# On the Faithfulness of Formal Models *

**Zohar Manna**
Stanford University†
and
Weizmann Institute of Science
zm@cs.stanford.edu

**Amir Pnueli**
Weizmann Institute of Science‡
amir@wisdom.weizmann.ac.il

**Abstract**

The paper presents a critical examination of the way certain central aspects of concurrent programs are formally modeled. The main formal model examined is the operational model of *fair transition system* which represents concurrency by interleaving of actions considered *atomic*. Several questions concerning the faithfulness of this representation naturally arise. The paper considers some of these questions and, while presenting and analyzing some of the alternatives, attempts to justify the design decisions actually taken in the construction of the model.

## 1 Introduction

There is an obvious fact which, self-evident as it is, omission of its clear statement has caused no end of confusion and led to overheated arguments about the value and possible contribution of formal methods to program construction.

Stated bluntly, this fact is that the scope of validity of each of the formal and analytical methods proposed for program construction, such as formal specification, verification, and systematic development, is always a certain *mathematical model* of the execution of a program, rather than the real thing itself. It follows that, while these methods have absolute validity (assuming they are mathematically sound) over the model, their relevance to the actual running of the program is only as good as the degree of faithfulness to which the model represents real executions of the program.

*This research was supported in part by the National Science Foundation under grants CCR-89-11512 and CCR-89-13641, by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, by the United States Air Force Office of Scientific Research under contract AFOSR-90-0057, and by the European Community ESPRIT Basic Research Action Project 3096 (SPEC).

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

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

The observation that mathematical methods are fully valid only over mathematical models is so well recognized by the philosophers of science, and considered to be one of the most elementary postulates of modern science, that it is seldom stated explicitly. However, every novice engineer who ever uses mathematical techniques for validating his design is continuously aware of the fact that his method applies to a certain approximation of reality. In practice, this is often observed by the fact that, after precisely computing what is a sufficient width of a pillar that is expected to support a building, the designer multiplies this result by a "safety factor", just to be sure.

In some fields, the inherent limitations of each individual model are overcome by constructing and using a *hierarchy* of models, each a refinement of it predecessor. A well known example is the design of integrated circuits where one designs and simulates first (even verifies) at a functional level, then successively tests and refines his design at gate level, transistor level, and even sometimes at the level of electron flows. An obvious tradeoff exists between the degree of detail (faithfulness) incorporated in the model and the complexity of using it. As a result, one tends to analyze the complete system within the least detailed model, and to apply the more accurate analysis only to small parts of the system which, according to judgment, may be problematic.

Somehow, these obvious facts of life have not been always consistently considered and respected in the area of program construction. Perhaps the fact that a program is nothing but a term in some formal language, have misled some people to forget that real executions do not involve programs alone but also the hardware executing them and the environment in which it is embedded. As a result we may have seen some careless declarations such as:

> Formally verify your programs and you will never have to debug again!

There is no wonder that such overzealous (and clearly false) declarations caused some strong reactions. An example is [3], whose analysis has been interpreted by some to state that, since formal verification cannot guarantee absolute assurance, it is worthless. Such an extreme conclusion is obviously equally unacceptable. If we were to adopt it consistently in all areas of science and engineering, then no mathematical method would have survived.

As usual, the truth (as we see it) lies somewhere in between these two extreme positions. A conclusion about executions of programs, obtained by a formal method, is reliable only to the extent to which the underlying mathematical model is faithful. On the other hand, by developing more and more faithful models that account for a larger set of phenomena associated with real executions, we get closer (arbitrary close if so wished) to the thing itself.

Therefore, the activity of developing formal methodologies should consist not only of the construction of new models and the invention of new techniques for analyzing existing models, but also of a continuous assessment and comparison of models, both with one another and with the actual reality.

The domain of interest considered here is the family of *reactive programs* [4]. A reactive program is a program whose role is not to produce a final result, but to maintain some ongoing interaction with its environment. Examples of reactive programs are operating systems and programs controlling mechanical or chemical processes, such as a plane or a nuclear reactor. Some reactive programs are not expected to terminate. The concept of reactivity is strongly correlated with concurrency. To specify and analyze a reactive program we study the interaction between the program and its environment which run concurrently. Also, when specifying and analyzing a concurrent program, each process can be studies as a reactive component, where the other processes serve as its environment.

This paper presents an assessment and examination of the model of *fair transition system* proposed for modeling reactive and concurrent programs. This model has been formally introduced in [5], but has been used without formal definition in many other places. To examine the faithfulness of the model, we consider several phenomena that occur in real executions of programs, and inspect their representation in the model. In some of the cases, we consider several alternative options for representation, and attempt to justify the choice we have made.

An important and valuable guideline for the construction of useful models is the principle of *Occam's Razor*. In our context it can be formulated as a recommendation for choosing the simplest model that adequately represents the set of realistic phenomena one wishes to include. While this recommendation rarely identifies a unique model, it may considerably help in comparing several candidates.

## 2 A Model for Reactive Programs

Our model for concurrent programs is presented in two stages. The first stage presents the generic model of *fair transition system* [5], which is general enough to cover a multitude of concurrent systems with radically different syntactic presentations and communication mechanisms. In the second stage, we usually choose a particular syntax and a set of synchronization and communication mechanisms, to which we refer as a *concrete model*. The link between the generic and concrete models is established by mapping each abstract entity of the generic model to a corresponding construct in the concrete model.

The main concrete model we will study in detail is a programming language in which concurrent processes communicate by *shared variables*, and synchronize by *semaphores*. We will mention briefly additional concrete models, where communication is performed by *asynchronous* or *synchronous message-passing*, and refer the reader to [7] and [6] where some of these additional models are discussed in greater detail.

As we will see in the discussion, some issues of faithful representation are resolved by placing restrictions on the syntax of admissible programs in the concrete model. This is often an acceptable device for keeping the models simple.

### 2.1 Fair Transition System

To express the syntax of a basic transition system, we use an underlying first-order language. We refer to formulae in this language as *assertions*.

A *fair transition system* $\langle \Pi, \Sigma, \mathcal{T}, \Theta, \mathcal{J}, \mathcal{C} \rangle$, intended to represent a reactive program, is given by the following components:

- $\Pi = \{u_1, \ldots, u_n\}$ : A finite set of typed *state variables*.

Some of the state variables represent *data variables*, which are explicitly declared and manipulated by statements in the program. These variables range over data domains used in programs, such as booleans, integers, or lists. By default, whenever we do not state otherwise, all our data variables are assumed to be integers. Other variables are *control variables*, which represent progress in the execution of the program; they may range over locations in the program. The *type* of each variable indicates the *domain* over which the variable ranges.

- $\Sigma$ : A set of *states*.

Each state $s$ in $\Sigma$ is an interpretation of $\Pi$, assigning to each variable $u$ in $\Pi$ a value over its domain, which we denote by $s[u]$.

- $\mathcal{T}$ : A finite set of *transitions*.

Each transition $\tau$ in $\mathcal{T}$ represents a state-transforming *atomic action* of the system and is defined as a function $\tau$: $\Sigma \rightarrow 2^{\Sigma}$ that maps a state $s$ in $\Sigma$ into the (possibly empty) set of states $\tau(s)$ that can be obtained by applying action $\tau$ to state $s$. Each state $s'$ in $\tau(s)$ is defined to be a $\tau$*-successor* of $s$. If $\tau(s) \neq \phi$, we say that $\tau$ is *enabled* on $s$; otherwise, $\tau$ is *disabled* on $s$. It is required that, for each $s \in \Sigma$, there exists a transition $\tau \in \mathcal{T}$ that is enabled on $s$.

- $\Theta$ : An assertion called the *initial condition*.

Assertion $\Theta$ characterizes the states at which execution of the program can begin. A state $s$ satisfying $\Theta$, i.e., $s \models \Theta$, is called an *initial state*.

- $\mathcal{J} \subseteq \mathcal{T}$ : A set of transitions called the *justice set*.

We refer to each transition $\tau \in \mathcal{J}$ as a *just transition*.

- $\mathcal{C} \subseteq \mathcal{T}$ : A set of transitions called the *compassion set*.

We refer to each transition $\tau \in \mathcal{T}$ as a *compassionate transition*.

### 2.2 The Transition Relation $\rho_\tau$

An effective representation of the state-transformation performed by each transition $\tau$ is provided by an assertion, called the *transition relation* for $\tau$ and denoted by $\rho_\tau(\Pi,\ \Pi')$.

Assertion $\rho_\tau$ relates the values of the state variables in a state $s$ to their values in a successor state $s'$ obtained by applying $\tau$ to $s$. The assertion refers to the state variables using two copies of $\Pi = \{u_1, \ldots, u_n\}$. An occurrence of $u$ refers to the value of $u$ in $s$, while an occurrence of $u'$ refers to the value of $u$ in $s'$. We call $u'$ the *primed version* of $u$ and denote the set of primed versions of all the state variables by $\Pi' = \{u'_1, \ldots, u'_n\}$.

For a transition $\tau$ associated with transition relation $\rho_\tau$, we define state $s'$ to be a $\tau$-successor of state $s$ if $\langle s, s' \rangle \models \rho_\tau$, where, for each $u \in \Pi$, the joint interpretation $\langle s, s' \rangle$ inerprets $u$ as $s[u]$ and interprets $u'$ as $s'[u]$. We restrict our attention to transitions whose state-transformation is defined by $\rho_\tau$ as described above.

Consider, for example, a transition $\tau$ whose transition relation is given by

$$\rho_\tau: \quad even(x) \ \wedge \ x+1 \leq x' \leq x+2,$$

where $x$ is an integer state variable. This transition is enabled only on states in which $x$ is even, and $s'$ is asuccessor of $x$ only if the value of $x$ in $s'$ equals to the value of $x+1$ or to the value of $x+2$ in $s$.

Normally, each transition modifies only few of the state variables and leaves the others unmodified. Consequently, a typical transition relation will have a conjunct of the form $u' = u$ for each unmodified variable. To simplify the presentation of transition relations we adopt a convention by which a variable $u$ such that $u'$ does not appear explicitly in $\rho_\tau$ is considered unmodified, and it is as though the conjunct $u' = u$ is automatically added to $\rho_\tau$.

For example, if $\Pi = \{x, y, z\}$ and $\rho_\tau : z' = x + y$, then $z$ is the only variable modified by $\tau$, while $x$ and $y$ are preserved.

### 2.3 Computations

Consider an infinite sequence of states over $\Pi$

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

For a given transition $\tau$, we say that $\tau$ is *continually enabled beyond position* $j$ in $\sigma$ if $\tau$ is enabled on $s_k$, for every $k \geq j$. We say that $\tau$ is *enabled infinitely many times in* $\sigma$ if $\tau$ is enabled on $s_k$ for infinitely many $k$'s.

Let $P$ be the fair transition system $\langle \Pi, \Sigma, \mathcal{T}, \Theta, \mathcal{J}, \mathcal{C} \rangle$. A sequence $\sigma$ is defined to be a *computation* of $P$ if $\sigma$ satisfies the following requirements:

- *Initiation*: The first state $s_0$ is initial, i.e., $s_0 \models \Theta$.
- *Consecution*: For each pair of consecutive states $s_i, s_{i+1}$ in $\sigma$, $s_{i+1} \in \tau(s_i)$ for some transition $\tau$ in $\mathcal{T}$. That is, $s_{i+1}$ is a $\tau$-*successor* of $s_i$. In this case we say that $\tau$ is *taken* at position $i$. Note that it is possible for more than one transition to be considered as taken at a given position.
- *Justice* (also known as *weak fairness*): For each $\tau$ in $\mathcal{J}$ it is not the case that $\tau$ is *continually* enabled beyond some position in $\sigma$ but taken only finitely many times.
- *Compassion* (also known as *srong fairness*): For each $\tau$ in $\mathcal{C}$ it is not the case that $\tau$ is enabled *infinitely many times* in $\sigma$ but taken only finitely many times.

We often display computations as a sequence of states connected by arrows that are labeled by the transitions that cause the system to move to the next state. Thus, a presentation of a computation as

$$s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} s_2 \longrightarrow \cdots$$

provides, in addition to the fact that $s_0, s_1, \ldots$ is a computation, also the identity of the transition $\tau_i \in \mathcal{T}$ leading from $s_i$ to $s_{i+1}$.

Consider, for example, a (fair) transition system with state variables $\Pi = \{x\}$, transitions $\{\tau_1, \tau_2\}$, whose transition relations are given by

$$\rho_1: \ even(x) \ \wedge \ x+1 \leq x' \leq x+2 \quad \text{and} \quad \rho_2: \ odd(x) \ \wedge \ x' = x+1,$$

respectively. Assume that $\Theta : x = 0$, the justice set is $\mathcal{J} = \{\tau_1, \tau_2\}$, and the compassion set is $\mathcal{C} = \phi$.

Then, the following sequence which periodically takes $\tau_1$ twice and then $\tau_2$ once is an example of a computation:

$$\langle x:0 \rangle \xrightarrow{\tau_1} \langle x:2 \rangle \xrightarrow{\tau_1} \langle x:3 \rangle \xrightarrow{\tau_2} \langle x:4 \rangle \xrightarrow{\tau_1} \langle x:6 \rangle \xrightarrow{\tau_1} \cdots .$$

## 3 A Programming Language: Syntax

As our main concrete model, we introduce a simple programming language in which processes communicate by shared variables. For each presented statement, we provide some intuitive explanation of its intended meaning.

- *Assignment*: For $y_1, \ldots, y_k$ a list of variables and $e_1, \ldots, e_k$ a list of expressions of corresponding types,

$$(y_1, \ldots, y_k) \ := \ (e_1, \ldots, e_k)$$

is an *assignment* statement. For the case that $k = 1$, we write simply $y := e$. The case $k = 0$ modifies no data variables, and is written as **skip**.

- *Await*: For $c$ a boolean expression,

$$\textbf{await } c$$

is an *await* statement. We refer to condition $c$ as the *guard* of the statement.

Execution of **await** $c$ changes no variables. Its sole purpose is to wait until $c$ becomes true, at which point it terminates.

- *Conditional*: For $S_1$ and $S_2$ statements and $c$ a boolean expression,

$$\textbf{if } c \textbf{ then } S_1 \textbf{ else } S_2$$

is a *conditional* statement. Its intended meaning is that the boolean expression $c$ is evaluated and tested. If the condition evaluates to $\mathrm{T}$, statement $S_1$ is selected for subsequent execution; otherwise, $S_2$ is selected. Thus, the first step in an execution of the *conditional* statement is the evaluation of $c$ and the selection of $S_1$ or $S_2$ for further execution. Subsequent steps continue to execute the selected substatement.

We abbreviate **if** $c$ **then** $S_1$ **else skip** to **if** $c$ **then** $S_1$.

- *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 the 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 $S_2, S_3, \ldots, S_k$.

With *concatenation*, we can define the *when* statement **when** $c$ **do** $S$ as an abbreviation for the concatenation **await** $c$; $S$.

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

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

is a *selection* statement. Its intended meaning is that, as a first step, one of $S_1, \ldots, S_k$, which is currently enabled, is selected and the first step in the selected statement is executed. Subsequent steps proceed to execute the rest of the selected substatement. If more than one of $S_1, \ldots, S_k$ is enabled, the selection is non-deterministic.

The *selection* statement is often applied to *when* statements. This combination leads to *conditional selection*. For example, the general conditional command of the *guarded command* language (proposed by Dijkstra [1], [2]), of the form

$$\textbf{if} \;\; c_1 \rightarrow S_1 \;\Box\; c_2 \rightarrow S_2 \;\Box\; \ldots \;\Box\; c_k \rightarrow S_k \;\; \textbf{fi}$$

can be represented in our language by a *selection* statement formed out of *when* statements:

$$[\textbf{when}\ c_1\ \textbf{do}\ S_1] \;\; \textbf{or} \;\; [\textbf{when}\ c_2\ \textbf{do}\ S_2] \;\; \textbf{or} \;\; \cdots \;\; \textbf{or} \;\; [\textbf{when}\ c_k\ \textbf{do}\ S_k].$$

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

  **while** $c$ **do** $S$

is a *while* statement. Its execution begins by evaluating $c$. If $c$ is found to be false, execution of the statement terminates. If $c$ is found true, subsequent steps proceed to execute $S$. When $S$ terminates, $c$ is tested again.

- *Request*: For $y$ an integer variable,

  **request**$(y)$

is a *request* statement. This is one of the two semaphore statements. It is enabled only when $y$ is positive, and its execution subtracts 1 from $y$. As we will show below, the semaphore statements are used for synchronization between processes.

- *Release*: For $y$ an integer variable,

  **release**$(y)$

is a *release* statement, which is the second semaphore statement. This statement is enabled whenever control is at its front, and its execution adds 1 to $y$. In that it is equivalent to the assignment $y := y + 1$.

### 3.1 Programs

A program $P$ has the form

$$P :: \;\; \Big[\text{declaration}; \;\; [P_1 :: S_1 \,\|\, \cdots \,\|\, P_m :: S_m]\Big],$$

where $P_1 :: S_1, \ldots, P_m :: S_m$ are *named processes*. Each $S_i$ is a statement, and $P_i$ is a name for the process. The names of the program and of the processes are optional, and may be omitted.

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

mode variable, $\ldots$, variable: type **where** $\varphi$.

The mode of each declaration statement may be **in**, **local**, or **out** identifying variables that are inputs to the program, local, or outputs of the program. Declaration statements with no explicit mode specification retain the mode of the preceding statement.

The list

variable, ..., variable: type,

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

The assertion $\varphi$, which appears in a declaration statement, imposes constraints on the initial values of some of the variables declared in this statement. Assertion $\varphi$ may be omitted from a declaration statement if no constraint is imposed on 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.

### 3.2 Labels and Locations in Programs

Each statement in a program may be labelled. It is required that the labels appearing in the program are distinct. In Fig. 1 we present program GCD which computes the greatest common divisor of the inputs $x_1$ and $x_2$. All statements in this program are labelled. In principle, we consider each program to be fully labelled, and denote the label of statement $S$ by $label(S)$. In actual presentations of programs, we often omit many of the labels.

$$
\begin{array}{lll}
\textbf{in} & x_1,\ x_2 & : \textbf{integer where } x_1 > 0,\ x_2 > 0 \\
\textbf{local} & y_1,\ y_2 & : \textbf{integer where } y_1 = x_1,\ y_2 = x_2 \\
\textbf{out} & g & : \textbf{integer}
\end{array}
$$

$$
\ell_0: \left[\begin{array}{l}
\ell_1: \left[\begin{array}{l}
\textbf{while } y_1 \neq y_2 \textbf{ do} \\
\ell_2: \left[\begin{array}{l}
\ell_3: \textbf{ when } y_1 > y_2 \textbf{ do } [\ell_4:\ y_1 := y_1 - y_2] \\
\quad \textbf{or} \\
\ell_5: \textbf{ when } y_2 > y_1 \textbf{ do } [\ell_6:\ y_2 := y_2 - y_1]
\end{array}\right]
\end{array}\right] \\
\ell_7:\ [g := y_1]
\end{array}\right]
$$

Fig. 1. A fully labeled GCD program.

Labels in programs have two important roles. The first is to provide a unique identification and reference to the statements. Thus we may talk about *when* statement $\ell_3$ whose body is assignment $\ell_4$.

The second important role of the labels is to serve as possible *sites* of control. Thus, we may consider a state in which control resides at label $\ell_2$ in the program of Fig. 1 and, when finding that $y_1 > y_2$, execution moves to $\ell_4$ where it pauses before subtracting $y_2$ from $y_1$.

The problem with interpreting labels as unique sites of control is that there are too many of them, and we do not necessarily wish to distinguish between them. For example, it is clear that when control is at $\ell_2$ it is also at $\ell_3$ and at $\ell_5$, because any of these statments can be immediately executed if its guard is true. Thus, we define a *location* to be an *equivalence class* of labels, where the equivalence is defined by the rules:

$$\begin{array}{lll} label([S_1; \cdots ; S_k]) & \sim_L & label(S_1) \quad \text{and} \\ label([S_1 \textbf{ or } \cdots \textbf{ or } S_k]) & \sim_L & label(S_i) \quad \text{for each } i = 1, \ldots, k. \end{array}$$

For a label $\ell$, we denote by $[\ell]$ the location corresponding to $\ell$.

# 4 A Programming Language: Semantics

In order to establish the correspondence between the presented programming language and the generic model of fair transition systems, we have to map each of the components of a fair transition system to a construct in the programming language.

## 4.1 State Variables and States

For the programming language, the *state variables* $\Pi$ consist of the *data variables* $Y = \{y_1, \ldots, y_n\}$ that are explicitly declared and manipulated by the program, and a single *control variable* $\pi$. The data variables $Y$ include the input, output, and local variables, and range over their respectively declared data domains. The control variable $\pi$ ranges over *sets of locations*. The value of $\pi$ in a state denotes all the locations of the program in which control currently resides.

As states we take all possible interpretations that assign to the state variables values over their respective domains.

Consider, for example, the program

$$\left[\begin{array}{c} \textbf{out } x\text{: } \textbf{integer where } x = 0 \\ [\ell_1\text{: } x := x + 1] \quad || \quad [\ell_2\text{: } x := x + 2] \end{array}\right].$$

The state variables for this program are $\Pi = \{\pi, x\}$. While there are many possible states, only four of them can ever arise in a computation of this program. These are:

$$\langle \pi : \{\ell_1, \ell_2\},\ x : 0\rangle \quad \langle \pi : \{\ell_2\},\ x : 1\rangle \quad \langle \pi : \{\ell_1\},\ x : 2\rangle \quad \langle \pi : \{\},\ x : 3\rangle$$

## 4.2 Transitions

For each statement $S$ in a program, we define a set of labels, denotes by $tloc(S)$, which represents the location at which execution resides after $S$ is terminated. This set of labels is defined inductively in a top-down manner as follows:

| | | |
|---|---|---|
| For $S$ a *process* of the program, | $tloc(S) = \phi$ | |
| For $S = [S_1; \cdots; S_k]$, | $tloc(S_k) = tloc(S)$ | |
| For $[\ell_1 : S_1; \cdots; \ell_k : S_k]$, | $tloc(S_i) = [\ell_{i+1}]$, | for each $i = 1, \ldots, k-1$ |
| For $S =$ **if** $c$ **then** $S_1$ **else** $S_2$, | $tloc(S_1) = tloc(S_2) = tloc(S)$ | |
| For $S = [S_1$ **or** $\cdots$ **or** $S_k]$, | $tloc(S_i) = tloc(S)$, | for each $i = 1, \ldots, k$ |
| For $\ell$: **while** $c$ **do** $\widetilde{S}$, | $tloc(\widetilde{S}) = [\ell]$ | |

For a statement $\ell : S$, we often denote $tloc(S)$ by $tloc(\ell)$.

To ensure that every state has some transition enabled on it, we standardly include in the transition system corresponding to each program the idling transition $\tau_I$, whose transition relation is

$$\rho_I\text{:}\ \ \mathrm{T}.$$

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

- *Assignment*: With the statement $\ell : (y_1, \ldots, y_k) := (e_1, \ldots, e_k)$, we associate a transition $\tau_\ell$, with the transition relation

$$\rho_\ell\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ (\pi' = \pi - [\ell] \cup tloc(\ell)) \ \wedge\ (y_1' = e_1) \wedge \cdots \wedge (y_k' = e_k).$$

- *Await*: With the statement $\ell$: **await** $c$, we associate a transition $\tau_\ell$, with the transition relation

$$\rho_\ell\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ c \ \wedge\ (\pi' = \pi - [\ell] \cup tloc(\ell)).$$

The transition $\tau_\ell$ is enabled only when control is at $\ell$ and the condition $c$ holds. When taken, it moves from $\ell$ to the location $tloc(\ell)$.

- *Conditional*: With the statement $\ell$: **if** $c$ **then** $[\ell_1\text{:}\, S_1]$ **else** $[\ell_2\text{:}\, S_2]$, we associate two transitions, $\tau_\ell^{\mathrm{T}}$ and $\tau_\ell^{\mathrm{F}}$ with transition relations

$$\rho_\ell^{\mathrm{T}}\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ \ c \ \wedge\ (\pi' = \pi - [\ell] \cup [\ell_1])$$
$$\rho_\ell^{\mathrm{F}}\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ \neg c \ \wedge\ (\pi' = \pi - [\ell] \cup [\ell_2]).$$

Relation $\rho_\ell^{\mathrm{T}}$ corresponds to the case that $c$ evaluates to true and execution proceeds to $\ell_1$, while $\rho_\ell^{\mathrm{F}}$ corresponds to the case that $c$ is false and execution proceeds to $\ell_2$.

- *While*: With the statement $\ell$: [**while** $c$ **do** $[\widetilde{\ell}\text{:}\, \widetilde{S}\,]]$, we associate two transitions, $\tau_\ell^{\mathrm{T}}$ and $\tau_\ell^{\mathrm{F}}$ with transition relations

$$\rho_\ell^{\mathrm{T}}\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ \ c \ \wedge\ (\pi' = \pi - [\ell] \cup [\widetilde{\ell}])$$
$$\rho_\ell^{\mathrm{F}}\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ \neg c \ \wedge\ (\pi' = \pi - [\ell] \cup tloc(\ell)).$$

Note that in the case of a true $c$ control moves, according to $\rho_\ell^{\mathrm{T}}$, from $\ell$ to $\widetilde{\ell}$, while in the case of a false $c$ it moves, according to $\rho_\ell^{\mathrm{F}}$, from $\ell$ to $tloc(\ell)$.

- *Request*: With the statement $\ell$: **request**$(y)$, we associate a transition $\tau_\ell$, with the transition relation

$$\rho_\ell\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ x > 0 \ \wedge\ (\pi' = \pi - [\ell] \cup tloc(\ell)) \ \wedge\ y' = y - 1.$$

- *Release*: With the statement $\ell$: **release**$(y)$, we associate a transition $\tau_\ell$, with the transition relation

$$\rho_\ell\text{:}\quad ([\ell] \subseteq \pi) \ \wedge\ (\pi' = \pi - [\ell] \cup tloc(\ell)) \ \wedge\ y' = y + 1.$$

The transitions associated with the semaphore statements *request* and *release*, are called *semaphore transitions*. We refer to the transitions corresponding to statements of the program, i.e., all transitions except for $\tau_I$, as the *diligent* transitions.

### 4.3 The Initial Condition

Consider a program

$$\Big[\text{declaration; } [P_1 :: \ [\ell_1\!: \ S_1] \ \| \ \cdots \ \| \ P_m :: \ [\ell_m\!: \ S_m]]\Big].$$

Let $\varphi$ denote the *data precondition* of the program. We define the *initial condition* $\Theta$ for program $P$ 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 set to the initial locations of the processes, *after* the initialization of the local and output variables has been performed.

### 4.4 Justice and Compassion

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

- *Justice*: As the *justice* set, we take all the diligent transitions, i.e. $\mathcal{T} - \{\tau_I\}$.
- *Compassion*: As the *compassion* set, we take all the semaphore transitions.

## 5 Assessment of the Interleaving Model

An essential element of the generic model of transition systems is that concurrency is represented by *interleaving*. This means that, according to the formal model, two parallel processes never execute their statements at precisely the same instant, but take turns in executing atomic transitions. Formally, when one of them executes an atomic transition, the other is inactive. This model of computation is very convenient for the formalization, analysis, and manipulation of concurrent programs.

However, actual concurrent systems are usually composed of several independent processors, each of them executing a program of its own (a *process* in our terminology). In such systems, the execution of statements in different processors usually overlap rather than interleave. We refer to this behavior as *overlapped execution*.

A crucial question is how to reconcile the formal notion of *interleaved computations*, defined in our models, and the notion of *overlapped executions*, as realized on actual systems.

Two problems have to be resolved to achieve this reconciliation. The first problem is that of *interference*, and the second is *independent progress*.

- *Interference*

Interleaved execution provides a higher degree of protection from interference than is available in overlapped execution. This is because interleaved execution requires that when a transition is taken, all other transitions are inactive, so no interference *during* a transition is possible.

Consider, for example, the following statement:

$$\textbf{when } y = y \textbf{ do } S.$$

In interleaved computations, the condition $y = y$ is tested in one atomic step, so it always yields the value T. In comparison, overlapped execution (under naive implementation, with no optimization) of this statement will reference $y$ twice. If, precisely between these two references, an overlapped statement performed by a concurrent processor changes $y$, the testing processor may find the condition to be false.

- *Independent Progress*

The problem of *independent progress* is that, in an overlapped execution, the computation of each process keeps advancing, since each processor is independently responsible for its own progress. In an interleaved computation, the only requirement is that enabled transitions be continuously chosen and executed. There is nothing to disallow a computation in which only transitions from one process are ever chosen. Such a computation ensures progress of the preferred process, but keeps all other processes stagnant.

Fair transition systems solve this problem by the two requirements of *justice* and *compassion*. We will discuss below the properties of each of these fariness requirements, and their implications.

## 5.1 Interleaving and Concurrency

Consider the following simple program, to which we refer as $A1$

$$\left[\begin{array}{c} \textbf{out } y\text{: } \textbf{integer where } y = 1 \\ P_1 :: \quad [\ell_0\text{: } y := y + 1] \quad \| \quad P_2 :: \quad [m_0\text{: } y := y - 1] \end{array}\right]$$

The overlapped execution of this program on a system consisting of two independent processors sharing a common memory yields $\{0, 1, 2\}$ as the set of possible outcomes for $y$. To see this, realize that execution of an *assignment* statement, such as $y := y + 1$, usually consists of three distinct steps:

- *Fetch Step*: The value in the shared-memory location corresponding to $y$ is fetched and stored in a local register.
- *Compute Step*: The register is incremented, with the resulting value possibly being deposited in another local register.
- *Store Step*: The value in the result register is stored in the shared-memory location corresponding to $y$.

The partition of execution of an assignment into these three steps is common to many other statements. More complicated assignments may have several fetch steps, depending on the number of variables appearing in their right-hand sides.

When we consider program $A1$, there are such steps in the execution of assignments by both processes $P_1$ and $P_2$. We identify four of these steps as *critical events*, in the sense that the relative ordering between them determines the final outcome of the execution. They are the events of fetching and storing values from and to the shared-memory location corresponding to $y$. We describe the critical events of this program as:

$P_1$ reads value $m$ from $y$, denoted by $r_1(m)$

$P_1$ writes value $m$ to $y$, denoted by $w_1(m)$

$P_2$ reads value $m$ from $y$, denoted by $r_2(m)$

$P_2$ writes value $m$ to $y$, denoted by $w_2(m)$.

Furthermore, we assume that these critical events are *atomic*, in the sense that they cannot overlap. For simple data types, this is usually guaranteed by the underlying hardware: while one process is accessing a memory location, no other process may access the same location.

In an overlapped execution of program $A1$, these events can occur in several orders, some of which are listed below.

$$
\begin{array}{lll}
E_1: & r_1(1),\ w_1(2),\ r_2(2),\ w_2(1), & \text{yielding } y = 1 \\
E_2: & r_1(1),\ r_2(1),\ w_1(2),\ w_2(0), & \text{yielding } y = 0 \\
E_3: & r_1(1),\ r_2(1),\ w_2(0),\ w_1(2), & \text{yielding } y = 2.
\end{array}
$$

This shows how an overlapped execution of this program can yield the three possible results $0, 1, 2$, depending on the relative ordering of the critical events of the execution. Note that we do not consider the calculation step as critical, since it operates only on internal registers that are not accessible to the other process.

The situation illustrated here is general. In any concurrent program it is always possible to identify some events in the execution of the program as *critical events*, such that their relative ordering in time uniquely determines the final result of the execution as well as its observable behavior, i.e., the sequence of values assumed by the observed variables. For programs that communicate by shared variables, these events are the reading and writing to the shared variables.

Consider now the possible results of computations of program $A1$, as defined in our model of fair transition system. The computational model assigns to program $A1$ two transitions $\ell_0$ and $m_0$ (shorthand notation for $\tau_{\ell_0}$ and $\tau_{m_0}$) corresponding to the *assignment* statements in $P_1$ and $P_2$. Thus, the only possible computations of program $A1$ in our interleaved model are the following:

$$
\begin{array}{l}
\langle\{\ell_0, m_0\},\ 1\rangle \xrightarrow{\ell_0} \langle\{m_0\},\ 2\rangle \xrightarrow{m_0} \langle\{\},\ 1\rangle\ \cdots \\
\langle\{\ell_0, m_0\},\ 1\rangle \xrightarrow{m_0} \langle\{\ell_0\},\ 0\rangle \xrightarrow{\ell_0} \langle\{\},\ 1\rangle\ \cdots .
\end{array}
$$

In the interleaved model only one final result is possible: $y = 1$.

This seems to indicate that interleaved computation fails to capture the full range of behaviors exhibited by overlapped execution.

However, the problem is not in the interleaving model per se, but in the assignment of atomic transitions to statements in the above program. According to the rules we gave for our programming language, each *assignment* statement is represented as a single atomic transition. This representation leads to the undesirable effect that a transition such as $\ell_0$ (corresponding to $y := y + 1$) forces the two critical events $r_1$ and $w_1$ to happen in one step, thereby precluding the possibility of any of the critical events $r_2$, $w_2$ occurring after $r_1$ but before $w_1$. Indeed it is these possibilities that led to the outcomes 0 and 2, which the interleaved computation failed to produce.

One possible solution to this difficulty is that an assignment such as $y := y + 1$ in program $A1$ should be associated with *two* atomic transitions, say $\tau'_{\ell_0}$ and $\tau''_{\ell_0}$. The first transition $\tau'_{\ell_0}$ should perform the fetch substep, while $\tau''_{\ell_0}$ should perform the store substep. The calculation substep can be absorbed into either $\tau'_{\ell_0}$ or $\tau''_{\ell_0}$.

### 5.2 Finer Granularity

There is another solution to the problem of discrepancy between interleaved computation and overlapped execution. This solution, which is the one we follow, is to modify program $A1$ in such a way that any atomic transition generated according to the rules as originally given, i.e., one transition per *assignment* statement, contains at most *one* critical event. Such a modification is presented by the following program $A2$:

$$\left[\begin{array}{c} \textbf{out } y\textbf{: integer where } y = 1 \\ P_1 :: \left[\begin{array}{l} \textbf{local } t_1\textbf{: integer} \\ \ell_0\text{: } t_1 := y \\ \ell_1\text{: } y := t_1 + 1 \end{array}\right] \quad || \quad P_2 :: \left[\begin{array}{l} \textbf{local } t_2\textbf{: integer} \\ m_0\text{: } t_2 := y - 1 \\ m_1\text{: } y := t_2 \end{array}\right] \end{array}\right]$$

Each of the assignments of program $A1$ has been broken into two successive assignments in $A2$. The first assignment in each sequence performs the fetch substep, while the second assignment performs the store substep. To emphasize that the exact timing of the calculation substep is immaterial (as long as it comes after fetching and before storing), we have included it in the storing assignment in $P_1$ and in the fetching assignment in $P_2$. Note that local variables $t_1$ and $t_2$ represent the internal registers of the two independent processors.

The interleaved computations of $A2$ recreate all the overlapped executions of $A2$, which in turn are identical to the overlapped executions of program $A1$. To see this, we illustrate below three computations of $A2$ leading to the final results 1, 0, and 2, respectively. We represent the states occurring in these computations by listing the values assumed by the variables $\pi$, $y$, $t_1$ and $t_2$.

$$\begin{aligned} \sigma_1\text{:}\quad & \langle\{\ell_0, m_0\},\ 1,\ -,\ -\rangle \xrightarrow{\ell_0} \langle\{\ell_1, m_0\},\ 1,\ 1,\ -\rangle \xrightarrow{\ell_1} \\ & \langle\{m_0\},\ 2,\ 1,\ -\rangle \xrightarrow{m_0} \langle\{m_1\},\ 2,\ 1,\ 1\rangle \xrightarrow{m_1} \langle\{\},\ 1,\ 1,\ 1\rangle \cdots, \end{aligned}$$

yielding $y = 1$.

$$\sigma_2\colon\quad \langle\{\ell_0, m_0\},\ 1,\ -,\ -\rangle \xrightarrow{\ell_0} \langle\{\ell_1, m_0\},\ 1,\ 1,\ -\rangle \xrightarrow{m_0}$$
$$\langle\{\ell_1, m_1\},\ 1,\ 1,\ 0\rangle \xrightarrow{\ell_1} \langle\{m_1\},\ 2,\ 1,\ 0\rangle \xrightarrow{m_1} \langle\{\},\ 0,\ 1,\ 0\rangle \cdots,$$

yielding $y = 0$.

$$\sigma_3\colon\quad \langle\{\ell_0, m_0\},\ 1,\ -,\ -\rangle \xrightarrow{\ell_0} \langle\{\ell_1, m_0\},\ 1,\ 1,\ -\rangle \xrightarrow{m_0}$$
$$\langle\{\ell_1, m_1\},\ 1,\ 1,\ 0\rangle \xrightarrow{m_1} \langle\{\ell_1\},\ 0,\ 1,\ 0\rangle \xrightarrow{\ell_1} \langle\{\},\ 2,\ 1,\ 0\rangle \cdots,$$

yielding $y = 2$.

This shows that, by proper restrictions on the syntax of programs we allow, it is possible to achieve a better matching between interleaved computations and overlapped executions.

Due to shortage of space, we will not present additional examples. However, the full version of this paper discusses many additional examples of discrepancies between the formal model and some phenomena arising in real executions, and how they can be resolved.

## References

[1] E. Dijkstra. Guarded commands, nondeterminancy, and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[2] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, New Jersey, 1976.

[3] J. Fetzer. Program verification: The very idea. *Comm. ACM*, 31:420–422, 1988.

[4] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[5] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proc. 10th ACM Symp. Princ. of Prog. Lang.*, pages 141–154, 1983.

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

[7] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 510–584. Lec. Notes in Comp. Sci. 224, Springer-Verlag, 1986.