedge \ t = 0 & \delta_3\text{:} & 3 & h_3\text{:} & T_2 \\
\varphi_2\text{:} & N_3 > 0 \ \wedge \ t = 0 & \delta_2\text{:} & 2 & h_2\text{:} & T_3 \\
\varphi_1\text{:} & at\_\ell_4[t] \ \wedge \ t \neq 0 & \delta_1\text{:} & 1 & h_1\text{:} & \ell_4[t].
\end{array}
$$

We proceed to check the premises of rule JH-WELL.

- Premise JH1 requires showing $p \rightarrow \cdots \vee \varphi_2 \vee \varphi_3$. Since $N_3 \geq 0$ is an obvious invariant of program TURN, it follows that p: $N_2 > 0 \wedge t = 0$ implies φ_2: $N_3 > 0 \wedge t = 0$ or φ_3: $N_2 > 0 \wedge N_3 = 0 \wedge t = 0$.
- Premise JH2 will be checked for each $i = 1, 2, 3$.
  - φ_3: $N_2 > 0 \ \wedge \ N_3 = 0 \ \wedge \ t = 0$

  Most of the transitions satisfy JH2 for φ_3 by preserving φ_3 and $\mathcal{E}(h_3)$. The only transitions that may falsify φ_3 or disable transitions in h_3: T_2 are transitions of the form $\ell_2[i]$. Transition $\ell_2[i]$ leads to a state satisfying φ_2: $N_3 > 0 \wedge t = 0$.
  - φ_2: $N_3 > 0 \ \wedge \ t = 0$

  The only transitions that can falsify φ_2 or disable a transition belonging to h_2: T_3 are of the form $\ell_3[i]$. However, this transition leads to a state satisfying φ_1: $at\_\ell_4[t] \wedge t \neq 0$.
  - φ_1: $at\_\ell_4[t] \ \wedge \ t \neq 0$

  The only transition that can falsify φ_1 is $\ell_4[t]$, which leads to a state satisfying the goal assertion q: $N_5 > 0$.
- Premise JH3 will be checked for each $i = 1, 2, 3$.

- φ_3: $N_2 > 0 \ \wedge \ N_3 = 0 \ \wedge \ t = 0$

  Here we show for each $\tau \in h_3 {:} T_2$

  $$\rho_\tau \ \wedge \ \varphi_3 \ \rightarrow \ \varphi_2' \ \wedge \ \delta_3 \succ \delta_2'.$$

  That is,

  $$\rho_{\ell_2[i]} \ \wedge \ \underbrace{\cdots \ \wedge \ t = 0}_{\varphi_3} \ \rightarrow \ \underbrace{N_3' > 0 \ \wedge \ t' = 0}_{\varphi_2'} \ \wedge \ 3 > 2.$$

  Since $\rho_{\ell_2[i]}$ implies $at'\_\ell_3[i]$ and $t' = t$, the implication is valid.

- φ_2: $N_3 > 0 \ \wedge \ t = 0$

  Here we show for each $\tau \in h_2 {:} T_3$

  $$\rho_\tau \ \wedge \ \varphi_2 \ \rightarrow \ \varphi_1' \ \wedge \ \delta_2 \succ \delta_1'.$$

  That is,

  $$\rho_{\ell_3[i]} \ \wedge \ \underbrace{N_3 > 0 \ \wedge \ t = 0}_{\varphi_2} \ \rightarrow \ \underbrace{at'\_\ell_4[t'] \ \wedge \ t' \neq 0}_{\varphi_1'} \ \wedge \ 2 > 1.$$

  Since $\rho_{\ell_3[i]}$ under $t = 0$ implies $at'\_\ell_4[i]$ and $t' \neq 0$, the right-hand side follows.

- φ_1: $at\_\ell_4[t] \ \wedge \ t \neq 0$

  We will show

  $$\rho_{\ell_4[t]} \ \wedge \ \underbrace{at\_\ell_4[t] \ \wedge \ t \neq 0}_{\varphi_1} \ \rightarrow \ \underbrace{N_5' > 0}_{q'} \ .$$

  Since $\rho_{\ell_4[t]}$ and $t \neq 0$ imply $at'\_\ell_5[i]$, the right-hand side follows.

- Premise JH4 is obviously satisfied, since φ_i implies $\mathcal{E}(h_i) \neq \emptyset$ for $i = 1, 2, 3$.

We can summarize this JH-WELL proof in the verification diagram presented in Fig. 19. Note the assertion $t = 0$ that has been factored out of φ_3 and φ_2. ▟

6.3 State-dependent Helpful Sets

In all the preceding examples, the helpful sets h_i, $i = 1, \ldots, m$, depended on the index i but not on the state (or position) at which they were evaluated. This is not always sufficient and, in more complicated examples, we encounter cases where state-dependent helpful sets are necessary.

Consider program RACE, presented in Fig. 20. This program consists of M competing processes. The first process $P[k_1]$ to perform statement ℓ_1 will set its local variable $t[k_1]$ to 1, while incrementing the shared variable y to 2. This process eventually gets back to location ℓ_0 and, on finding $t[k_1] = 1$, will proceed to ℓ_2. All processes performing statement ℓ_1 later than $P[k_1]$ will obtain values of $t[k]$ greater than 1, and cannot exit the while loop at ℓ_0.

The property we wish to establish for this program is that eventually some process reaches ℓ_2. This property is stated by the following response formula:

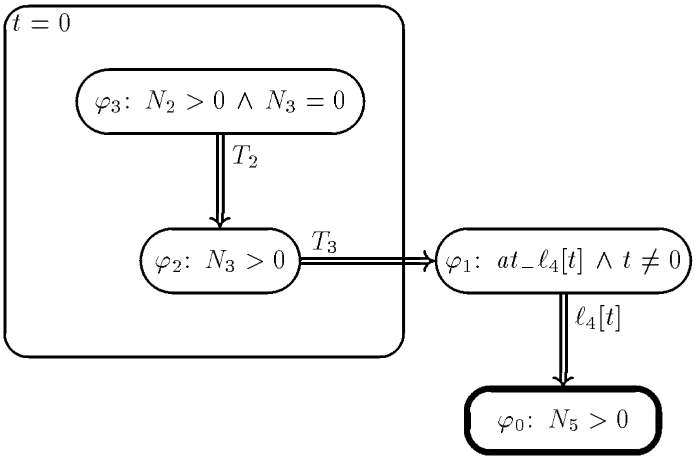

FIG. 19. Verification diagram for program TURN.

$$
\begin{array}{l}
\textbf{in} \quad M\text{: } \textbf{integer where } M > 0 \\
\textbf{local } y \text{ : } \textbf{integer where } y = 1 \\
\\
\mathop{\Big\|}_{k=1}^{M} P[k] :: \left[\begin{array}{l}
\textbf{local } t\text{: } \textbf{integer where } t = 0 \\
\ell_0\text{: } \textbf{while } t \neq 1 \textbf{ do} \\
\qquad \ell_1\text{: } (t,\ y) := (y,\ y+1) \\
\ell_2\text{:}
\end{array}\right]
\end{array}
$$

FIG. 20. Program RACE.

$$\underbrace{N_0 > 0 \ \land \ y = 1}_{p} \ \Rightarrow \ \Diamond \underbrace{N_2 > 0}_{q} \ .$$

It is easy to trace the progress of the computation towards the goal $N_2 > 0$ through the stages

$$\varphi_3\text{:} \quad N_0 > 0 \ \land \ y = 1$$

(initial) and

$$\varphi_2\text{:} \quad N_1 > 0 \ \land \ y = 1,$$

up to

$$\varphi_1\text{:} \quad \exists k\text{: } at\_\ell_0[k] \ \land \ t[k] = 1.$$

The question is how to justify the last helpful step from φ_1 to

$$\varphi_0\text{:} \quad N_2 > 0.$$

More specifically, the problem is the identification of the set h_1 of transitions that are helpful for φ_1.

A natural candidate for h_1 is T_0: $\{\ell_0[k] \mid k \in [1..M]\}$. Unfortunately, this does not work since not all $\ell_0[k]$-transitions are helpful. A transition $\ell[k]$ such that $t[k] \neq 1$ does not lead from φ_1 to φ_0 but is only guaranteed to preserve φ_1.

The right choice for the helpful set of φ_1 is

$$h_1: \quad \{\ell_0[k] \ \mid \ k \in [1..M] \ \land \ at\_\ell_0[k] \ \land \ t[k] = 1\}$$

This set is not fixed and may contain different transitions in different states. We say that h_1 is a *state-dependent* helpful set. However, as required by premise JH4 of rule JH-WELL, the set h_1 contains at least one enabled transition in each state satisfying φ_1: $\exists k \in [1..M]$: $at\_\ell_0[k] \land t[k] = 1$.

Consequently, we choose assertions, helpful sets, and ranking functions as follows:

φ_3: $N_0 > 0$	h_3: T_0	δ_2: 3
φ_2: $N_1 > 0$	h_2: T_1	δ_2: 2
φ_1: $\exists k \in [1..M]$: $at\_\ell_0[k] \land t[k] = 1$	h_1: $\{\ell_0[k] \mid k \in [1..M] \land at\_\ell_0[k] \land t[k] = 1\}$	δ_1: 1
φ_0: $N_2 > 0$.		

Note that φ_1 can also be written as $h_1 \neq \emptyset$.

It is straightforward to check that premise JH1 and premises JH2–JH4 for $i = 2, 3$ are all P-state valid. Let us dwell a while longer on premises JH2–JH4 for $i = 1$.

- Premise JH2 for $i = 1$ can be established by showing

$$\rho_\tau \ \land \ \underbrace{\exists k \in [1..M]\colon\ at\_\ell_0[k] \ \land \ t[k] = 1}_{\varphi_1} \ \rightarrow$$
$$\cdots \lor \cdots \lor \left(\varphi_1' \ \land \ 1 = 1 \ \land \ \mathcal{E}(h_1) \subseteq \mathcal{E}'(h_1)\right).$$

  As previously observed, it is sufficient to establish JH2 only for $\tau \notin h_1$. Obviously, no transition $\tau \notin h_1$ can change the local state (location and value of t) of any process $P[k]$ satisfying $at\_\ell_0[k] \land t[k] = 1$. Consequently φ_1 will also hold after the transition, and $\mathcal{E}(h_1) = h_1 \subseteq h_1' = \mathcal{E}'(h_1)$.

- Premise JH3 for $i = 1$ can be established by showing

$$\rho_{\ell_0[k]} \ \land \ \underbrace{at\_\ell_0[k] \ \land \ t[k] = 1}_{\tau \in h_1} \ \land \ \cdots \ \rightarrow \ \underbrace{N_2' > 0}_{q'} \ \lor \ \cdots .$$

  Clearly, taking $\ell_0[k]$ when $t[k] = 1$ establishes, by $\rho_{\ell_0[k]}$, $at'\_\ell_2[k]$ from which $N_2' > 0$ follows.

- Consider premise JH4 for $i = 1$. Observe that

$$\exists \tau \in h_1\colon En(\tau) \ = \ \exists \tau \in \{\ell_0[k] \mid k \in [1..M] \land at\_\ell_0[k] \land t[k] = 1\}\colon En(\tau)$$

  can be simplified to

$$\exists k \in [1..M]\colon\ at\_\ell_0[k] \ \wedge\ t[k] = 1 \ \wedge\ En\big(\ell_0[k]\big)$$

which is equivalent to

$$\underbrace{\exists k \in [1..M]\colon\ at\_\ell_0[k] \ \wedge\ t[k] = 1}_{\varphi_1}\ .$$

Thus, premise JH4 reduces to the trivial implication

$$\underbrace{\exists k \in [1..M]\colon\ at\_\ell_0[k] \ \wedge\ t[k] = 1}_{\varphi_1} \ \rightarrow\ \underbrace{\exists k \in [1..M]\colon\ at\_\ell_0[k] \ \wedge\ t[k] = 1}_{\exists \tau \in h_1\colon\ En(\tau)}\ .$$

This concludes the proof that property $N_0 > 0 \wedge y = 1 \Rightarrow \diamondsuit(N_2 > 0)$ is valid over program RACE. In Fig. 21, we present a verification diagram representing this proof.

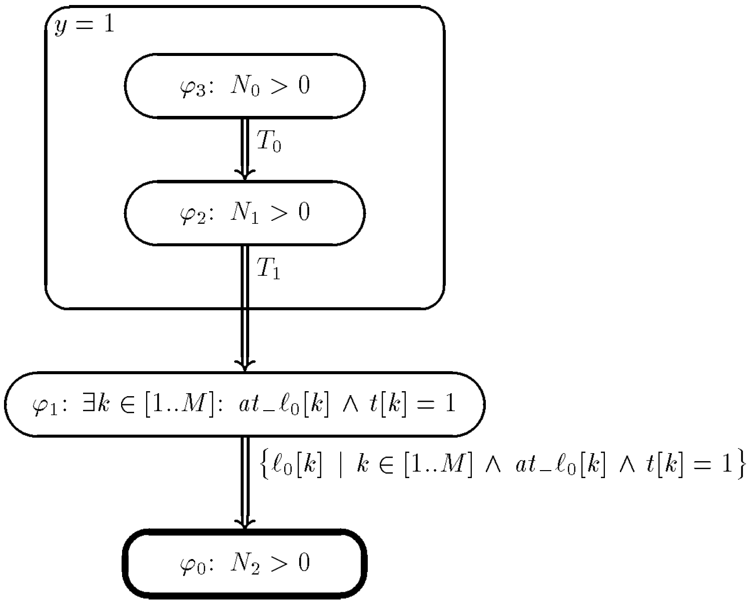

FIG. 21. Verification diagram for program RACE.

Let ℓ_r be (the label of) a statement within a process $P[k]$, $k \in [1..M]$. We introduce the following notations:

$$T_r^{t=1}\colon\quad \Big\{\ell_r[k] \ \mid\ k \in [1..M] \ \wedge\ at\_\ell_r[k] \ \wedge\ t[k] = 1\Big\}$$

Thus, $T_r^{t=1}$ denotes in a given state s the set of all transitions $\ell_r[k]$, $k \in [1..M]$, such that $at\_\ell_r[k] \wedge t[k] = 1$ holds at s. These notations often simplify the identification of state-dependent helpful sets.

In the example of program RACE, the helpful set h_1 can be replaced by $T_0^{t=1}$, and assertion φ_1 can be written as $T_0^{t=1} \neq \emptyset$. Consequently, we can rewrite the verification diagram of Fig. 21 as the verification diagram of Fig. 22.

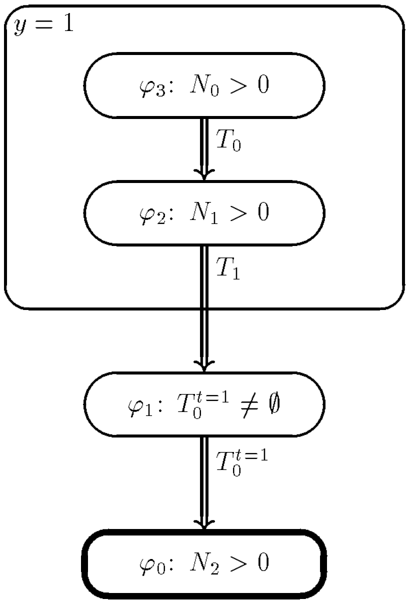

FIG. 22. Simplified verification diagram for program RACE.

7 Coordination by Add-and-store

As our final set of examples, illustrating the techniques for proving response properties of parameterized programs, we present a family of algorithms based on a special synchronization instruction.

Some computers have special "add-and-store" instructions that fetch a value from a location in memory, add to it a number and store the result in the same location, all within one execution cycle. If several processors are connected to a common shared memory then, while one of them performs such an add-and-store instruction, no other processor can interfere with the value being fetched and stored at this location. This provides a natural implementation of the atomic instruction $y := y + e$ for a shared variable y and a *local expression* e. A local expression is an expression that depends only on variables local to the process. Usually, the value which is stored is also retained in an internal register. This enables a single transition implementation of the multiple assignment statement

$$(t,\, y) \;:=\; (y+e,\, y+e),$$

where y is a shared variable, e is a local expression, and t is a variable local to the process that is executing this instruction.

To emphasize the special role of this statement, it is written in the form

$$t \;:=\; y \;:=\; y+e,$$

in all the programs in this section.

Add-and-store statements of the described form have often been suggested as means for implementing semaphores. As we will see below, they are not as strong as semaphores, since they lack the compassion property associated with semaphores. That is, the transitions associated with add-and-store statements are not compassionate while semaphore transitions are.

We present two programs that coordinate entry to critical sections by means of add-and-store statements and investigate their response properties. Since these statements are not compassionate, they cannot directly ensure individual accessibility of each process to its critical section. Instead, they can only ensure the weaker property of communal accessibility by which, if some process wishes to enter the critical section, some process (not necessarily the same) will eventually enter its critical section. However, as will be shown, communal accessibility can be used to program more involved algorithms that ensure individual accessibility.

7.1 Notations and Inferences

To facilitate proofs of the programs in this section, we introduce some special notations that rely on the special structure of the considered programs. All programs considered here consist of a parallel composition of M processes $P[1], \ldots, P[M]$, where M is an input to the program. Each process has a local variable t. Let ℓ_a and ℓ_b, $a \leq b$, be two locations in the program. We define the following notations:

$$L^{t\sim c}_{a..b}\colon \quad \left\{i \in [1..M] \;\mid\; at\_\ell_{a..b}[i] \;\wedge\; t[i] \sim c\right\},$$

$$N^{t\sim c}_{a..b}\colon \quad \left|L^{t\sim c}_{a..b}\right|$$

where $\sim$ is one of the six binary relations $\{<, \leq, =, \neq, \geq, >\}$.

Thus, $L^{t>0}_{3..4}$ denotes the set of all process indices $i \in [1..M]$, such that $P[i]$ is currently at ℓ_3 or at ℓ_4 and $t[i] > 0$, and $L^{t>c}_{3..4}$ denotes the total number of such processes.

For the case that $b = a$, we write $L^{t\sim c}_{a..b}$ and $N^{t\sim c}_{a..b}$ simply as $L^{t\sim c}_{a}$ and $N^{t\sim c}_{a}$. For the case that $b = a+1$, we write $L^{t\sim c}_{a..b}$ and $N^{t\sim c}_{a..b}$ simply as $L^{t\sim c}_{a,b}$ and $N^{t\sim c}_{a,b}$.

As in the previous section, we define

$$T^{t\sim c}_{a}\colon \quad \left\{\ell_r[k] \;\mid\; k \in [1..M] \;\wedge\; at\_\ell_a[k] \;\wedge\; t[k] \sim c\right\}.$$

There is an obvious connection between assertions referring to expressions such as $N^{t\sim c}_{a..b}$ and assertions that use existential quantification over

$$
\begin{array}{l}
\textbf{in} \quad M\colon \textbf{integer where } M > 0 \\
\textbf{local } y\ \colon \textbf{integer where } y = 1 \\
\\
\overset{M}{\underset{i=1}{\|}}\ P[i] :: \left[\begin{array}{l}
\textbf{local } t\colon\ \textbf{integer} \\
\ell_0\colon\ \textbf{loop forever do} \\
\qquad \left[\begin{array}{l}
\ell_1\colon\ \textbf{noncritical} \\
\ell_2\colon\ t := -1 \\
\ell_3\colon\ \textbf{while } t < 0 \textbf{ do} \\
\qquad \left[\begin{array}{l}
\ell_4\colon\ \textbf{await } y > 0 \\
\ell_5\colon\ t := y := y - 1 \\
\ell_6\colon\ \textbf{if } t = 0 \textbf{ then} \\
\qquad \ell_7\colon\ \textbf{critical} \\
\ell_8\colon\ y := y + 1
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}
$$

FIG. 23. Program MUX-AST (mutual exclusion by add-and-store).

the index i. For example, the following equivalence is a direct consequence of the definitions:

$$N^{t>0}_{a..b} > 0 \quad\leftrightarrow\quad \exists i\colon\ \big(at\_\ell_{a..b}[i]\ \wedge\ t[i] > 0\big).$$

All quantified variables in this section are rigid variables denoting process indices which range over $[1..M]$. We therefore omit their range specifications from the formulas.

7.2 Mutual Exclusion by Add-and-store

We first consider program MUX-AST presented in Fig. 23, which implements mutual exclusion by add-and-store instructions.

The program consists of $M > 0$ processes that coordinate their entry to their critical sections via the shared variable y. In addition, each process $P[i]$ has a local variable t which is used in the program. The program for each process is an endless loop, in which the process alternates between the execution of its noncritical section at ℓ_1 and the attempt to enter its critical section at ℓ_7.

The process may elect to stay forever in its noncritical section.

If the process exits its noncritical section, the protocol for trying to enter the critical section starts at ℓ_2, where the variable t is set to -1. Then, the process enters a loop at ℓ_3 which is terminated only when t becomes nonnegative. A nonnegative value of t (actually 0) signals that the process managed to enter its critical section, and may therefore return to ℓ_1.

Within the loop, the process waits first at ℓ_4 for y to become positive. When the process detects a positive y it performs an add-and-store statement at ℓ_5 with y and the constant -1, retaining the value stored in y also in t.

One may wonder why we need further checks after having detected a positive y at ℓ_4. The answer is that two processes $P[i]$ and $P[j]$ may detect a positive y at ℓ_4 and move, one after the other, to ℓ_5. Assume $P[i]$ to be the first to perform ℓ_5. It will lower y to 0 and obtain a $t[i] = 0$. The second process $P[j]$, performing ℓ_5 one step later, will get to $y = t[j] = -1$. Therefore, it is necessary for a process to check its own t after the addition to see whether it was the first to perform ℓ_5.

Indeed, at ℓ_6 process $P[i]$ checks whether it was the lucky one. On finding $t[i] = 0$, $P[i]$ concludes that it was the first to perform ℓ_5 and proceeds to enter the critical section at ℓ_7. On exit it increments y, offsetting the subtraction at ℓ_5. Since $t[i] = 0$ in this case, the loop of ℓ_3 terminates, and lucky process $P[i]$ returns to ℓ_1.

If the process finds a negative t, it concludes that it is not its turn to enter, and proceeds to ℓ_8. Here it increments y to offset the subtraction at ℓ_5. Since $t < 0$, the loop of ℓ_3 does not terminate, and the process returns to ℓ_4 to try its luck once more.

Proving Safety Properties: Mutual Exclusion

As a first step in the verification of program MUX-AST, let us prove the safety property of mutual exclusion. This property states that at most one process may be at ℓ_7, and is expressible by:

$$N_7 \ \leq \ 1.$$

We prove several invariants that lead to the desired conclusion.

The first invariant is given by

$$at\_\ell_7[i] \quad \rightarrow \quad t[i] = 0.$$

Observe that the assertions refer to the instance of variable t that belongs to process $P[i]$ as $t[i]$. This invariant can be verified by checking the relevant transitions. They are $\ell_6[i]$ which proceeds to ℓ_7 only if $t[i] = 0$, and transitions $\ell_2[i]$ and $\ell_5[i]$ which modify $t[i]$ but do not end up at $\ell_7[i]$.

We can obviously rewrite this invariant as

$$at\_\ell_7[i] \quad \rightarrow \quad at\_\ell_{6..8}[i] \ \wedge \ t[i] = 0,$$

due to the trivial observation that $at\_\ell_7[i]$ implies $at\_\ell_{6..8}[i]$.

Another way of expressing this invariant is by the inequality

$$I_1\colon \quad N_7 \ \leq \ N_{6..8}^{t=0}.$$

This inequality states that the number of processes $P[i]$ such that $at\_\ell_7[i]$ is true is smaller or equal to the number of processes $P[i]$ such that $at\_\ell_{6..8}[i] \wedge t[i] = 0$ is true.

A second invariant is given by

$$I_2\colon \quad N_{6..8} + y \ = \ 1.$$

This is easily verifiable by observing that initially $N_{6..8} = 0$ and $y = 1$. Then we check the relevant transitions and find that $\ell_5[i]$ decrements y by 1 but increments $N_{6..8}$ by 1. The latter is because process $P[i]$ has just joined the set $L_{6..8}$ by entering ℓ_6. Similarly, transition ℓ_8 increments y by 1 but decrements $N_{6..8}$ by 1. Thus, both preserve the sum $N_{6..8} + y$.

From this invariant it follows that $y \leq 1$ is also invariant.

The last invariant we consider is:

$$I_3: \quad N_{6..8}^{t=0} \ \leq\ 1.$$

The only transition which may endanger the validity of this invariant is $\ell_5[i]$, and then only if $N_{6..8}^{t=0} = 1$ before the transition. However, in this case $N_{6..8} > 0$ which, in view of I_2, implies $y \leq 0$. Since transition $\ell_5[i]$ leads to $t'[i] = y - 1$, it follows that $t'[i] < 0$. Therefore, i does not satisfy $at'\_\ell_{6..8}[i] \wedge t'[i] = 0$ and will not join $L_{6..8}^{t=0}$ as a result of taking $\ell_5[i]$.

This leads to $\left(N_{6..8}^{t=0}\right)' = 1$, which preserves assertion I_3.

From I_1 and I_3 we conclude

$$N_7 \ \leq\ 1.$$

Proving Response Property: Communal Accessibility

As we have already commented, the add-and-store instructions are not strong enough to guarantee individual accessibility, i.e., that *each* process wishing to enter its critical section will eventually do so. Instead, we prove the weaker property of *communal accessibility*. This can be expressed by:

$$N_2 > 0 \quad \Rightarrow \quad \Diamond(N_7 > 0).$$

Let us prove this property.

Consider the progress of a process from ℓ_2 to ℓ_7. It can certainly advance unhindered until it reaches ℓ_4. At ℓ_4 it has to wait until y becomes positive. Due to I_2, y becomes positive only when the range $\ell_{6..8}$ is evacuated of all processes currently residing there. When y becomes positive, one or more processes will reach ℓ_5. The first process to execute ℓ_5 while y is positive will get a zero value of t and will eventually get to ℓ_7.

Thus, we can partition the description of progress into phases, identified as follows:

1. Some process gets to ℓ_4.
2. While some process is at ℓ_4 and $y \leq 0$, the range $\ell_{6..8}$ is evacuated until y becomes 1.
3. Some process executes ℓ_5 while y is still 1, gets a zero t, and proceeds to ℓ_7.

Consequently, we prove these stages in the progress towards entry to the critical section as separate lemmas.

Lemma A $\quad N_2 > 0 \quad \Rightarrow \quad \Diamond(N_4 > 0)$

In Fig. 24 we present a verification diagram proof of this response formula.

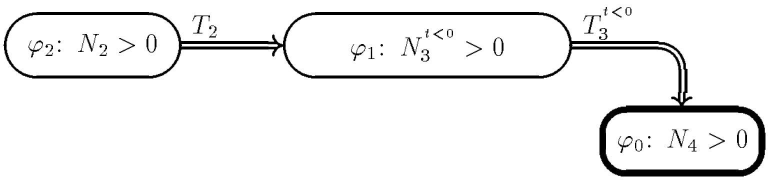

FIG. 24. A verification diagram for Lemma A.

Once we know that some process is at ℓ_4, there are two cases to consider, depending on whether y is positive. Consider first the more difficult one, in which $y \leq 0$.

Lemma B $\quad N_4 > 0 \ \wedge\ y \leq 0 \quad \Rightarrow \quad \diamondsuit(N_4 > 0 \ \wedge\ y = 1)$

We prove this lemma by rule JH-WELL. The proof is presented by the verification diagram of Fig. 25.

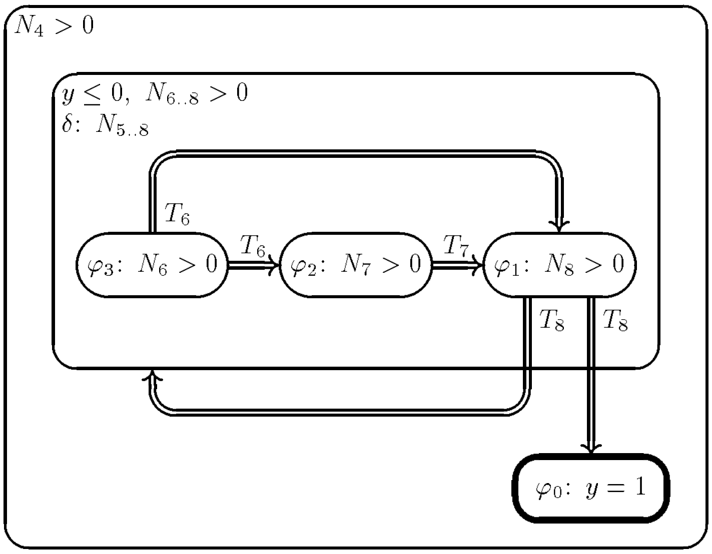

FIG. 25. A verification diagram for Lemma B.

Note that any transition that causes $N_{6..8}$ to become 0 leads, according to I_2, to $y = 1$.

The reason we use the ranking $N_{5..8}$ instead of $N_{6..8}$, which is mentioned in I_2, is that $N_{6..8}$ can increase by execution of ℓ_5. The rank $N_{5..8}$ cannot increase while $y \leq 0$, because no process can pass ℓ_4 if $y \leq 0$.

Lemma C $\quad N_4 > 0 \;\wedge\; y = 1 \quad\Rightarrow\quad \diamondsuit(N_7 > 0)$

This lemma is also proven by rule JH-WELL, tracing the progress from ℓ_4 to ℓ_7. The proof is presented in the verification diagram of Fig. 26.

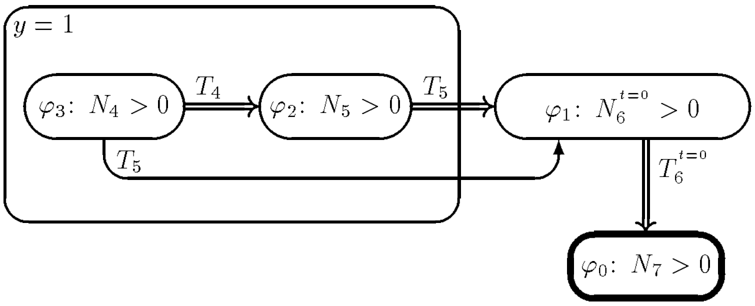

FIG. 26. A verification diagram for Lemma C.

The three lemmas can be combined into a single verification diagram, presented in Fig. 27. Note that when N_4 becomes positive, there are two possibilities: either $y \le 0$, or $y > 0$ which by I_2: $N_{6..8} + y = 1$ implies $y = 1$.

It is possible to omit most of the internal details of this diagram, and retain just the top-level structure, identifying the lemmas which lead from one phase to the next. This leads to the diagram of Fig. 28.

This schematic diagram identifies the partition of the proof into three lemmas. Each lemma is represented as a box, with double-edge exits that lead to subsequent boxes. The box represents the response property, guaranteeing that eventually the computation will exit to one of the successors of this box along one of the exits labeled by the lemma's name.

7.3 From Communal to Individual Accessibility

Program MUX-AST for mutual exclusion by add-and-store instructions does not fully satisfy our expectations for a satisfactory solution to the mutual exclusion problem. It falls short by guaranteeing communal accessibility rather than individual accessibility. However, once some process is admitted to the critical section we can appoint it as an arbitrator for the next round. This is because while being in the critical section it can perform activities such as determining which process will be the next to be admitted without having to worry about possible interference from other processes.

In Fig. 29 we present program IMUX-AST, which implements individual accessibility. This program extends program MUX-AST in several ways. It contains an additional variable *next* ranging over $1..M$, and whose value represents the index of a process that has been given high priority in the

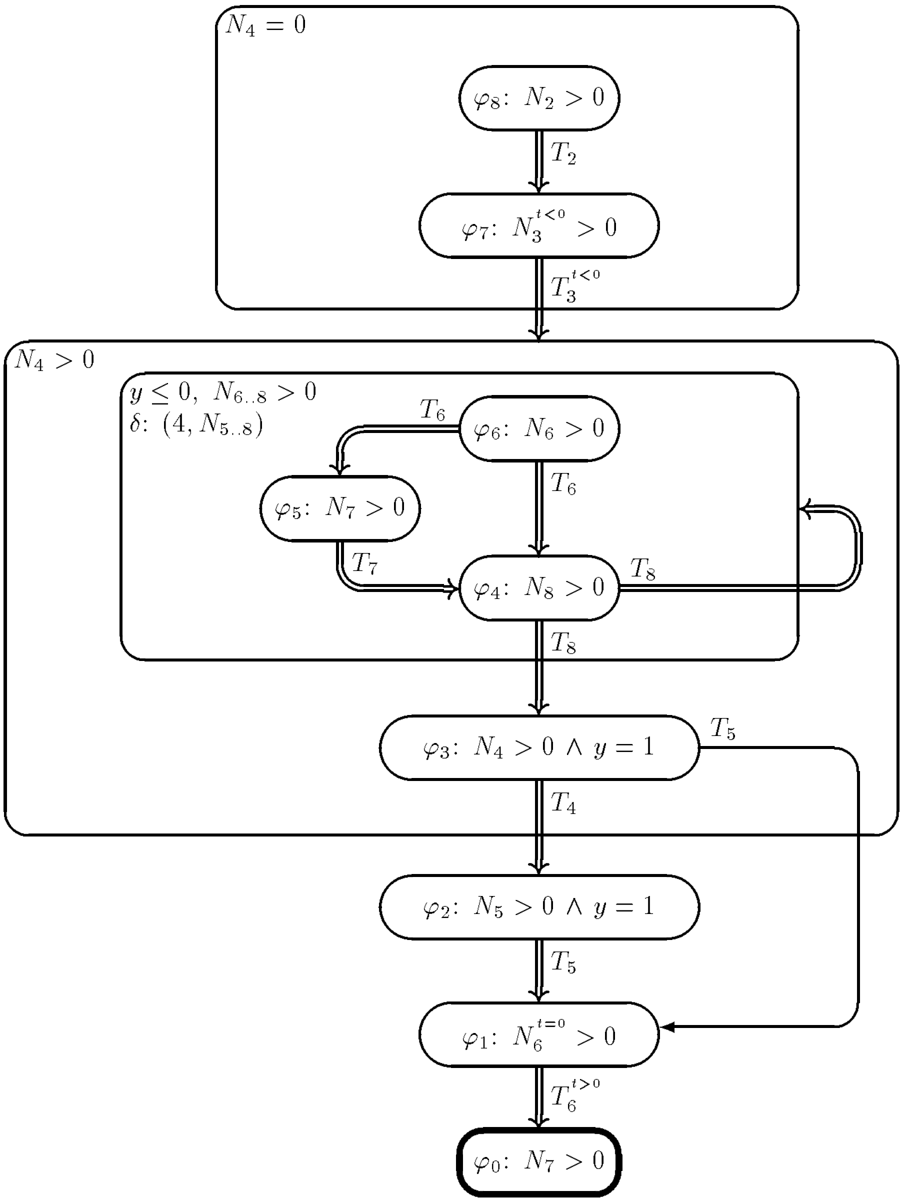

FIG. 27. A verification diagram for $N_2 > 0 \Rightarrow \diamondsuit(N_7 > 0)$.

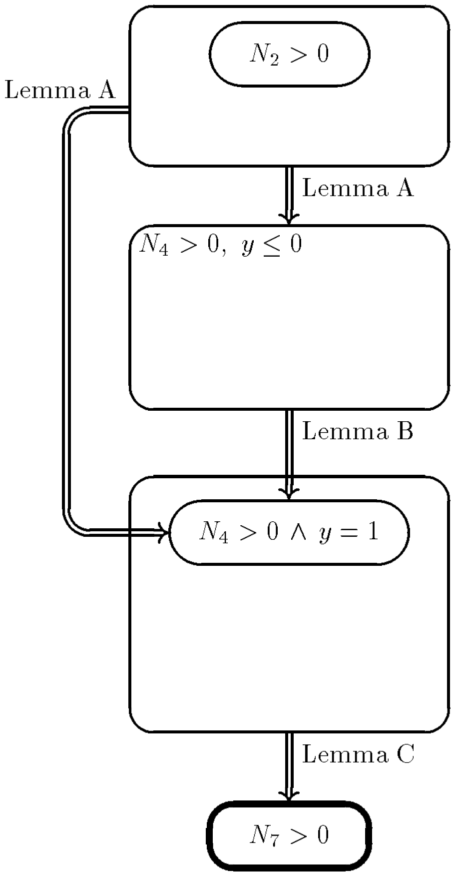

FIG. 28. The top-level structure of the proof.

last arbitration round. The local variable t of MUX-AST is transformed in IMUX-AST into an array $t[1..M]$ that can be inspected by each of the processes. Initially $next = 0$.

The protocol proceeds very much as before, except that at ℓ_4 we introduced a new gate. The function of the gate at ℓ_4 is to hold all processes except the one that has been given high priority. Note that if no process has been given priority, $next = 0$ and all processes can pass the gate at ℓ_4. From there on, the protocol proceeds as before, except that there is an additional code performing an arbitration round at $\ell_{9..15}$. Since this code follows immediately after the critical section it is guaranteed to be performed in exclusion, by a single process $P[i]$.

The first question asked at ℓ_9 is whether a new round of arbitration

$$
\begin{array}{l}
\textbf{in} \quad M \quad\quad\quad : \textbf{integer where } M = 0 \\
\textbf{local } j,\ y,\ \mathit{next}: \textbf{integer where } y = 1,\ \mathit{next} = 0 \\
\quad\quad t \quad\quad\quad : \textbf{array } [1..M] \textbf{ of integer where } t = 0
\end{array}
$$

$$
\overset{M}{\underset{i=1}{\Big\|}} P[i] :: \left[\begin{array}{l}
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\text{: } \textbf{noncritical} \\
\ell_2\text{: } t[i] := -1 \\
\ell_3\text{: } \textbf{while } t[i] < 0 \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_4\text{: } \textbf{await } \mathit{next} = 0 \vee \mathit{next} = i \\
\ell_5\text{: } \textbf{await } y > 0 \\
\ell_6\text{: } t[i] := y := y - 1 \\
\ell_7\text{: } \textbf{if } t[i] = 0 \textbf{ then} \\
\quad \left[\begin{array}{l}
\ell_8\text{: } \textbf{critical} \\
\ell_9\text{: } \textbf{if } \mathit{next} = 0 \ \vee \ \mathit{next} = i \textbf{ then} \\
\quad \left[\begin{array}{l}
\ell_{10}\text{: } \mathit{next} := 0 \\
\ell_{11}\text{: } j := (i \ \mathit{mod} \ M) + 1 \\
\ell_{12}\text{: } \textbf{while } \mathit{next} = 0 \ \wedge \ j \neq i \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_{13}\text{: } \textbf{if } t[j] < 0 \textbf{ then} \\
\quad\quad \ell_{14}\text{: } \mathit{next} := j \\
\ell_{15}\text{: } j := (j \ \mathit{mod} \ M) + 1
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\ell_{16}\text{: } y := y + 1
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

FIG. 29. Program IMUX-AST — guaranteeing individual accessibility.

is needed. A new round is needed if no process has been assigned a high priority, i.e., $next = 0$, or if the process with the high priority is $P[i]$ itself.

The other case is that some other process, $P[k]$, for $k \neq i$, has been assigned a high priority, but $P[i]$ entered the critical section ahead of $P[k]$. The algorithm is such that some overtaking is possible. In this case, $P[k]$ is still attempting entry to its critical section and we should not modify its priority. Hence, in such a case, $P[i]$ proceeds directly to ℓ_{16} and no arbitration takes place.

On entering the arbitration section, variable *next* is reset to 0 for the case that its previous value was i. Process $P[i]$ then sets variable j to the index following i in cyclic order, and enters the *while* loop at ℓ_{12}. The loop searches for the first j in cyclic order such that $t[j] < 0$. Note that $t[j] < 0$ is a reliable indicator that $P[j]$ is interested in entering the critical section but has not done so yet. If such a j is found, $P[i]$ sets *next* to j, thus declaring j to have a high priority, and then exits the loop. Another possibility is that no such j has been found, and then j will close a full cycle and return to the value i. The loop terminates in both of these cases.

The reader is invited to verify that this program guarantees individual accessibility, i.e., that it satisfies the requirement

$$at\_\ell_2[k] \quad \Rightarrow \quad \diamondsuit\, at\_\ell_7[k].$$

7.4 Readers-Writers with Add-and-store Instructions

An interesting extension of the mutual exclusion problem is the readers-writers problem. In this problem we distinguish two types of critical sections, called a *reading section* and a *writing section*. The required exclusion property is that

> while some process resides in a writing section, no other process may reside in either a reading or a writing section.

Note that this allows several processes to cohabit a reading section.

The Program

Program READ-WRITE for solving the readers-writers problem, using add-and-store instructions, is presented in Fig. 30. After the statement *non-critical* at ℓ_1, which represents the noncritical activity of the processes, each process branches nondeterministically to either the *read protocol* R, at location $\ell_{2..7}$, or to the *write protocol* W, at $\ell_{8..14}$.

In the *read protocol*, each reader tries to decrement y and achieve a nonnegative value in t (and in y). Since the initial value of y is M, up to M readers may decrement y by 1, and still obtain a nonnegative value. If a reader succeeds in obtainining a nonnegative value for t, it proceeds to the *read section* at ℓ_6. If it does not, it increments y at ℓ_7 (compensating for the subtraction at ℓ_4) and continues to loop.

The *write protocol* is similar, except that a writer subtracts M from y, attempting to obtain a nonnegative value. This is possible only if that particular writer is the only one currently subtracting from y. Even if one reader has subtracted and lowered y to $M-1$, the writer, subtracting a further M, will already obtain -1, and be barred from entering the writing section.

Another special feature of the write protocol is the wait at ℓ_{10} for y to become M. This is similar to the **await** $y > 0$ we had in the mutual exclusion program MUX-AST (Fig. 23). This is necessary, because otherwise we can construct a computation in which two writers chase one another around the ℓ_9 loop, keeping y always negative, without any of them or a reader being able to enter any of the critical sections. With the *await* gate at ℓ_{10}, this cannot happen, as we will prove below.

```
in     M: integer where M > 0
local y : integer where y = M

  M
 ||  P[i] :: [ local t: integer where t = 0
i=1            ℓ0: loop forever do
                  [ ℓ1: noncritical
                    [ R :: [ ℓ2: t := −1
                             ℓ3: while t < 0 do
                                   [ ℓ4: t := y := y − 1
                                     ℓ5: if t ≥ 0 then
                                           ℓ6: Read
                                     ℓ7: y := y + 1 ] ]
                      or
                      W :: [ ℓ8: t := −1
                             ℓ9: while t < 0 do
                                   [ ℓ10: await y = M
                                     ℓ11: t := y := y − M
                                     ℓ12: if t ≥ 0 then
                                            ℓ13: Write
                                     ℓ14: y := y + M ] ] ] ] ]
```

FIG. 30. Program READ-WRITE (readers-writers).

Proving Mutual Exclusion

The first property we prove is that of exclusion, as required by the problem. This is established by several invariants.

- *Invariant* I_1

  $$N_{5..7} \;+\; M \cdot N_{12..14} \;+\; y \;\;=\;\; M.$$

  This equality holds initially, since $y = M$ and $N_{5..7} = N_{12..14} = 0$. It is preserved by transitions, such as ℓ_4 which increments $N_{5..7}$ by 1 and decrements y by 1, or ℓ_{11} which increments $N_{12..14}$ by 1 and decrements y by M. Similarly, it is preserved by ℓ_7 and ℓ_{14}.

- *Invariant* I_2

  $$N^{t\geq 0}_{5..7} \;+\; M \cdot N^{t\geq 0}_{12..14} \;\;\leq\;\; M.$$

  There are two transitions that may endanger the invariance of this assertion.

  - $\ell_4[i]$ while $N^{t\geq 0}_{5..7} + M \cdot N^{t\geq 0}_{12..14} = M$

However, due to I_1,

$$\begin{aligned} y \;&=\; M \;-\; N_{5..7} \;-\; M\cdot N_{12..14} \\ &\le\; M \;-\; N^{t\geq 0}_{5..7} \;-\; M\cdot N^{t\geq 0}_{12..14} \;=\; 0. \end{aligned}$$

So the execution of $\ell_4[i]$ will produce $t'[i] < 0$, and hence $N^{t\geq 0}_{5..7}$ does not increase.

- $\ell_{11}[i]$ while $N^{t\geq 0}_{5..7} + M\cdot N^{t\geq 0}_{12..14} > 0$

This is a dangerous situation since by adding one more element to $N^{t\geq 0}_{12..14}$ the sum will increase beyond M. However due to I_1, and a calculation identical to the one before, we obtain

$$y \;\le\; M \;-\; N^{t\geq 0}_{5..7} \;-\; M\cdot N^{t\geq 0}_{12..14} \;<\; M.$$

Consequently, the execution of $\ell_{11}[i]$ produces $t'[i] < 0$, and does not increase $N^{t\geq 0}_{12..14}$.

- *Invariant* I_3

$$L_6 \;\subseteq\; L^{t\geq 0}_{5..7}.$$

This is equivalent to the implication

$$at\_\ell_6[i] \;\rightarrow\; t[i] \geq 0,$$

claimed for every $i = 1,\ldots,M$. The invariant can be verified by considering the transitions that enter ℓ_6. Since a process $P[i]$ can enter ℓ_6 only if $t[i] \geq 0$, it follows that $i \in L^{t\geq 0}_{5..7}$, after the transition.

- *Invariant* I_4

Similarly to the above invariant, we can also prove

$$L_{13} \;\subseteq\; L^{t\geq 0}_{12..14},$$

which is equivalent to

$$at\_\ell_{13}[i] \;\rightarrow\; t[i] \geq 0.$$

From I_3 and I_4 we can conclude

$$N_6 \;+\; M\cdot N_{13} \;\le\; N^{t\geq 0}_{5..7} \;+\; M\cdot N^{t\geq 0}_{12..14}.$$

Using I_2 we conclude

$$N_6 \;+\; M\cdot N_{13} \;\le\; M.$$

From this it is easy to infer

$$N_{13} > 0 \;\rightarrow\; N_6 = 0 \;\wedge\; N_{13} = 1,$$

which is precisely the exclusion property required. It states that if some process is at ℓ_{13}, then it is the only one there, and no process is at ℓ_6.

Proving Accessibility

The response property we prove for this program is again that of *communal accessibility*. Since this program has two types of critical sections, a *read* and a *write* section, communal accessibility has an even broader interpretation. It states that if some process is interested in entering *one* of the critical sections, then some process will eventually enter *one* of the critical sections. We cannot even guarantee, for example, that if some process wants to read (write) then some process will eventually read (write). This property is expressible by

$$N_2 + N_8 > 0 \quad \Rightarrow \quad \diamond(N_6 + N_{13} > 0).$$

We prove this property by two lemmas, concentrating first on the *writers*.

Lemma A $\quad N_8 > 0 \quad \Rightarrow \quad \diamond(N_6 + N_{13} > 0)$

This lemma states that if a process is interested in writing, then eventually some process will either read (visit ℓ_6) or write (visit ℓ_{13}).

The proof of the lemma is established by a sequence of simpler lemmas, identifying important intermediate stages in getting from $N_8 > 0$ to $N_6 + N_{13} > 0$. We refer the reader to Fig. 31, which presents a high-level verification diagram showing the structure of the proof.

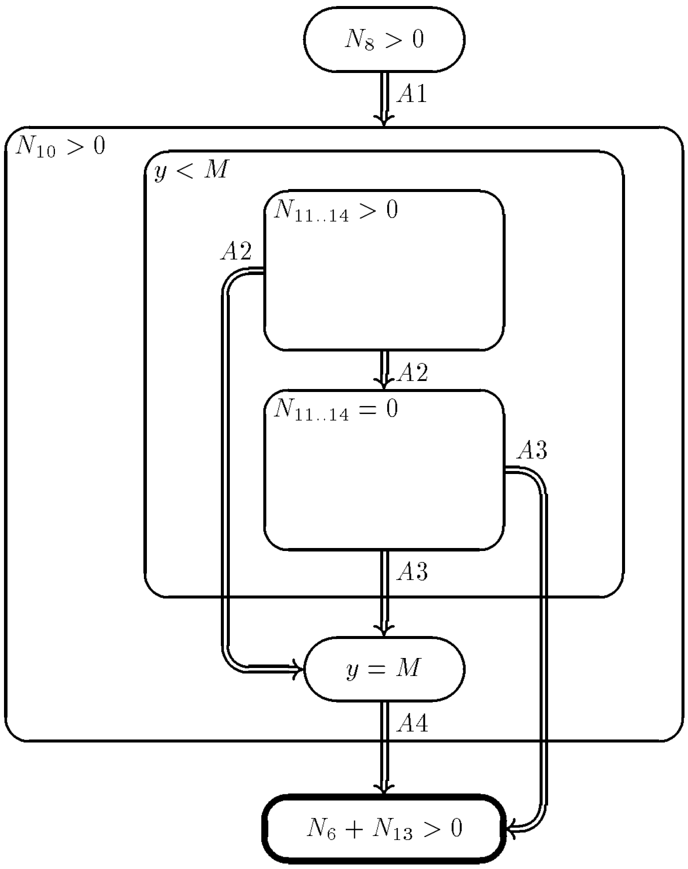

FIG. 31. Structure of the proof of Lemma A.

Lemma A1 $\quad N_8 > 0 \quad \Rightarrow \quad \diamond(N_{10} > 0)$

This lemma ensures that if some process is currently at ℓ_8 then eventually some process will arrive at ℓ_{10}.

This simple property is proven by the verification diagram of Fig. 32.

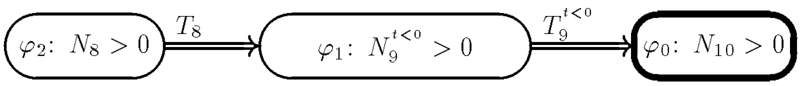

FIG. 32. A verification diagram for Lemma A1.

Arriving at a state satisfying $N_{10} > 0$, there are two cases to be considered. The easier one is that $y = M$, and Lemma A4 below shows how to get from this situation to the goal $N_6 + N_{13} > 0$. The more complicated situation is when $y < M$ (in view of the invariant I1: $y \leq M$, $y \neq M$ implies $y < M$). The two lemmas, A2 and A3, show that being at $N_{10} > 0 \wedge y < M$ we eventually get to the easier case of $N_{10} > 0 \wedge y = M$, unless we reach first a state which already satisfies the goal assertion $N_6 + N_{13} > 0$. The lemmas split the case $N_{10} > 0 \wedge y < M$ into two sub-cases, according to whether $N_{11..14}$ is positive or zero, and show that from both we eventually arrive to $N_{10} > 0 \wedge y = M$ or to $N_6 + N_{13} > 0$.

Lemma A2 $\quad N_{10} > 0 \ \wedge \ y < M \ \wedge \ N_{11..14} > 0 \quad \Rightarrow$

$$\diamond\big(N_{10} > 0 \ \wedge \ (N_{11..14} = 0 \ \vee \ y = M)\big)$$

The proof of this lemma can be based on rule JH-WELL, and is presented in the verification diagram of Fig. 33.

Note that transitions from ℓ_{14} may either retain $N_{11..14} > 0$ and $y < M$, or evacuate $L_{11..14}$ completely, or increase y to become M. It is possible for y to equal M, while some processes are still at ℓ_{11}.

Note that we also have to consider the possibility that a transition ℓ_7 increases y to become M.

In the proof we rely on the fact that while $y < M$, all transitions of the form $\ell_{10}[i]$ are disabled.

Lemma A3 $\quad N_{10} > 0 \ \wedge \ y < M \ \wedge \ N_{11..14} = 0 \quad \Rightarrow$

$$\diamond\big((N_{10} > 0 \ \wedge \ y = M) \ \vee \ N_6 + N_{13} > 0\big)$$

The proof of the lemma is presented in the verification diagram of Fig. 34.

We observe that, due to I_1, the antecedent implies that $0 < N_{4..7} < M$ and $0 < y < M$. This situation is split in the diagram into several subcases:

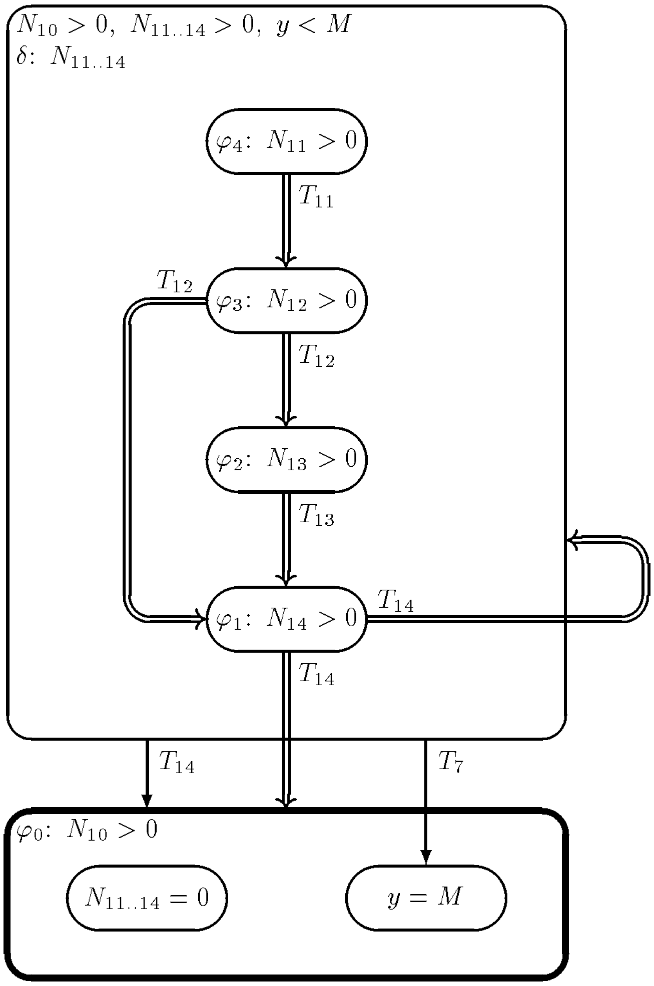

FIG. 33. A verification diagram for Lemma A2.

- $N_4 = N_6 = 0 \ \wedge \ N_5^{t<0} + N_7 > 0$ is covered by φ_3 and φ_4
- $N_4 > 0$ is covered by φ_2
- $N_5^{t\geq 0} > 0$ is covered by φ_1
- $N_6 > 0$ is covered by the disjunct $N_6 + N_{13} > 0$ of φ_0.

The sum $N_5^{t<0} + N_7$ decreases on each ℓ_7 transition. It must eventually drop to zero, ensuring $y = M$, unless N_4 becomes positive before.

Combining Lemmas A2 and A3 together, we obtain

$$N_{10} > 0 \wedge y < M \ \Rightarrow\ \Diamond\big((N_{10} > 0 \wedge y = M) \vee N_6 + N_{13} > 0\big).$$

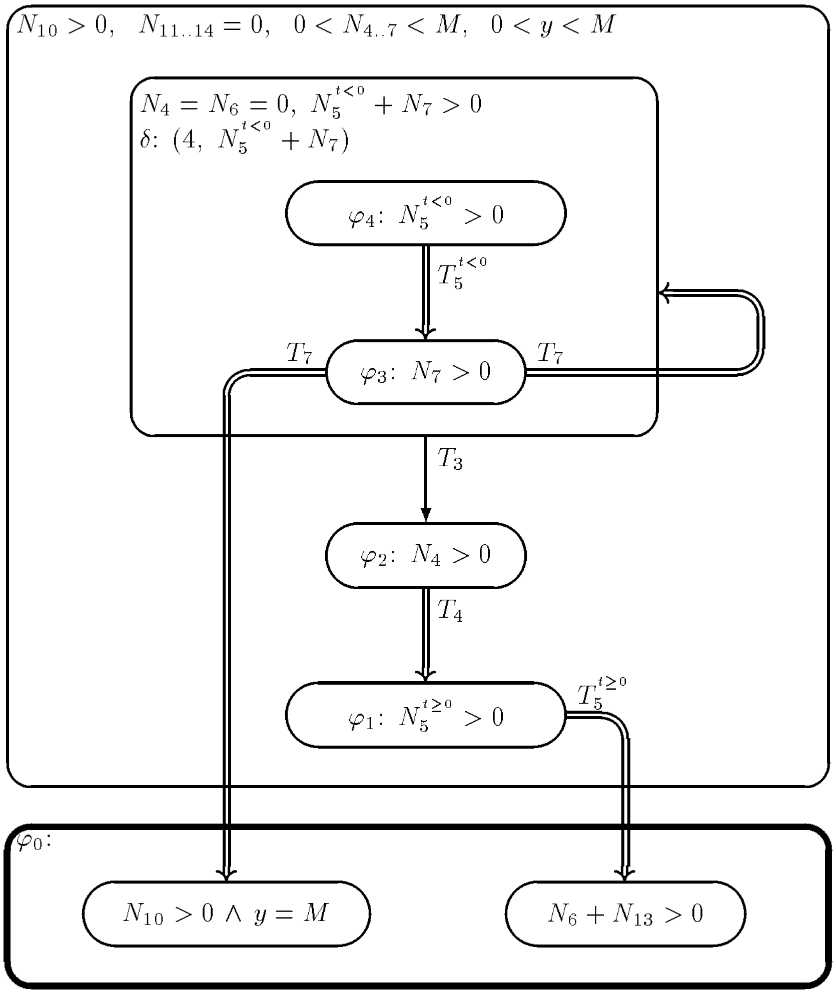

FIG. 34. A verification diagram for Lemma A3.

It remains to show that from $(N_{10} > 0) \wedge (y = M)$ we are also guaranteed to reach $N_{13} + N_6 > 0$. This is claimed by the next lemma.

Lemma A4 $\quad N_{10} > 0 \ \wedge \ y = M \quad \Rightarrow \quad \Diamond(N_6 + N_{13} > 0)$

The proof of this lemma is presented in verification diagram of Fig. 35. We split the initial situation into two cases:

- $N_{10} > 0, \ N_{11} = 0$ covered by φ_4
- $N_{11} > 0,$ covered by φ_3.

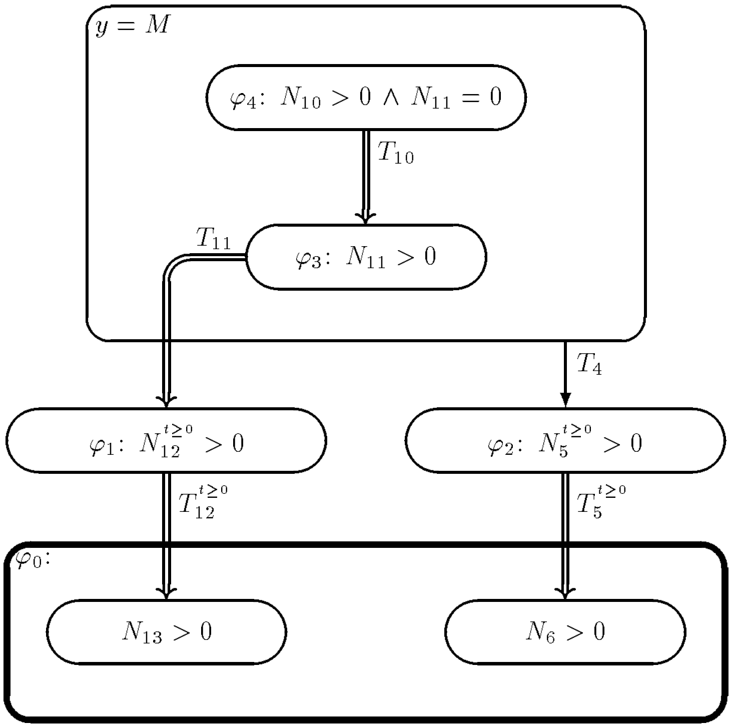

FIG. 35. A verification diagram for Lemma A4.

This concludes the proof of Lemma A, which states that if a process wishes to write then eventually some process will read or some process will write.

By combining Lemmas A1, A2, and A3, we can also obtain Corollary A.

Corollary A $\quad N_{10} > 0 \;\Rightarrow\; \diamondsuit(N_6 + N_{13} > 0)$

The second case we have to handle is the possibility that a process wishes to read. This is covered by the following lemma:

Lemma B $\quad N_2 > 0 \;\Rightarrow\; \diamondsuit(N_6 + N_{13} > 0)$

The proof of Lemma B is also split into several intermediate stages. In Fig. 36 we present these stages and the lemmas that lead from one stage to the next.

Lemma B1 $\quad N_2 > 0 \;\Rightarrow\; \diamondsuit\left(N_{3..7}^{t<0} > 0\right)$

This simple lemma follows a process, that wishes to read, from location ℓ_2 to location ℓ_3, where it arrives with $t[i] < 0$. The lemma is so simple that we skip its proof.

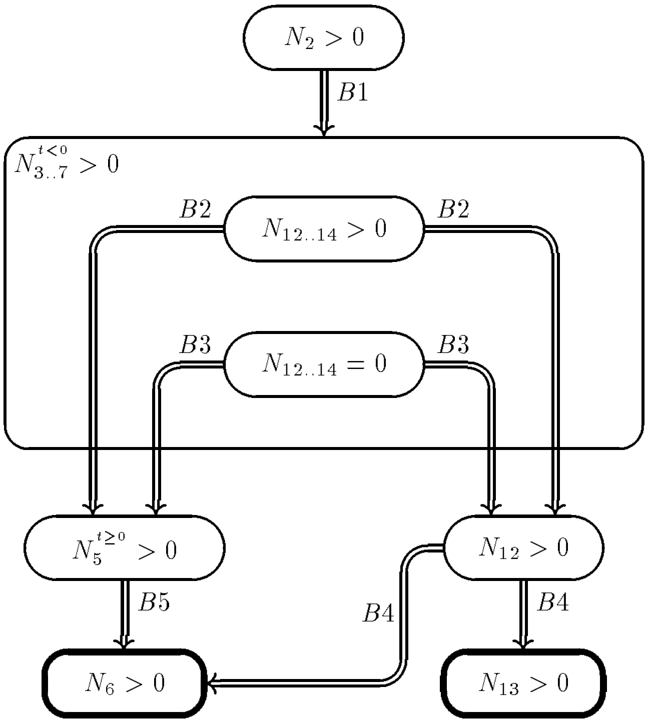

FIG. 36. Structure of the proof of Lemma B.

Arriving at ℓ_3, it is important to distinguish between the case that $y > 0$ and the case $y \leq 0$. Due to the invariant I_1, if $y \leq 0$ then either $N_{12..14} > 0$ or $N_{5..7} = M$. Since $N_3 > 0$, it is impossible for all processes to be at $\ell_{5..7}$, and therefore we conclude that $N_{12..14} > 0$. The next lemma shows that this case must develop to the simpler case in which $N_{12..14} = 0$. However, we cannot hope that while this happens, the process which is currently at ℓ_3 will stand still. What can it do? It can loop around $\ell_{3..7}$ and perhaps even get to ℓ_5 with a nonnegative $t[i]$. If this happens then we are close to our goal $N_6 > 0$. If it does not happen, we should at least make sure that this process does not escape the loop $\ell_{3..7}$ without visiting ℓ_6 first. To contain some processes in the loop we use the set $L^{t<0}_{3..7}$, observing that the only way a process can leave this set is by moving to $L^{t\geq 0}_5$.

Consequently, we have the following:

Lemma B2

$$N^{t<0}_{3..7} > 0 \;\wedge\; N_{12..14} > 0 \quad\Rightarrow\quad \Diamond \begin{bmatrix} \left(N^{t<0}_{3..7} > 0 \;\wedge\; N_{12..14} = 0\right) \vee \\ N^{t\geq 0}_5 > 0 \;\vee\; N_{12} > 0 \end{bmatrix}$$

The proof of this lemma is presented in the verification diagram of Fig. 37. It uses the ranking function $N_{12..14}$, which decreases on each ℓ_{14} transition. By I1, y is smaller than M, as long as $N_{12..14} > 0$.

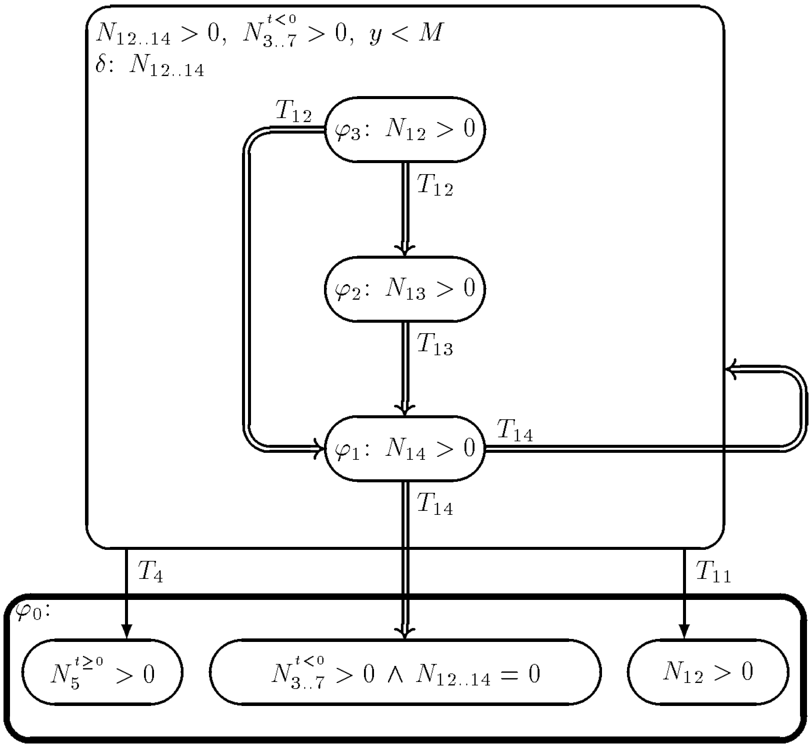

FIG. 37. A verification diagram for Lemma B2.

In a more general situation we would have to worry about the possibility of $N_{12..14}$ increasing due to an ℓ_{11} transition. Here, however, any ℓ_{11} transition makes L_{12} nonempty, which is one of the goals of this lemma.

Similarly, the only transition that can violate $N^{t<0}_{3..7} > 0$ is ℓ_4, taken in a state with a positive y. This, however, makes $L^{t\geq 0}_5$ nonempty, which is another goal.

The next lemma considers a situation in which there is still some unsatisfied process (i.e., $t[i] < 0$) in the $\ell_{3..7}$ loop, while $N_{12..14} = 0$. It shows that eventually one of the sets $L^{t\geq 0}_5$ or L_{12} must become nonempty.

Lemma B3 $\quad N^{t<0}_{3..7} > 0 \ \wedge\ N_{12..14} = 0 \quad \Rightarrow \quad \Diamond\Big(N^{t\geq 0}_5 > 0 \ \vee\ N_{12} > 0\Big)$

The lemma is proved in the verification diagram of Fig. 38. Note that while being at ℓ_6, $t[i] \geq 0$, so that $L_6^{t<0}$ is always empty. Therefore, having $N_{3..7}^{t<0} > 0$ implies that we must have a nonempty $L_{3..5,7}$. If L_4 is nonempty, then due to $N_{12..14} = 0$ and to I1, we are guaranteed to have a positive y. Therefore, any ℓ_4 transition from such a state leads to a nonempty $L_5^{t\geq 0}$.

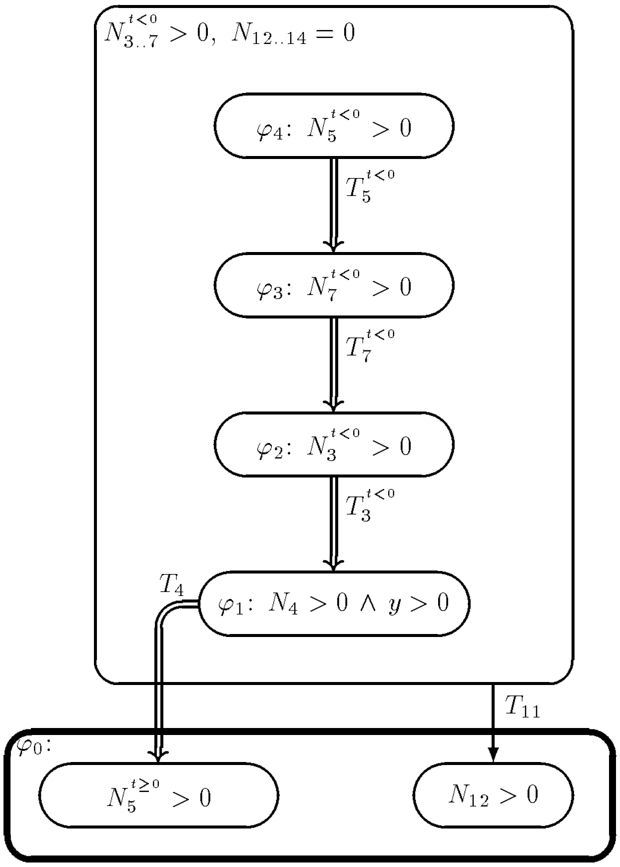

FIG. 38. A verification diagram for Lemma B3.

Lemma B4 $\quad N_{12} > 0 \quad \Rightarrow \quad \Diamond(N_6 + N_{13} > 0)$

This lemma starts from a situation in which we identify a writer at ℓ_{12}. If $t[i] \geq 0$ for this process, then we can easily prove that eventually $N_{13} > 0$. Otherwise, $t[i] < 0$, and we can easily trace the progress of this process from ℓ_{12}, through ℓ_{14}, ℓ_9 until it reaches ℓ_{10}. We now use Lemma A to show that if L_{10} is nonempty, then eventually some process will reach ℓ_6 or ℓ_{13}.

Lemma B5 $N_5^{t\geq 0} > 0 \;\Rightarrow\; \diamondsuit(N_6 > 0)$

This lemma requires one helpful step, performed by any of the processes in $L_5^{t\geq 0}$, to achieve a nonempty L_6.

Acknowledgement

We gratefuly acknolwedge the numerous critical comments and helpful suggestions made by A. Anuchitanukul, N. Bjørner, A. Kapur, and T. Uribe.

Bibliography

1. J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
2. E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 8:244–263, 1986.
3. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274, 1987.
4. Z. Manna and A. Pnueli. An exercise in the verification of multi-process programs. In W. Feijen, A. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is Our Business*, pages 289–301. Springer-Verlag, 1990.
5. Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130, 1991.
6. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
7. Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. Rashid, editor, *Carnegie Mellon Computer Science: A 25-year Commemorative*, pages 121–156. ACM Press and Addison–Wesley, 1991.
8. Z. Manna and A. Pnueli. Temporal verification diagrams. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lect. Notes in Comp. Sci.*, pages 726–765. Springer-Verlag, 1994.
9. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.