Temporal Verification Diagrams *

Zohar Manna[1] and Amir Pnueli[2]

[1] Department of Computer Science, Stanford University
Stanford, CA 94305
e-mail: manna@cs.stanford.edu

[2] Department of Applied Mathematics and Computer Science
Weizmann Institute, Rehovot, Israel
e-mail: amir@wisdom.weizmann.ac.il

Abstract. Most formal approaches to the verification of temporal properties of reactive programs infer temporal conclusions from verification conditions that are state formulas, i.e., contain no temporal operators. These proofs can often be effectively presented by the use of *verification diagrams*. In this paper, we present a self-contained presentation of verification diagrams for proving various temporal properties.
Beginning with safety properties, we present WAIT-FOR and INVARIANCE diagrams for proving wait-for (precedence) and invariance formulas. Proceeding to liveness properties, we present verification diagrams for response properties that require a bounded number of helpful steps (CHAIN diagrams) and response properties that require an unbounded number of helpful steps (RANK diagrams).
Additional types of diagrams are proposed for handling response properties for parameterized programs (e.g., P-RANK diagrams) and response properties that rely on the full spectrum of fairness requirements, including compassionate helpful transitions (e.g., F-CHAIN diagrams).

Keywords: verification diagrams, temporal logic, reactive systems, formal verification.

Table of Contents

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

1 Introduction

Temporal logic has proven to be a powerful and convenient language for specifying properties of reactive programs. Most of the interesting properties of reactive programs, such as safety, invariance, precedence, response, infinite recurrence, and the various flavors of fairness, can be expressed by natural and succinct temporal formulas.

If one adopts temporal logic as the specification language of choice, the next relevant question is that of verification. Given a temporal specification φ of a contemplated program and a proposed implementation P, how does one prove or disprove that the proposed program P meets its specification, i.e., that all computations of P satisfy the temporal formula φ?

Several approaches to the formal verification of temporal properties of reactive programs have been proposed over the years, for example, [HO83], [NGO85], [OL82], [AS89], [Lam91]. Our own efforts evolved from [MP83] through [MP84] to a proof system that is summarized in [MP91a] and in a forthcoming book [MP94].

Verification of temporal properties of reactive programs is not significantly more complex than verification of state properties, such as partial correctness and termination of sequential programs. The two key facts which justify this observation are:

- Most of the temporal properties of interest can be expressed by formulas characterizable by a small number of "canonical" forms.
- For each of these canonical forms, there exists a rule which is complete for verifying properties expressible by formulas of this form. Furthermore, the premises of these rules are all first-order and contain no temporal formulas.

Thus, if one uses only properties expressible by these few canonical formulas, which seems the practical thing to do, then almost no temporal reasoning is required and most of the verification effort is spent in proving first-order formulas, a type of reasoning which is much more familiar than temporal reasoning.

This argument was already advanced in [MP91a], where we identified the classes of properties expressible by six canonical formulas and presented for each of these classes a rule with first-order premises.

This paper is a further step in our continuing effort to popularize the temporal methodology, making verification of temporal properties of reactive programs more feasible and manageable.

The paper suggests eliminating proof rules altogether and replacing them by *verification diagrams*, which are a visual representation of the basic constituents required in a proof of a temporal property. Each diagram informally identifies the possible evolution of a computation of the program by the assertions its states satisfy. In the case of a progress property, this evolution progresses towards a given goal, and the assertions traversed on the way identify major intermediate stages in this evolution.

However, each diagram has also a formal meaning, which is a set of *verification conditions* spawned by the diagram. These verification conditions are first-order formulas (with one exception) and their validity establishes the validity of the temporal formula supported by the diagram.

Corresponding to the mentioned classes of properties, we present a verification diagram for the more important classes. It is not difficult to derive from them verification diagrams for the remaining classes.

We have been using verification diagrams for quite some time. However, in all previous instances, the diagrams served as an illustration device, secondary to the formal rule. This is the first time that we venture to propose a completely rule-free proof system, in which the main formal devices are the diagrams themselves.

There is some similarity between the motivation for this paper and that of [AS89]. However, [AS89] does not use temporal logic whatsoever and uses automata both as a specification language and as generators of verification conditions.

Another connection with previous work is with the lattice diagrams of [OL82]. However, similar to our own previous use of verification diagrams, the proof lattices of [OL82] serve only as an illustrating device for a rule, where the rule has the main formal import.

The techniques presented here can be extended to verify properties of real-time and hybrid systems [MP93].

2 Preliminaries

The qualitative model of reactive systems uses an abstract notion of time, based on the ordering of events during an observed computation. This is the main model used, for example, in [MP91b].

2.1 Computational Model: Fair Transition System

The computational model for reactive systems is that of *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. We assume each variable to be associated with a domain over which it ranges.
  We define a *state* s to be a type-consistent interpretation of V, assigning to each variable $u \in V$ a value $s[u]$ over the variable's domain. We denote by Σ the set of all states.
- 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 *enabled* on s iff $\tau(s) \neq \emptyset$. Otherwise τ is *disabled* on s.
  The function associated with a transition τ is represented by an assertion $\rho_\tau(V, V')$, called the *transition relation*, which relates a state $s \in \Sigma$ to its τ-successor $s' \in \tau(s)$ by referring to both unprimed and primed versions of the system variables. An unprimed version of a system variable refers to its value in s, while a primed version of the same variable refers to its value in s'. For example, the assertion $x' = x + 1$ states that the value of x in s' is greater by 1 than its value in s.
- $\mathcal{J} \subseteq \mathcal{T}$: A set of *just* transitions (also called *weakly fair* transitions). The requirement of justice for $\tau \in \mathcal{J}$ disallows a computation in which τ is continually enabled beyond a certain point but taken only finitely many times.
- $\mathcal{C} \subseteq \mathcal{T}$: A set of *compassionate* transitions (also called *strongly fair* transitions). The requirement of compassion for $\tau \in \mathcal{C}$ disallows a computation in which τ is enabled infinitely many times but taken only finitely many times.

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

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

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 $s' = s$.

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 $\mathcal{V}$-states $\sigma : s_0, s_1, s_2, ...,$ for some vocabulary $\mathcal{V}$ that contains V, satisfying the following requirements:

- *Initiation:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For each $j = 0, 1, ...,$ the state s_{j+1} is a τ-successor of the state s_j, i.e., $s_{j+1} \in \tau(s_j)$ for some $\tau \in \mathcal{T}$. In this case, we say that the transition τ is *taken* at position j in σ.
- *Justice:* For each $\tau \in \mathcal{J}$, it is not the case that τ is continually enabled beyond some position in σ but taken at only finitely many positions in σ.
- *Compassion:* For each $\tau \in \mathcal{C}$, it is not the case that τ is enabled on infinitely many states of σ but taken at only finitely many positions in σ.

2.2 A Simple Programming Language: Syntax

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

- *Assignment*: For a variable y and an expression e of compatible type,
  $y \ := \ e$
  is an *assignment* statement.
- *Await*: For a boolean expression c,
  await c
  is an *await* statement. We refer to condition c as the *guard* of the statement. Execution of **await** c changes no data variables. Its sole purpose is to wait until c becomes true, at which point it terminates.
- *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.
- *Concatenation*: For statements $S_1, \ldots, S_k$,
  $S_1; \cdots; S_k$
  is a *concatenation* statement. Its intended meaning is sequential composition. The first step in an execution of $S_1; \cdots; S_k$ is the first step in an execution of S_1. Subsequent steps continue to execute the rest of S_1, and when S_1 terminates, proceed to execute $S_2, S_3, \ldots, S_k$.
  In a program presented as a multi-line text, we often omit the separator ';' at the end of a line.
- *While*: For a boolean expression c and a statement S,
  while c **do** S

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

We introduce the notation

loop forever do S

as an abbreviation for

while T **do** S.

- *Request*: For an integer variable r,

  request r

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

  release r

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

Schematic Statements

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

- *Noncritical*: The statement

  noncritical

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

  critical

  represents the critical activity in programs for mutual exclusion, where coordination between the processes is required. It is required that this statement terminate.
- *Produce*: For an integer variable x, the statement

  produce x

  represents the production activity in producer-consumer programs. The effect of this terminating statement is to assign a nonzero value to variable x, representing the production of a value.
- *Consume*: For an integer variable y, the statement

  consume y

  represents the consumption activity in producer-consumer programs. The effect of this terminating statement is to set variable y to zero, representing the consumption of y's previous value.

It is assumed that none of these statements modify any program variables, except for the production variable x and the consumption variable y, explicitly mentioned in the corresponding statements.

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

Programs

A program P has the form

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

where $P_1 :: [\ell_1\colon S_1;\ \widehat{\ell_1} :], \ldots, P_m :: [\ell_m\colon S_m;\ \widehat{\ell_m} :]$ are *named processes*. The names of the program and of the processes are optional, and may be omitted. The *body* $[\ell_i\colon S_i;\ \widehat{\ell_i} :]$ 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 (Peterson's algorithm) Consider program PET presented in Figure 1 which implements Peterson's algorithm for mutual exclusion without semaphores. This program will serve as our running example for most of the verification diagrams we intend to present.

The basic mechanism protecting the accesses to the critical sections is provided by the boolean variables y_1 and y_2. Each process P_i, $i = 1, 2$, that is interested in entering its critical section sets its y_i variable to T. On exiting the critical section, the corresponding y_i is reset to F.

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

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

2.3 Semantics of the Programming Language

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

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

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

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

the five components of a fair transition system for a given program.

Consider a program P given by

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

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 PET are $V : \{\pi, y_1, y_2, s\}$, where y_1 and y_2 range over the booleans, s ranges over the integers, and π ranges over subsets of $\{\ell_0, \ldots, \ell_6, m_0, \ldots, m_6\}$.

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 PET is $\langle \pi : \{\ell_0, m_0\},\ y_1 : \mathrm{F},\ y_2 : \mathrm{F},\ s : 1\rangle$.

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 progrsm PET, $\pi = \{\ell_0, m_0\}$ 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

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

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 PET is given by

$$\Theta: \quad \pi = \{\ell_0, m_0\} \ \wedge\ y_1 = \mathrm{F} \ \wedge\ y_2 = \mathrm{F} \ \wedge\ s = 1.$$

Transitions

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

$$move(\ell_1, \ell_2): \quad at\_\ell_1 \ \wedge\ \pi' = (\pi - \{\ell_1\}) \cup \{\ell_2\}.$$

This formula expresses the situation that control is currently at ℓ_1 and moves in the next state to location ℓ_2.

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

$$\rho_I: \ V' = V.$$

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

- *Assignment*: Consider the statement $[\ell : y := e\,;\ \widehat{\ell} : \ldots]$, where $\widehat{\ell}$ is the location to which control moves after statement terminates. We refer to $\widehat{\ell}$ as the *post-location* of statement ℓ.

  With this statement we associate a transition τ_ℓ, with the transition relation

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

  The last conjunct claims that all data variables, excluding y, retain their values over the transition τ_ℓ.
- *Await*: With the statement $[\ell$: **await** $c\,;\ \widehat{\ell} : \ldots]$, 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 $\widehat{\ell}$.

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

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

  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 $\Big[\ell\colon \big[\textbf{while } c \textbf{ do } [\widetilde{\ell}\colon \widetilde{S}]\big];\ \widehat{\ell}:\ldots,\Big]$ we associate a transition τ_ℓ, with the transition relation

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

  According to ρ_ℓ, when c evaluates to T control moves from ℓ to $\widetilde{\ell}$, and when c evaluates to F control moves from ℓ to $\widehat{\ell}$. Note that, in this context, the post-location of $\widetilde{S}$ is ℓ. Note also that the enabling condition of τ_ℓ is at_ℓ, which does not depend on the value of c.
- *request*: With the statement $[\ell\colon \textbf{request}(r);\ \widehat{\ell}\colon\ldots]$, we associate a transition τ_ℓ, with the transition relation

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

  Thus, this statement is enabled when control is at ℓ and r is positive. When executed it decrements r by 1.
- *release*: With the statement $[\ell\colon \textbf{release}(r);\ \widehat{\ell}\colon\ldots]$, we associate a transition τ_ℓ, with the transition relation

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

  This statement increments r by 1.
- *Noncritical*: With the statement $[\ell\colon \textbf{noncritical};\ \widehat{\ell}\colon\ldots]$, 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 $[\ell\colon \textbf{critical};\ \widehat{\ell}\colon\ldots]$, 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.
- *Produce*: With the statement $[\ell\colon \textbf{produce } x;\ \widehat{\ell}\colon\ldots]$, we associate a transition τ_ℓ, with transition relation

$$\rho_\ell\colon\quad move(\ell,\widehat{\ell}) \ \wedge\ x' \neq 0 \ \wedge\ pres(Y-\{x\}).$$

  The observable action of the produce statement is to assign a nonzero value to variable x. Note that this transition is nondeterministic; that is, a state s may have more than one τ_ℓ-successors. In fact it may have infinitely many successors, one for each possible value of x'.

- *Consume*: With the statement [ℓ: **consume** y; $\widehat{\ell}$: ...], we associate a transition τ_ℓ, with transition relation
  $$\rho_\ell: \quad move(\ell, \widehat{\ell}) \ \wedge \ y' = 0 \ \wedge \ pres(Y - \{y\}).$$
  The observable action of the consume statement is to set variable y to zero.

Thus, the fair transition system corresponding to program PET has the transitions
$$\tau_I, \tau_{\ell_0}, \ldots, \tau_{\ell_6}, \tau_{m_0}, \ldots, \tau_{m_6}.$$
Transition τ_{ℓ_3}, for example, has the transition relation
$$\rho_{\ell_3}: \quad move(\ell_3, \ell_4) \ \wedge \ s' = 1 \ \wedge \ y_1' = y_1 \ \wedge \ y_2' = y_2.$$

Justice and Compassion

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

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

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

2.4 Requirement Specification Language: Temporal Logic

As a requirement specification language for reactive systems (under the qualitative model) 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:

$\bigcirc$ – Next $\qquad\qquad$ $\mathcal{U}$ – 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,
  $(\sigma, j) \vDash p \quad \Longleftrightarrow \quad s_j \vDash p$
  That is, we evaluate p locally, using the interpretation provided by s_j.
- $(\sigma, j) \vDash \neg p \quad \Longleftrightarrow \quad (\sigma, j) \nvDash p$
- $(\sigma, j) \vDash p \vee q \quad \Longleftrightarrow \quad (\sigma, j) \vDash p$ or $(\sigma, j) \vDash q$
- $(\sigma, j) \vDash \bigcirc p \quad \Longleftrightarrow \quad (\sigma, j+1) \vDash p$
- $(\sigma, j) \vDash p \,\mathcal{U}\, q \quad \Longleftrightarrow$ 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:

$\diamondsuit\, p = \mathrm{T}\, \mathcal{U}\, p$ – 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:

$$p \Rightarrow q \quad \Longleftrightarrow \quad \square(p \rightarrow q).$$

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.

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.

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

The formulas p and q are defined to be *congruent*, denoted $p \approx q$, if the formula $\square(p \leftrightarrow q)$ 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*.

In this paper, we will mainly consider safety properties expressible by the *invariance* formula $\square\,\varphi$, where φ is a state formula, and the *wait-for* formula $p \Rightarrow (\varphi\, \mathcal{W}\, q)$ for state formulas p, φ, and q. The formula $p \Rightarrow (\varphi\, \mathcal{W}\, q)$ states that, following each p-position, there is a succession of φ-positions that either extends to infinity or is terminated by a q-position.

A more elaborate safety formula that can be constructed, using the $\mathcal{W}$ operator, is a *nested wait-for* formula of the general form

$$p \;\Rightarrow\; q_m\, \mathcal{W}\, q_{m-1} \;\cdots\; q_1\, \mathcal{W}\, q_0,$$

for assertions $p, q_0, \ldots, q_m$. This parentheses-free formula should be interpreted by association to the right, as

$$p \;\Rightarrow\; (q_m\, \mathcal{W}\, (q_{m-1} \;\cdots\; (q_1\, \mathcal{W}\, q_0)\cdots)).$$

A model satisfies this formula if every p-position starts a succession of intervals, beginning with a q_m-interval, and ending in a q_1-interval, that may be terminated by a q_0-position. Each of these intervals, say the q_i-interval, may be empty or extend to infinity. In the latter case there need not be a following q_j-interval, for $j < i$, nor a terminating q_0-position.

The following is a list of several safety formulas that are valid over program PET and therefore specify properties of this program:

- $\square(s = 1 \;\vee\; s = 2)$
  This formula claims that s has the values 1 or 2 in all states appearing in computations of PET.

- $\Box(at\_\ell_{3..6} \leftrightarrow y_1)$
  This formula claims that y_1 is true if and only if control of process P_1 is at one of the locations $\ell_3, \ldots, \ell_6$. Note the use of the expression $at\_\ell_{3..6}$ as an abbreviation for the disjunction $at\_\ell_3 \vee \cdots \vee at\_\ell_6$.
- $\Box\neg(at\_\ell_5 \;\wedge\; at\_m_5)$
  This formula states the property of *mutual exclusion* according to which processes P_1 and P_2 can never execute their critical sections at the same time.
- $\underbrace{at\_\ell_4}_{p} \;\Rightarrow\; \underbrace{\neg at\_m_5}_{q_3} \;\mathcal{W}\; \underbrace{at\_m_5}_{q_2} \;\mathcal{W}\; \underbrace{\neg at\_m_5}_{q_1} \;\mathcal{W}\; \underbrace{at\_\ell_5}_{q_0}.$
  This nested wait-for formula states that any position satisfying at_ℓ_4 is followed by an interval in which P_2 is not at m_5, followed by an interval in which P_2 is at m_5, followed by an interval in which P_2 is again not at m_5, which can be terminated only by a state in which P_1 is at ℓ_5.
  Consequently, this formula states the property of 1-*bounded overtaking* which claims that, from the time P_1 is at ℓ_4, P_2 may get to its critical section at m_5 ahead of P_1 (overtaking P_1) at most once. The case of P_2 not reaching m_5 at all is obtained by taking an empty at_m_5 interval.

Response Properties

In this paper we consider response properties that can be expressed by a formula

$$p \Rightarrow \Diamond q,$$

for state formulas p and q.

For example, the response formula

$$at\_\ell_2 \Rightarrow \Diamond\, at\_\ell_5$$

states the property of *accessibility* for program PET. This property ensures that whenever process P_1 wishes to enter its critical section, as is evident by observing it at ℓ_2, it will eventually enter the critical section at ℓ_5.

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 PET, consider for example the verification condition of transition ℓ_3 with respect to assertions $p = q:\; s = 1 \;\vee\; s = 2$.

$\{s = 1 \vee s = 2\}\ \ell_3\ \{s = 1 \vee s = 2\}$.

Expanding the definition of the verification condition, this yields

$$\underbrace{\cdots \wedge\ s' = 1\ \wedge\ \cdots}_{\rho_{\ell_3}} \quad \wedge \quad \underbrace{s = 1 \vee s = 2}_{p} \quad \rightarrow \quad \underbrace{s' = 1 \vee s' = 2}_{q'}.$$

which is state valid. This shows that every ℓ_3-successor of a state satisfying $s = 1 \vee s = 2$ also satisfies $s = 1 \vee s = 2$.

3 Verification Diagrams for Safety Properties

In proofs of properties of reactive systems, it is typically necessary to deal with several assertions at the same time and trace which transitions lead from one assertion to another. It is convenient to visualize this situation by a diagram that summarizes the assertions under consideration and the possible transitions between them.

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

- *Nodes* in the graph are labeled by assertions. 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 the name of a transition in the program. We refer to an edge labeled by τ as a *τ-edge*.
- One of the nodes may be designated as a *terminal node* ("goal" node). In the graphical representation, this node is distinguished by having a boldface boundary. No edges depart from a terminal node.

Verification diagrams provide a concise representation of sets of verification conditions as follows.

For a nonterminal node (labeled by) φ and transition τ, let $\varphi_1, \ldots, \varphi_k$ be the nodes reached by τ-edges departing from φ. The *verification condition associated with* φ and τ is given by

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

Note that for the case $k = 0$, i.e., no τ-edges depart from φ, the verification condition for φ and τ is given by

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

We describe the situation that no τ-edges depart from assertion φ by saying that there is an *implicit* τ-edge connecting φ to itself.

No verification conditions are associated with terminal nodes.

Since a diagram provides a succinct representation of a large set of verification conditions, it can often present a useful and illuminating overview of a complex proof.

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

3.1 Wait-for Diagrams

A verification diagram with nodes $\varphi_m, \ldots, \varphi_0$ is called a WAIT-FOR *diagram* if φ_0 is a terminal node and the diagram is weakly acyclic, i.e., whenever node φ_i is connected by an edge to node φ_j, then $i \geq j$. P-valid WAIT-FOR diagrams can be used to establish the P-validity of a nested wait-for formula.

The consequences of having a P-valid WAIT-FOR diagram are stated by the following claim.

Claim W A P-valid WAIT-FOR diagram establishes that the formula

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

is P-valid.

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

$$\text{(W1)} \quad p \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \text{(W2)} \quad \varphi_i \;\rightarrow\; q_i \quad \text{for} \quad i = 0, 1, \ldots, m,$$

then we we can conclude the P-validity of

$$p \;\Rightarrow\; q_m \; \mathcal{W} \; q_{m-1} \; \cdots \; q_1 \; \mathcal{W} \; q_0.$$

Justification Assume that the verification conditions associated with the given WAIT-FOR diagram are P-valid. Consider a state s_k satisfying $\bigvee_{j=0}^{m} \varphi_j$. It follows that there exists some index j_k, $0 \leq j_k \leq m$, such that s_k satisfies φ_{j_k}.

If $j_k = 0$ we are done, since φ_0 terminates the required sequence of intervals.

Otherwise, $j_k > 0$ and we consider s_{k+1}, the successor of s_k by some τ. The verification condition for φ_{j_k} and τ implies that s_{k+1} satisfies $\varphi_{j_{k+1}}$ for some node $\varphi_{j_{k+1}}$, a τ-successor of node φ_{j_k} in the diagram. Since the diagram is acyclic, $j_{k+1} \leq j_k$. We now repeat the argument for s_{k+1}, and so on.

Denoting the indices of the assertions established for s_k and its successors by $j_k, j_{k+1}, \ldots$, we obtain the following nonincreasing sequence of indices.

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

It is not difficult to see that this guarantees a sequence of intervals (some of which may be empty) satisfying respectively

$$\varphi_m, \; \varphi_{m-1}, \; \cdots,$$

which may either terminate at a state satisfying φ_0 or extend to infinity.

The second part of the claim follows by monotonicity of the temporal operators. ∎

Example (Peterson) As an example for an application of WAIT-FOR diagrams, consider again Peterson's algorithm for implementing mutual exclusion, presented in program PET of Figure 1.

One can establish the following invariants for this program:

$$\chi_0\colon \quad s = 1 \;\vee\; s = 2$$
$$\chi_1\colon \quad y_1 \;\leftrightarrow\; at\_\ell_{3..6}$$
$$\chi_2\colon \quad y_2 \;\leftrightarrow\; at\_m_{3..6}$$

χ_3: $\quad at\_\ell_4 \ \wedge \ at\_m_5 \ \rightarrow \ y_2 \ \wedge \ s=1$
χ_4: $\quad at\_\ell_5 \ \wedge \ at\_m_4 \ \rightarrow \ y_1 \ \wedge \ s=2.$

We wish to prove that a process waiting to enter its critical section can be overtaken at most once by its competitor. This property is called *1-bounded overtaking* and can be expressed by the nested wait-for formula:

$$\psi: \quad \underbrace{at\_\ell_4}_{p} \Rightarrow \underbrace{\neg at\_m_5}_{q_3} \ \mathcal{W} \ \underbrace{at\_m_5}_{q_2} \ \mathcal{W} \ \underbrace{\neg at\_m_5}_{q_1} \ \mathcal{W} \ \underbrace{at\_\ell_5.}_{q_0}$$

To use claim W, we have to find four assertions φ_0, φ_1, φ_2, φ_3, whose disjunction is implied by at_ℓ_4 (satisfying W1), which strengthen assertions at_ℓ_5, $\neg at\_m_5$, at_m_5, $\neg at\_m_5$, respectively (satisfying W2), and to construct a WAIT-FOR diagram such that all the verification conditions associated with the diagram are state valid for program PET. We propose the WAIT-FOR diagram of Figure 2.

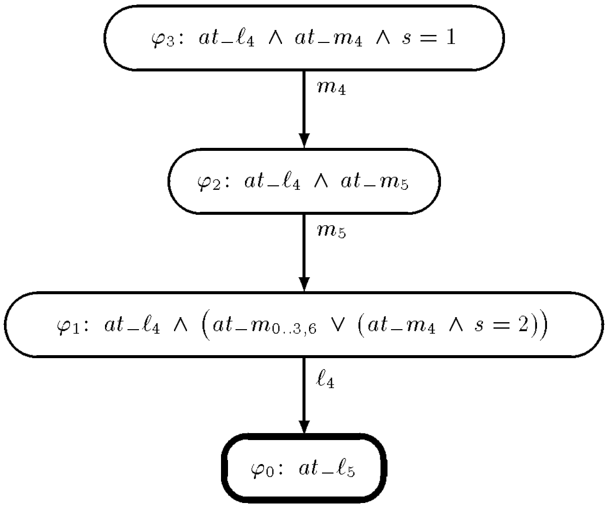

Fig. 2. WAIT-FOR diagram for 1-bounded overtaking from ℓ_4.

The diagram states the following verification conditions:

- Conditions from φ_3

  $\{\varphi_3\}\ m_4\ \{\varphi_3 \ \vee \ \varphi_2\} \qquad \{\varphi_3\}\ \overline{m_4}\ \{\varphi_3\}.$
- Conditions from φ_2

  $\{\varphi_2\}\ m_5\ \{\varphi_2 \ \vee \ \varphi_1\} \qquad \{\varphi_2\}\ \overline{m_5}\ \{\varphi_2\}.$
- Conditions from φ_1

  $\{\varphi_1\}\ \ell_4\ \{\varphi_1 \ \vee \ \varphi_0\} \qquad \{\varphi_1\}\ \overline{\ell_4}\ \{\varphi_1\}.$

In these conditions, we use the notation $\overline{m_4}$ to denote the set of all transitions in $\mathcal{T}$ different from m_4, and similarly for $\overline{m_5}$ and $\overline{\ell_4}$.

These verification conditions are not generally state valid. However, they are P-state valid for program PET. This can be proved by showing that each of them is implied by the conjunction of the invariants χ_0–χ_4.

Consequently, the WAIT-FOR diagram of Figure 2 is valid over program PET.

Also, the implications

$$\underbrace{at\_\ell_4}_{p} \;\rightarrow\; \bigvee_{j=0}^{3} \varphi_j$$

$$\varphi_0 \;\rightarrow\; \underbrace{at\_\ell_5}_{q_0}, \quad \varphi_1 \;\rightarrow\; \underbrace{\neg at\_m_5}_{q_1}, \quad \varphi_2 \;\rightarrow\; \underbrace{at\_m_5}_{q_2}, \quad \varphi_3 \;\rightarrow\; \underbrace{\neg at\_m_5}_{q_3},$$

are P-state valid.

Consequently, this establishes the validity of the formula

$$at\_\ell_4 \;\Rightarrow\; (\neg at\_m_5)\;\mathcal{W}\;at\_m_5\;\mathcal{W}\;(\neg at\_m_5)\;\mathcal{W}\;at\_\ell_5$$

over program PET. ▟

Completeness

Whenever a new formal proof method is introduced, a naturally arising question is how powerful is this new method? Is it strong enough to prove all valid properties of interest? If the answer is positive then we need not look for any other rules. On the other hand, if the answer is negative, we should look for additional rules and, whenever we fail to prove a property using the suggested rule, there is always the possibility that the failure is due to the shortcomings and inadequacy of the rule.

Fortunately, most of the diagram-based proof methods we introduce in this paper are complete. Here we state completeness for WAIT-FOR diagrams.

Claim CW If the formula

$$p \;\Rightarrow\; q_m \;\mathcal{W}\; q_{m-1} \;\cdots\; q_1 \;\mathcal{W}\; q_0.$$

is P-valid, then there exist assertions $\varphi_m, \ldots, \varphi_0$, and a P-valid WAIT-FOR diagram D whose nodes are labeled by $\varphi_m, \ldots, \varphi_0$, such that the following implications are P-state valid:

$$p \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \varphi_i \;\rightarrow\; q_i \quad \text{for} \quad i = 0, 1, \ldots, m.$$

The proof of the claim is based on constructions similar to the ones presented in [MP91a] and is brought in full in [MP94]. ▟

3.2 Invariance Diagrams

Another class of verification diagrams can be used to establish invariance properties.

A verification diagram is called an INVARIANCE *diagram* if it is exit-free, i.e., it has no terminal node. Unlike WAIT-FOR diagrams, INVARIANCE diagrams may contain cycles.

Claim I A P-valid INVARIANCE diagram with nodes $\varphi_1, \ldots, \varphi_m$ establishes that the formula

$$\bigvee_{j=1}^{m} \varphi_j \;\Rightarrow\; \Box\Big(\bigvee_{j=1}^{m} \varphi_j\Big).$$

is P-valid.

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

$$\text{(I1)}\quad \Theta \;\rightarrow\; \bigvee_{j=1}^{m} \varphi_j \qquad \text{and} \qquad \text{(I2)}\quad \bigvee_{j=1}^{m} \varphi_j \;\rightarrow\; q,$$

then we can conclude the P-validity of

$$\Box\, q.$$

Justification Denote by φ the disjunction

$$\varphi: \quad \bigvee_{j=1}^{m} \varphi_j.$$

Consider an assertion φ_i and a transition τ. Let $\varphi_{j_1}, \ldots, \varphi_{j_k}$ be all the τ-successors of φ_i. Since the diagram is P-valid, we know that the verification condition

$$\{\varphi_i\}\ \tau\ \{\varphi_i \vee \varphi_{j_1} \vee \cdots \vee \varphi_{j_k}\}: \quad \rho_\tau \wedge \varphi_i \;\rightarrow\; \varphi_i' \vee \varphi_{j_1}' \vee \cdots \vee \varphi_{j_k}'$$

is P-state valid. Obviously this implies the weaker implication

$$\{\varphi_i\}\ \tau\ \{\varphi\}: \quad \rho_\tau \;\wedge\; \varphi_i \;\rightarrow\; \varphi_1' \;\vee\; \cdots \;\vee\; \varphi_m'.$$

Since this was derived for an arbitrary $i \in [1..m]$, we can take the disjunction of the implications for all $i = 1, \ldots, m$. This leads to

$$\{\varphi\}\ \tau\ \{\varphi\}: \quad \rho_\tau \;\wedge\; (\varphi_1 \;\vee\; \cdots \;\vee\; \varphi_m) \;\rightarrow\; \varphi_1' \;\vee\; \cdots \;\vee\; \varphi_m'.$$

Since this was derived for an arbitrary $\tau \in \mathcal{T}$, we can take the conjunction of the verification conditions for all $\tau \in \mathcal{T}$ and conclude

$$\{\varphi\}\ \mathcal{T}\ \{\varphi\}.$$

We can therefore conclude

$$\varphi \;\Rightarrow\; \Box\,\varphi.$$

The second part of the claim follows by monotonicity. ▟

Example (Peterson) Let us establish that

$$\chi_1:\ y_1 \;\leftrightarrow\; at\_\ell_{3..6}$$

is an invariant of program PET (Figure 1), that is,

$$\Box(\underbrace{y_1 \;\leftrightarrow\; at\_\ell_{3..6}}_{q}).$$

Consider the INVARIANCE diagram presented in Figure 3. It is not difficult to ascertain that this diagram is valid for program PET.

Condition I1, monotonicity with respect to Θ, follows from the fact that

$$\underbrace{at\_\ell_0 \;\wedge\; \neg y_1 \;\wedge\; \cdots}_{\Theta} \quad\rightarrow\quad \underbrace{at\_\ell_{0..2} \;\wedge\; \neg y_1}_{\varphi_1} \;\vee\; \underbrace{\cdots}_{\varphi_2}.$$

It is straightforward to establish condition I2, monotonicity with respect to q, observing that each of the assertions implies q:

$$\underbrace{at\_\ell_{0..2} \;\wedge\; \neg y_1}_{\varphi_1} \;\rightarrow\; \underbrace{y_1 \;\leftrightarrow\; at\_\ell_{3..6}}_{q}$$

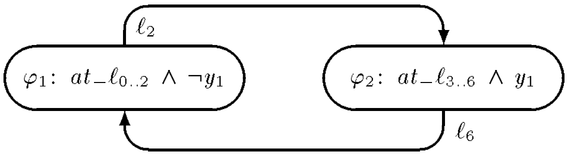

Fig. 3. INVARIANCE diagram.

$$\underbrace{at\_\ell_{3..6} \;\wedge\; y_1}_{\varphi_2} \;\;\rightarrow\;\; \underbrace{y_1 \;\leftrightarrow\; at\_\ell_{3..6}}_{q}\,.$$

We conclude that the assertion

$$y_1 \;\leftrightarrow\; at\_\ell_{3..6}$$

is an invariant of program PET. ▟

Completeness of Invariance Diagrams

The following claim states that invariance diagrams are complete for proving invariance properties.

> **Claim CI** If the formula $\square\, q$ is P-valid, then there exist assertions $\varphi_m, \ldots, \varphi_1$, and a P-valid INVARIANCE diagram D whose nodes are labeled by $\varphi_m, \ldots, \varphi_1$, such that the following implications are P-state valid:
>
> $$\Theta \;\rightarrow\; \bigvee_{j=1}^{m} \varphi_j \qquad \text{and} \qquad \bigvee_{j=1}^{m} \varphi_j \;\rightarrow\; q,$$

In fact, the claim can alwyas be satsified with $m = 1$, i.e., using a single intermediate assertion.

The proof of this claim is presented in [MP91a] for a rule which is the textual analogue of INVARIANCE diagrams.

3.3 Compound Nodes

We introduce some encapsulation conventions, inspired by the Statecharts formalism of Harel [Har87]. These conventions lead to more structured 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 of Figure 4.

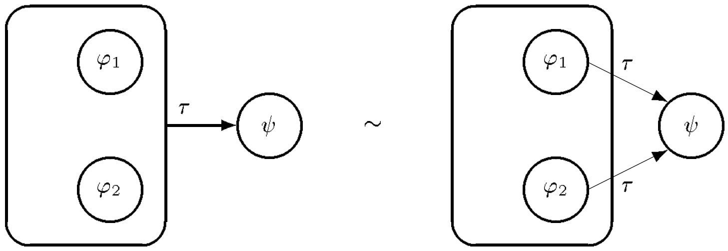

Fig. 4. Departing edges.

- *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 of Figure 5.

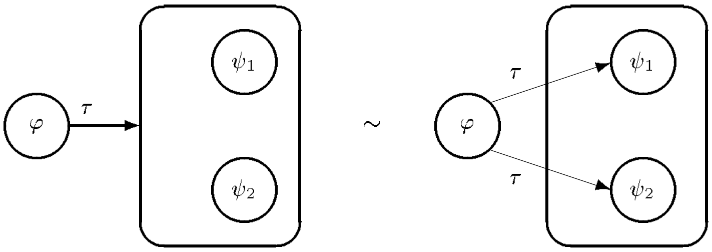

Fig. 5. Arriving edges.

- *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 Figure 6.

Example (Peterson) As illustration for the encapsulation conventions, we present in Figure 7 a WAIT-FOR diagram which presents the information of Figure 2 in a different form.

Note that the diagram of Figure 7 contains more information than that of Figure 2. It indicates that, within the compound node φ_1, the program can only progress from $at\_m_{0..3,6}$ to $at\_m_4 \land s = 2$, but not vice versa. ▟

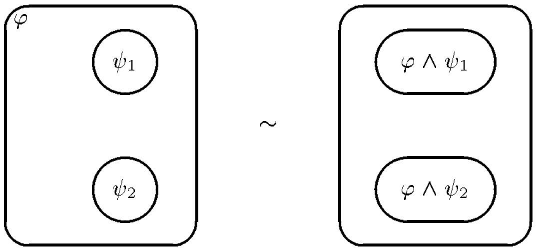

Fig. 6. Common factors.

4 Verification Diagrams for Response Properties

In this section, we present verification diagrams that support proofs of response properties; that is, properties specifiable by formulas of the form

$$p \Rightarrow \Diamond q,$$

for state formulas p and q.

To handle these types of properties, we extend the notion of verification diagrams by allowing two kinds of edges in the graph: a single-lined edge and a double-lined edge. For example, the diagram of Figure 8 contains a single edge connecting node φ_3 to node φ_2 and a double edge connecting node φ_2 to node φ_1. Both kinds of edges are labeled by transitions. As will be explained below, double edges represent *helpful* transitions that are guaranteed to bring us closer to the goal q.

4.1 Chain Diagrams

A verification diagram with nodes $\varphi_m, \ldots, \varphi_0$ is called a CHAIN *diagram* if φ_0 is a terminal node and the following requirements are satisfied.

- If a single (-line) edge connects node (labeled by) φ_i to node φ_j, then $i \geq j$.
- If a double (-line) edge connects node (labeled by) φ_i to node φ_j, then $i > j$.
- Every node φ_i, $i > 0$, has a double edge departing from it. This identifies the transition labeling such an edge as *helpful* for assertion φ_i. All helpful transitions must be just.
- No transition can label both a single and a double edge departing from the same node.

The first two requirements ensure that the diagram is weakly acyclic in the sense defined earlier, that is, whenever node φ_i is connected by an edge (single or double) to node φ_j, $j \leq i$. The stronger second requirement ensures that the subgraph based on the double edges is acyclic, forbidding self-connections by double edges. The third requirement demands that every nonterminal assertion (i.e., φ_i for $i > 0$) has at least one helpful transition associated with it.

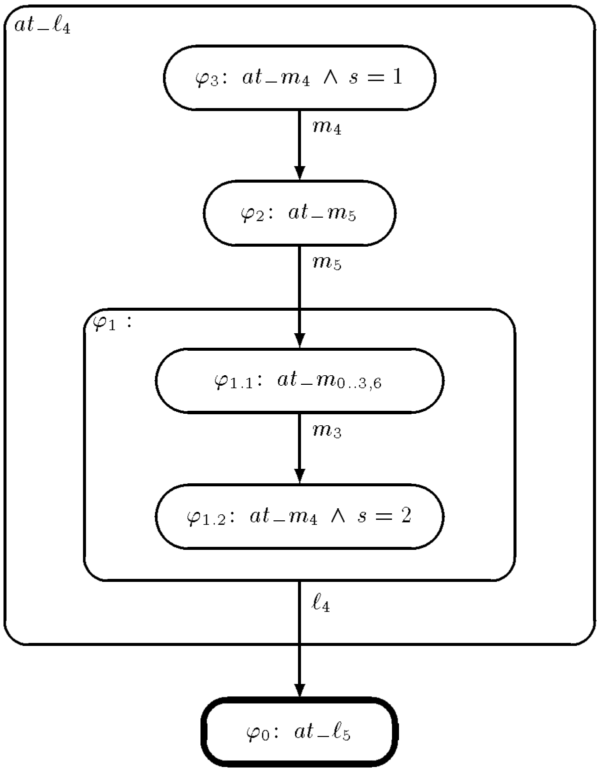

Fig. 7. Structured WAIT-FOR diagram for 1-bounded overtaking.

Verification Conditions for CHAIN Diagrams

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

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

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

  Transition τ, labeling only single-lined edges, is identified as unhelpful for φ. The associated verification condition allows τ to lead from a φ-state back to a φ-state, recording no progress.
  The case of a transition τ that does not label any edges departing from φ is interpreted as though τ labels $k = 0$ single-lined edges departing from φ. That is, with such a transition we associate the verification condition

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

- If $k > 0$ and all edges connecting φ to its τ-successors are double, we associate with φ and τ the verification condition

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

Transition τ, identified as helpful, is required to lead away from φ. This, with the requirement of acyclicity, implies that when this transition is taken from a φ-state, the computation gets closer to the goal φ_0.

- If τ labels a double edge departing from φ, we require

  $$\varphi\ \rightarrow\ En(\tau),$$

  that is, a transition helpful for φ is enabled on all φ-states. We refer to this requirement as the *enabling requirement*.

Valid CHAIN Diagrams

A CHAIN diagram is said to be *valid over program P* (*P-valid* for short) if all the verification conditions and enabling requirements are P-state valid for every nonterminal node φ_i, $i > 0$, and every transition τ.

The consequences of having a P-valid CHAIN diagram are stated in the following claim.

Claim C A P-valid CHAIN diagram establishes that the response formula

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

is P-valid.

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

$$\text{(C1)}\quad p\ \rightarrow\ \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \text{(C2)}\quad \varphi_0\ \rightarrow\ q,$$

then we can conclude the P-validity of

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

Justification Assume that all the verification conditions of a given CHAIN diagram are P-state valid. Consider a P-computation σ and a position k that satisfies $\bigvee_{j=0}^{m} \varphi_j$. We wish to prove that some later position satisfies φ_0.

Assume to the contrary that all positions later than k (including k itself) do not satisfy φ_0. Since position k satisfies the disjunction, it must satisfy φ_j for some $j \geq 0$. Index j cannot be 0 because we assumed that no position beyond k satisfies φ_0. Thus, position k satisfies φ_j, for some $j > 0$. Let τ be the transition taken at k and leading from state s_k to state s_{k+1}. By the verification condition corresponding to φ_j and transition τ, state s_{k+1} satisfies some φ_r, for some node φ_r which is one of the τ-successors of node φ_j. Independently of whether τ labels a single or a double edge, the weak acyclicity of a CHAIN diagram implies $r \leq j$. Again, $r > 0$, due to our assumption that φ_0 never holds beyond k. Continuing in this manner, it follows that every position beyond k satisfies some φ_j for $j > 0$, to which we refer as the *rank* of the position.

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

Assume that $i > 0$ is the rank of the state at position t. Let τ_i be the transition identified as helpful for φ_i, i.e., the transition which labels a double edge departing from φ_i. Since φ_0 is never satisfied and the rank never decreases beyond position t, it follows that φ_i holds continually beyond t and transition τ_i is never taken. This is because, by the verification condition for double edges, taking a helpful transition always decreases the rank. By the enabling requirement, τ_i is continually enabled beyond t. Yet, by the argument above, it is never taken. This violates the requirement of justice for τ_i.

It follows that if all the premises of the rule are P-state valid then the conclusion $\bigvee_{j=0}^{m} \varphi_j \;\Rightarrow\; \diamondsuit\, \varphi_0$ is P-valid. ▟

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

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

Example (Peterson) We intend to verify the property of accessibility for program PET, which can be expressed by the response formula

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

The CHAIN diagram for the proof of this property is presented in Figure 8.

This diagram partitions the range $m_{0..4}$ into three regions. The region $m_{0..2}$, represented by φ_3, guarantees the enableness of ℓ_4. However the computation may move into the state space described by φ_2, where no transition of P_1 is guaranteed to be enabled. Being at φ_2, m_3 is the helpful transition which eventually leads into φ_1. In φ_1, ℓ_4 is enabled, and since P_2 cannot falsify $s = 2$, eventually ℓ_4 is taken and leads to φ_0. ▟

4.2 Rank Diagrams

CHAIN diagrams can only establish response properties that rely for their correctness on a bounded number of helpful steps. This is why no completeness claim was stated for these diagrams. To extend the method to properties that require an unbounded number of helpful steps, we need several extensions.

We define a *well-founded domain* $(\mathcal{A}, \succ)$ to consist of a set $\mathcal{A}$, and a *well-founded order* relation $\succ$ over $\mathcal{A}$. A binary relation $\succ$ is called an *order* if it is transitive (i.e., $a \succ b$ and $b \succ c$ imply $a \succ c$) and irreflexive (i.e., $a \succ a$ for no $a \in \mathcal{A}$). The relation $\succ$ is called *well-founded* if there does not exist an infinite 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 $\mathbb{N}$ is the set of natural numbers (including 0), and $>$ is the greater-than relation.

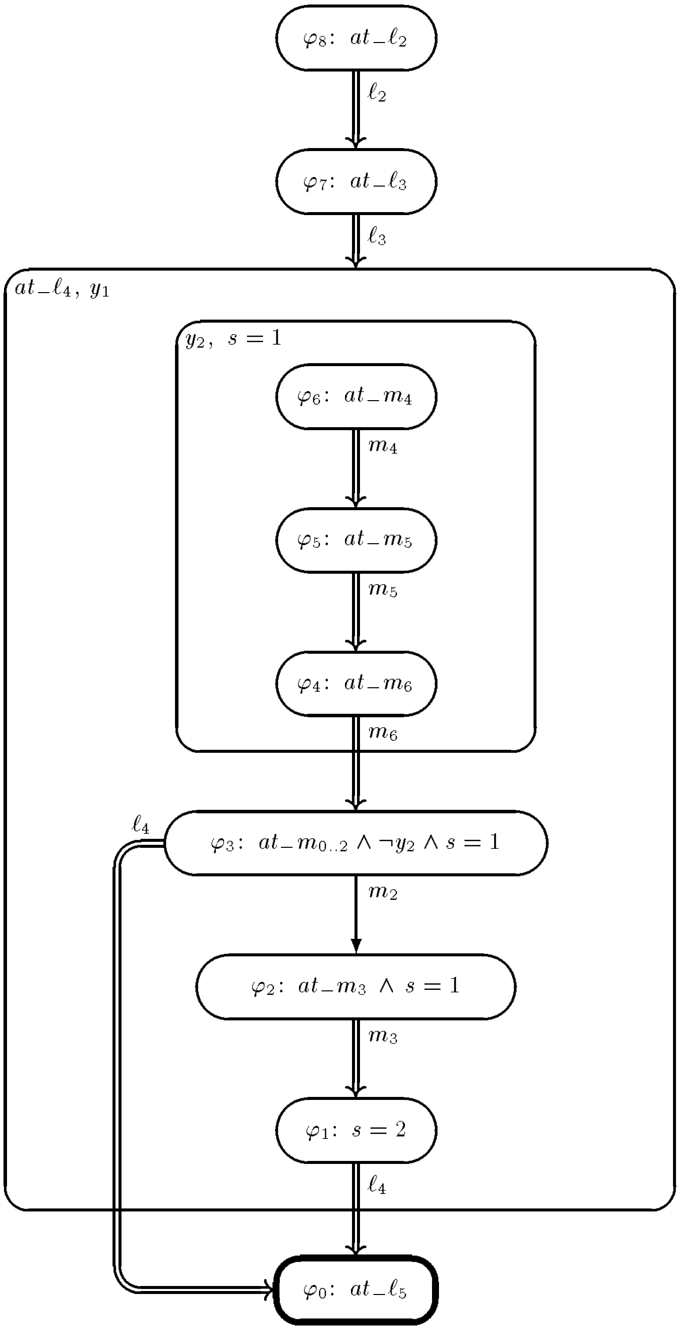

Fig. 8. CHAIN diagram for accessibility.

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

- $\mathcal{A}$ is defined as $\mathcal{A}_1 \times \mathcal{A}_2$, i.e., the set of pairs (a_1, a_2), such that $a_1 \in \mathcal{A}_1$ and $a_2 \in \mathcal{A}_2$.
- $\succ$ is a *lexicographic* order defined to hold between $(a_1, a_2) \in \mathcal{A}$ and $(b_1, b_2) \in \mathcal{A}$ iff
$$a_1 \succ_1 b_1 \quad \text{or} \quad a_1 = b_1 \ \wedge \ a_2 \succ_2 b_2.$$

The importance of the lexicographic product follows from the observation that, if the domains $(\mathcal{A}_1, \succ_1)$ and $(\mathcal{A}_2, \succ_2)$ are well-founded, then so is their lexicographic product $(\mathcal{A}, \succ)$.

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

- Every node φ_i, $i > 0$, has a double edge departing from it. This identifies the transition labeling such an edge as *helpful* for assertion φ_i. All helpful transitions must be just.
- No transition can label both a single and a double edge departing from the same node.

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

Verification and Enabling Requirements for RANK Diagrams

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

- If transition τ is unhelpful for φ, i.e., labels only single edges departing from the node, then we associate with φ and τ the following verification condition
$$\Big\{\varphi \wedge \delta = u\Big\} \ \tau \ \Big\{(\varphi \wedge u \succeq \delta) \ \vee \ (\varphi_1 \wedge u \succ \delta_1) \ \vee \ \cdots \ \vee \ (\varphi_k \wedge u \succ \delta_k)\Big\}.$$
- If τ is helpful for φ (labels double edges), we associate with φ and τ the following verification condition
$$\Big\{\varphi \wedge \delta = u\Big\} \ \tau \ \Big\{(\varphi_1 \wedge u \succ \delta_1) \ \vee \ \cdots \ \vee \ (\varphi_k \wedge u \succ \delta_k)\Big\}.$$
- For every nonterminal node φ and a transition τ labeling a double edge departing from φ, we require
$$\varphi \quad \rightarrow \quad En(\tau).$$

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

A RANK 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 RANK diagram are stated in the following claim.

Claim R A P-valid RANK diagram establishes that the response formula

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

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

$$p \quad \Rightarrow \quad \diamondsuit\, q.$$

Justification Assume that the verification conditions of a given RANK diagram are P-state valid. Consider a P-computation σ and a position k that satisfies $\bigvee_{j=0}^{m} \varphi_j$. We wish to prove that some later position satisfies φ_0. Assume to the contrary that all positions later than k (including k itself) do not satisfy φ_0.

For every position $t \geq k$, let φ_{i_t} be the assertion with the lowest index holding at t, and let $r_t \in \mathcal{A}$ be the value of δ_{i_t} at position t. We refer to r_t as the rank at position t.

Since position k satisfies the disjunction, it must satisfy φ_j for some $j \geq 0$, and therefore the assertion φ_{i_k} and the rank r_k are well-defined at k. By the third verification condition, transition τ_{i_k} (the helpful transition for φ_{i_k}) is enabled at position k.

Consider the transition τ taken at position k, leading into position $k+1$. By the first and second verification conditions associated with φ_{i_k} and τ, either position $k+1$ has a lower rank $u_{k+1} \prec u_k$, or it has the same rank, but then τ_{i_k} is still the helpful transition at $k+1$ and is enabled there. In the case that the rank at $k+1$ equals r_k, we can continue the argument from $k+1$ to $k+2$, $k+3$, etc. However, we cannot have all positions $t > k$ with the same rank. Otherwise, by the third verification condition, τ_{i_k} would be continuously helpful and enabled and, by the second verification condition, τ_{i_k} is not taken beyond k. Thus, assumping that all positions beyond k have the same rank leads to the situation that τ_{i_k} is continuously enabled and not taken, violating the justice requirement for τ_{i_k}.

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

We conclude that every $\bigvee_{j=0}^{m} \varphi_j$-position must be followed by a φ_0-position, establishing the consequence of the rule. ▟

4.3 Distributing the Ranking Functions

To make RANK diagrams more readable, we introduce an additional encapsulation convention. A previously introduced convention allowed a compound 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. 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 node was labeled with the ranking function $\delta\colon\ i$, where i is the index of the node (and the assertion labeling it).

Sometimes we are faced with the question of how to compare lexicographic tuples of unequal lengths. Since all our examples will be based on tuples of nonnegative integers, we agree that the relation holding between $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_k)$ for $i < k$ is determined by lexicographically comparing $(a_1, \ldots, a_i, 0, \ldots, 0)$ to $(b_1, \ldots, b_i, b_{i+1}, \ldots, b_k)$. That is, we pad the shorter tuple by zeros on the right until it assumes the length of the longer tuple.

Example (up down) Consider program UP-DOWN presented in Figure 9. Process P_1 increments y, counting up at locations ℓ_0, ℓ_1, as long as $x = 0$. Once P_1 finds that x is different from 0, it proceeds to ℓ_2, ℓ_3, where y is decremented until it becomes 0. The single action of Process P_2 is to set x to 1. Obviously, due to justice, x will eventually be set to 1. However, one cannot predict the number of helpful steps required for P_1 to terminate. The longer P_2 waits before performing m_0, the higher value y will attain on the move to ℓ_2. It is this value of y which determines the number of remaining steps to termination.

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

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

Fig. 9. Program UP-DOWN.

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

Consequently, we need claim R to prove the response property

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

for program UP-DOWN.

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

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

$$(n_1,\ n_2,\ n_3) \ \succ \ (m_1,\ m_2,\ m_3)$$

$$\text{iff} \ \ n_1 = m_1,\ \ldots,\ n_{i-1} = m_{i-1}, \ \text{ and } \ n_i > m_i \ \text{ for some } \ i,\ 1 \le i \le 3.$$

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

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

In Figure 10 we present a valid RANK diagram which implies, by monotonicity, the property of termination for program UP-DOWN.

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

The diagram provides a detailed description of the progress of the computation from φ_5 to φ_0. It shows that progress from φ_5 to φ_2 is due to a CHAIN-like reasoning. Then, the progress from φ_2 and φ_1 to φ_0 requires a well-founded ar-

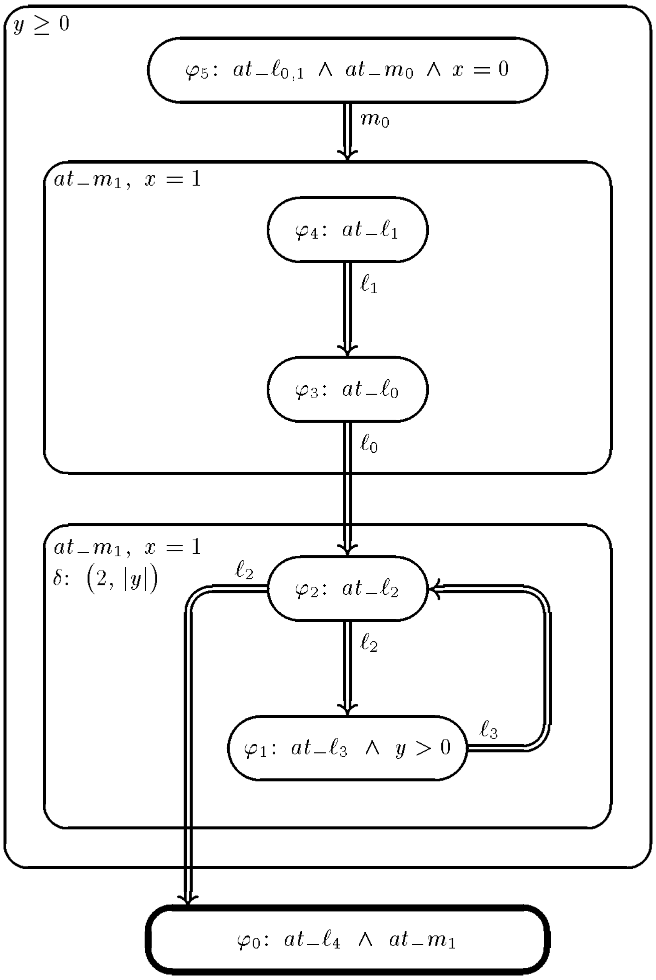

Fig. 10. RANK diagram for termination.

gument with the measure $|y|$ for coarse progress, and the index $j = 1, 2$ of φ_j for measuring fine progress.

When padded to the maximum length of 3, the ranking functions appearing in this diagram are given by

δ_5: $(5,\ 0,\ 0)$
δ_4: $(4,\ 0,\ 0)$
δ_3: $(3,\ 0,\ 0)$
δ_2: $(2,\ |y|,\ 2)$
δ_1: $(2,\ |y|,\ 1)$
δ_0: $(0,\ 0,\ 0)$.

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

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

Completeness of RANK Diagrams

RANK diagrams only take account of just transitions. Consequently, their completeness is limited to programs with empty compassion sets, as stated by the following claim.

Claim CR Let P be a program with empty compassion set. If the formula

$$p \ \Rightarrow\ \diamondsuit q.$$

is P-valid, then there exists a P-valid RANK diagram D whose nodes are labeled by assertions $\varphi_m, \ldots, \varphi_0$ and ranking functions $\delta_m \ldots, \delta_0$ mapping states into some well-founded domain $(\mathcal{A}, \succ)$, such that the following implications are P-state valid:

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

It can be shown that it is enough to take m to be the number of just transitions in program P.

The proof of this claim is based on the constructions presented in [MP91a]. As is shown there, it is necessary to extend the assertion language to include a fixed-point operator and it is convenient to use a segment of the countable ordinals as a canonical well-founded domain.

5 Parameterized Programs

Parameterized programs allow the variable size (dynamic) statements

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

In the general case, the elaboration of such a statement is deferred until execution reaches it. At that point, the variable (or expression) M is evaluated and the program prepares to execute the statement

$$S[1] \,||\, \ldots \,||\, S[m],$$

where m is the current value of M. Each parameterized statement $S[i]$, for $i = 1, \ldots, m$, is obtained by substituting the numeral i for the formal parameter j appearing in the text of the program and using it as a subscript in the labels and locally declared variables. Note that the variable M may be an input variable or may even be computed during the execution up to this point. In both cases, it is required that, on arrival to the dynamic statement, M has the value of a positive integer.

In the simpler cases considered here, we only consider variable size parallel statements as the body of the program. This means that, for a given value m of variable M, we execute the program

$$\Big[\text{declaration;}\ \big[S[1] \parallel \cdots \parallel S[m]\big]\Big]$$

To verify parameterized programs, we introduce several notations: If ℓ_i is a location within the parameterized statement $S[k]$, we write $at\_\ell_i[k]$ to denote that control is currently at location ℓ_i within statement $S[k]$. For a parameterized program, we write L_i for the set of processes whose control is currently at ℓ_i, i.e.,

$$L_i \quad = \quad \big\{k \mid k \in [1..M] \ \wedge \ at\_\ell_i[k]\big\},$$

and

$$N_i = |L_i|$$

for the size of this set. These notations are extended to sets and ranges of locations, such as

$$L_{i..j} \ = \ L_i \ \cup \ L_{i+1} \ \cup \ \cdots \ \cup \ L_j \quad \text{and} \quad N_{i..j} \ = \ |L_{i..j}|.$$

To represent proofs of parameterized programs by verification diagrams, we follow the same conventions as before, but with one exception: edges are labeled by sets of transitions rather than by a single transition.

We require that, if a node has a departing edge labeled by set expression X_1 and another departing edge labeled by X_2, then either $X_1 = X_2$ or $X_1 \cap X_2 = \emptyset$. Furthermore, every nonterminal node has precisely one set expression h labeling double edges departing from the node. The expression h does not label any single edge departing from that node.

The verification conditions associated with edges labeled by set expressions are described below. We illustrate the generalization due to set expressions labeling edges in RANK diagrams. We refer to RANK diagrams with this extension as P-RANK *diagrams*. We can similarly extend the INVARIANCE, WAIT-FOR, and CHAIN diagrams to handle parameterized programs conveniently, by allowing edges to be labeled with sets of transitions.

Verification and Enabling Requirements for P-RANK Diagrams

Consider a nonterminal node n labeled by assertion φ and ranking function δ, and let X be a set expression specifying a set of transitions, which labels one or more edges departing from n. Let $\varphi_1, \ldots, \varphi_k$, $k > 0$, be the successors of φ by edges labeled with X, and $\delta_1, \ldots, \delta_k$ be the ranking functions of these successors. Let h be the set expression labeling double edges departing from n.

- If X labels a single edge, then the following verification condition is implied for every $\tau \in X$

  $$\{\varphi \ \wedge \ \delta = u \ \wedge \ e(h) = E\} \ \ \tau$$
  $$\big\{(\varphi \ \wedge \ u = \delta \ \wedge \ E \subseteq e(h)) \ \vee \ (\varphi_1 \ \wedge \ u \succ \delta_1) \ \vee \ \cdots \ \vee \ (\varphi_k \ \wedge \ u \succ \delta_k)\big\}.$$

  The notation $e(h)$ refers to the set of all transitions in h that are currently enabled. The verification condition allows a successor of a φ-state to be another φ-state with equal rank, but then, the set of enabled helpful transitions characterized by $e(h)$ cannot decrease.

- If $X = h$ is the helpful set expression labeling double edges, then the following verification condition is implied for every $\tau \in h$
$$\{\varphi \ \wedge \ \delta = u\} \ \tau \ \{(\varphi_1 \ \wedge \ u \succ \delta_1) \ \vee \ \cdots \ \vee \ (\varphi_k \ \wedge \ u \succ \delta_k)\}.$$
- The following verification condition is implied for every transition $\tau \in \mathcal{T}$ that does not belong to any set expression X labeling an edge departing from node n
$$\{\varphi \ \wedge \ \delta = u \ \wedge \ e(h) = E\} \ \tau \ \{\varphi \ \wedge \ u = \delta \ \wedge \ E \subseteq e(h)\}.$$
- With node n and helpful set expression h, we associate the enabling requirement
$$\varphi \quad \rightarrow \quad \exists \tau \in h\colon \ En(\tau).$$

Example (turn setting) Consider parameterized program TURN of Figure 11 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 angular brackets at statement ℓ_3 signify a *grouped statement*, which means that the whole enclosed statement is to be performed in a single transition. This grouped statement sets t to i only if it finds $t = 0$. If $P[i]$ finds $t \neq 0$ it does not modify t but still moves to ℓ_4. At ℓ_4 the process checks whether its execution of ℓ_3 succeeded in setting t to i. Finding a value $t = i$ at ℓ_4, $P[i]$ enters its critical section ℓ_5; otherwise it returns to ℓ_2 and then to ℓ_3 to try again.

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

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

Fig. 11. Program TURN (mutual exclusion by turn setting).

Mutual exclusion is ensured by the invariant
$$\chi\colon \quad i \in L_{5,6} \quad \rightarrow \quad t = i.$$
We wish to show a weak-accessibility property, formulated by
$$\psi\colon \quad \underbrace{N_2 > 0 \ \wedge \ t = 0}_{p} \quad \Rightarrow \quad \diamondsuit \underbrace{N_5 > 0}_{q}.$$
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, can be expressed by the formula

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

which is not valid for this program.

To verify the weak-accessibility property expressed by ψ, we use a parameterized version of the chain diagram, presented in Figure 12.

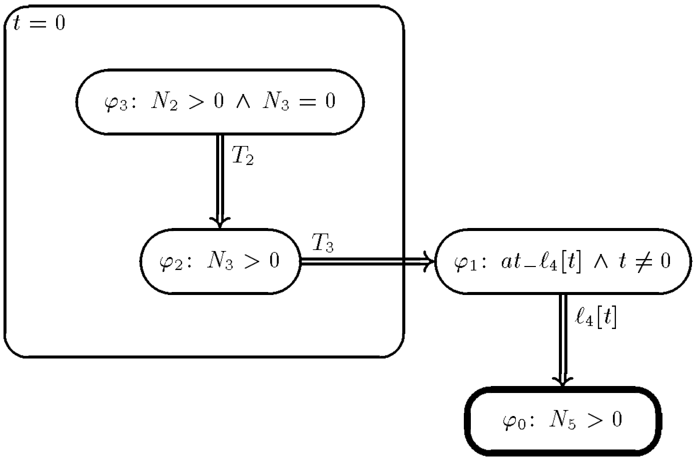

Fig. 12. P-RANK diagram for accessibility.

This diagram uses the set expressions T_2 and T_3, which are defined as follows.

$$T_2 : \quad \{\tau_{\ell_2}[k] \mid k \in [1..M]\}$$
$$T_3 : \quad \{\tau_{\ell_3}[k] \mid k \in [1..M]\}$$

Thus, the set T_2 includes the transitions $\ell_2[k]$ of all the processes $P[k]$, $k \in [1..M]$.

It is not difficult to see that φ_3 implies that $e(T_2)$ is not empty, i.e., at least one transition of T_2 is enabled. Similarly, φ_2 implies that $e(T_3)$ is not empty. ▟

6 Verification Diagrams with Compassionate Transitions

Diagrams can also be used to provide a concise representation of proofs handling the compassion restriction. The main extension over the previous verification diagrams is the introduction of a third type of edge — a *solid* edge, which corresponds to a helpful transition that is compassionate. Consequently, edges connecting nodes (assertions) in a diagram can be single, double, or solid.

We will rarely have diagrams all of whose helpful edges are solid. The common situation is that some of the helpful transitions will be just (and label double edges) and some others will be compassionate (labeling solid edges).

Consequently, we view these diagrams as relying on the full spectrum of fairness requirements and refer to them as F-CHAIN and F-RANK diagrams.

6.1 F-chain Diagrams

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

- If a single edge connects node φ_i to node φ_j, then $i \geq j$.
- If a double or a solid edge connects φ_i to φ_j, then $i > j$.
- Every node φ_i, $i > 0$, has at least one departing edge that is either double or solid. The transition labeling this edge is identified as helpful for assertion φ_i.
- Every transition can label at most one type of edge (single, double, or solid) departing from the same node.

Verification Conditions for F-CHAIN Diagrams

Let φ be a nonterminal node and $\varphi_1, \ldots, \varphi_k$, $k \geq 0$, be the τ-successors of φ.

- If τ labels a single edge, we associate with φ and τ the verification condition
  $$\{\varphi\}\ \tau\ \{\varphi\ \vee\ \varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}.$$
- If τ labels a double or solid edge, we associate with φ and τ the verification condition
  $$\{\varphi\}\ \tau\ \{\varphi_1\ \vee\ \cdots\ \vee\ \varphi_k\}.$$
- If τ labels a double edge departing from φ, we associate with φ and τ the enabling requirement
  $$\varphi\ \rightarrow\ En(\tau).$$
- If τ labels a solid edge departing from φ, we associate with φ and τ the following enabling requirement
  $$\varphi\ \Rightarrow\ \Diamond\big(\neg\varphi\ \vee\ En(\tau)\big).$$
  Note that, unlike previous enabling requirements that were state formulas, the enabling requirement associated with solid edges is a response formula.

Valid F-CHAIN Diagrams

An F-CHAIN diagram is said to be P-*valid* if the verification conditions associated with all edges and the enabling requirements associated with double edges are P-state valid, and the enabling requirements associated with solid edges are P-valid.

The consequences of having a valid F-CHAIN diagram are stated in the following claim.

> **Claim FC** A P-valid F-CHAIN diagram establishes that the response formula

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

is P-valid.

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

$$\text{(FC1)} \quad p \;\rightarrow\; \bigvee_{j=0}^{m} \varphi_j \qquad \text{and} \qquad \text{(FC2)} \quad \varphi_0 \;\rightarrow\; q$$

then, we can conclude the P-validity of

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

Example (producer-consumer) We illustrate the use of F-CHAIN diagrams on program PROD-CONS, presented in Figure 13.

$$
\begin{array}{ll}
\textbf{local} & r, ne, nf : \textbf{integer where } r = 1, ne = N, nf = 0 \\
 & b \qquad\quad\;\; : \textbf{list of integer where } b = \Lambda
\end{array}
$$

$$
\begin{array}{l}
Prod :: \left[\begin{array}{l}
\textbf{local } x : \textbf{integer} \\[6pt]
\ell_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\text{: } \textbf{produce } x \\
\ell_2\text{: } \textbf{request } ne \\
\ell_3\text{: } \textbf{request } r \\
\ell_4\text{: } b := b \bullet x \\
\ell_5\text{: } \textbf{release } r \\
\ell_6\text{: } \textbf{release } nf
\end{array}\right]
\end{array}\right] \\
\| \\
Cons :: \left[\begin{array}{l}
\textbf{local } y : \textbf{integer} \\[6pt]
m_0\text{: } \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
m_1\text{: } \textbf{request } nf \\
m_2\text{: } \textbf{request } r \\
m_3\text{: } (y, b) := (hd(b),\; tl(b)) \\
m_4\text{: } \textbf{release } r \\
m_5\text{: } \textbf{release } ne \\
m_6\text{: } \textbf{consume } y
\end{array}\right]
\end{array}\right]
\end{array}
$$

Fig. 13. Program PROD-CONS (producer-consumer).

The statement at location m_3 of this program is the multiple assignment

$$m_3 : \; (y, b) := (hd(b),\; tl(b))$$

which, in one transition, assigns to y the first element of the list b, and reassigns to b the value of the list minus its first element.

The response property we wish to establish for this program is accessibility, which for process *Prod* is given by

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

In the proof, we are aided by the following invariants

$$\begin{aligned}
\chi_0&: \quad r \geq 0 \;\wedge\; ne \geq 0 \;\wedge\; nf \geq 0\\
\chi_1&: \quad r \,+\, at_\ell_{4,5} \,+\, at_m_{3,4} \;=\; 1\\
\chi_2&: \quad ne \,+\, nf \,+\, at_\ell_{3..6} \,+\, at_m_{2..5} \;=\; N.
\end{aligned}$$

The accessibility proof is presented in the CHAIN diagram of Figure 14.

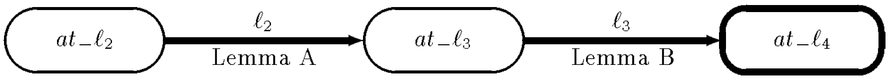

Fig. 14. F-CHAIN diagram for accessibility.

Since each transition labeling a solid edge needs a subproof that establishes the validity of the compassionate requirement $\varphi \Rightarrow \Diamond(\neg\varphi \vee En(\tau))$, we often attach a reference to such edges, which points to the subproofs.

Since ℓ_2 and ℓ_3 are compassionate *request* statements, we need to prove separately,

$$\text{Lemma A} \qquad \underbrace{at\_\ell_2}_{\varphi_2} \;\Rightarrow\; \Diamond\Big(\underbrace{\cdots}_{\neg\varphi_2} \;\vee\; \underbrace{at\_\ell_2 \;\wedge\; ne > 0}_{En(\ell_2)}\Big)\,.$$

The proof is presented in the auxiliary CHAIN diagram of Figure 15(a). The proof uses the invariant $ne \geq 0$ to infer that if $\varphi_2 : at\_\ell_2$ holds but transition ℓ_2 is disabled, then $ne = 0$. Invariant χ_2 is used to deduce that, when $ne = 0$ and $at\_\ell_2 \wedge at\_m_1$ holds, then $nf = N > 0$. Invariant χ_1 is used to deduce that, when $at\_\ell_2 \wedge at\_m_2$ holds, then $r = 1 > 0$.

An interesting point is that, even though m_1 and m_2 are also compassionate transitions, they are treated as just transitions in the CHAIN diagram of Figure 15(a). This is possible because, under the assumption that P_1 is frozen at ℓ_2 and $ne = 0$, these transitions are immediately enabled when control is at their respective locations.

We also need to prove separately,

$$\text{Lemma B} \quad \underbrace{at\_\ell_3}_{\varphi_1} \;\Rightarrow\; \Diamond\Big(\underbrace{\cdots}_{\neg\varphi_1} \;\vee\; \underbrace{at\_\ell_3 \;>\; r > 0}_{En(\ell_3)}\Big)\,.$$

The proof is presented in the auxiliary CHAIN diagram of Figure 15(b). The proof uses the invariant $r \geq 0$ to infer that if $r \leq 0$ then $r = 0$. Then, we use χ_1 to deduce that if $r = 0$ and P_1 is at ℓ_3, then P_2 can only be at m_3 or at m_4.

This concludes the proof of the accessibility property

$$at\_\ell_2 \;\Rightarrow\; \Diamond\, at\_\ell_4. \quad ▟$$

As is the case with RANK diagrams, no completeness result can be claimed for F-CHAIN diagrams. However, F-RANK diagrams, which extend the class of

RANK diagrams by allowing solid edges labeled by compassionate transitions are complete for proving any P-valid response formula, where P is a program with arbitrary compassion set. Details of the proof can be found in [MP91a].

Acknowledgement

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

References

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

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

[HO83] B.T. Hailpern and S.S. Owicki. Modular verification of computer commuincation protocols. *IEEE Trans. on Commun.*, COM-31(1):56–68, 1983.

[Lam91] L. Lamport. The temporal logic of actions. Technical report, Digital Equipment Corporation, Systems Research Center, 1991.

[MP83] Z. Manna and A. Pnueli. Verification of concurrent programs: A temporal proof system. In J.W. de Bakker and J. Van Leeuwen, editors, *Foundations of Computer Science IV, Distributed Systems: Part 2*, pages 163–255. Mathematical Centre Tracts 159, Center for Mathematics and Computer Science (CWI), Amsterdam, 1983.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comp. Prog.*, 32:257–289, 1984.

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

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

[MP93] Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.

[MP94] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems.* Springer-Verlag, New York, 1994. To Appear.

[NGO85] V. Nguyen, D. Gries, and S. Owicki. A model and temporal proof system for network of processes. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 121–131, 1985.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Prog. Lang. Sys.*, 4:455–495, 1982.

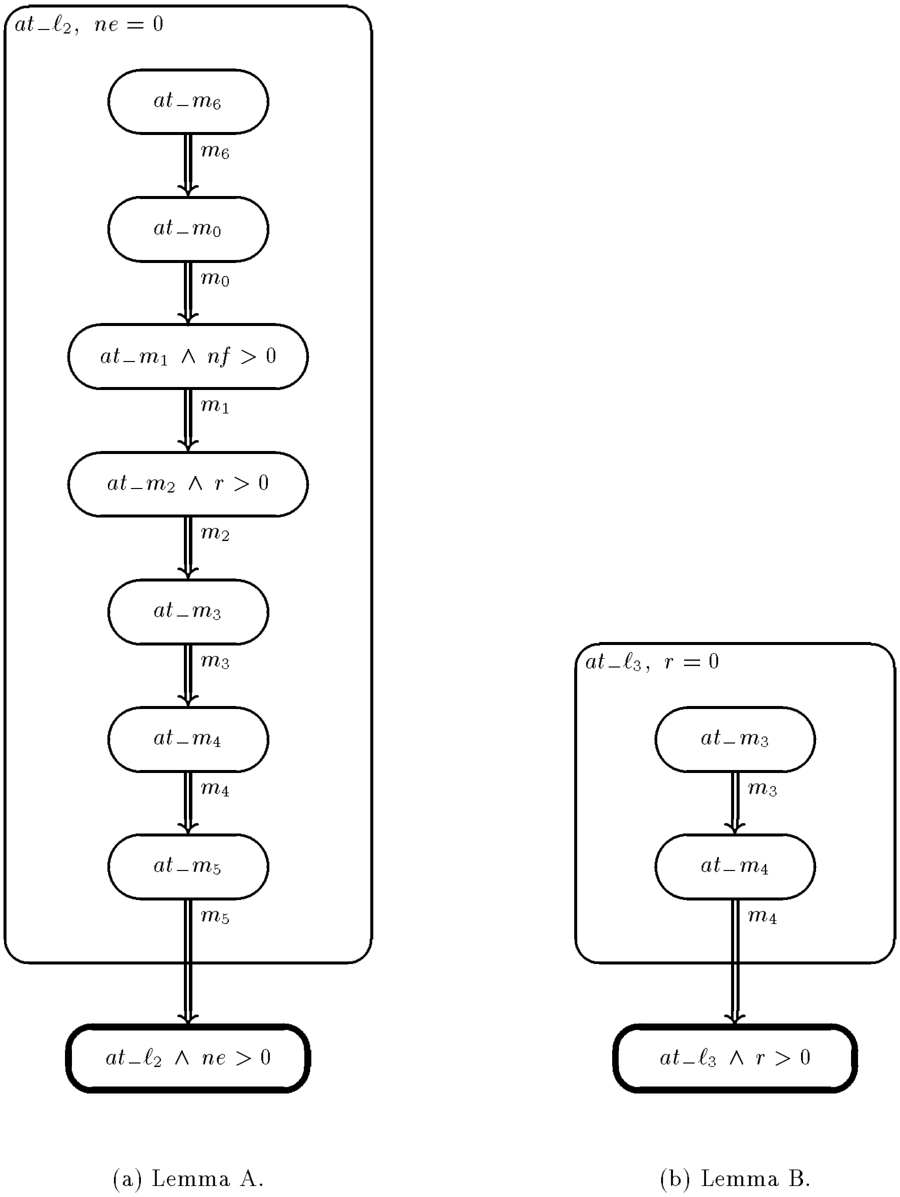

Fig. 15. Auxiliary CHAIN diagrams.