Temporal Specification and Verification of Reactive Modules *

Zohar Manna
Stanford University †
and
Weizmann Institute of Science‡

Amir Pnueli
Weizmann Institute of Science‡

March 3, 1992

Abstract. The paper addresses the problem of compositional specification and verification of reactive modules by temporal logic. We consider programs formed by parallel composition of *modules* that communicate by either *shared variables* or *message passing*. Each module specifies its *interface* with other modules by identifying the communication variables or channels on which it may *write* and those from which it may only *read*.

We introduce the notion of *modular validity* by which a formula φ is modularly valid over a module $\mathcal{M}$ if it is satisfied by any system in which $\mathcal{M}$ is a component. This leads to a simple proof rule in which, if φ_1 is modularly valid over $\mathcal{M}_1$ and φ_2 is modularly valid over $\mathcal{M}_2$, then $\varphi_1 \wedge \varphi_2$ is (modularly) valid over their parallel composition.

We show how to prove modular validity, and give examples of *compositional proofs* using this approach, as well as suggestions for decomposing a *global specification* of an entire program into a set of *modular specifications* for its components.

Keywords: Temporal logic, modularity, reactive module, modular validity, compositional proof, global specification, modular specification.

*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 NAG2-703, by the United States Air Force Office of Scientific Research under contract AFOSR-90-0057, by the European Community ESPRIT Basic Research Action Project 3096 (SPEC), and by a gift from Mitsubishi.

†Department of Computer Science, Stanford University, Stanford, CA 94305
‡Department of Applied Mathematics, Weizmann Institute, Rehovot, Israel

Contents

1 Introduction

This paper presents a modular approach to the specification and verification of reactive systems, using temporal logic.

Many of the developments of temporal proof systems for reactive systems, such as [MP81] and [OL82], were based on a non-compositional approach. This global approach adopts a *closed system* view of reactive programs. According to this view, the complete program is available for specification and analysis. If necessary, the environment is also modeled as a (usually highly non-deterministic) component of the presented system.

The global approach to the specification and verification of reactive systems has often been criticized because of its lack of compositionality. Some of the obvious criticism is that it only allows post-facto verification of completed systems and does not support *rigorous development* of systems in which components are constructed piecemeal and each component verified as soon as it is constructed. Another important consideration is that compositional approaches partition the effort into smaller subtasks and are therefore more conducive to dealing with very large programs.

In response to this criticism, there have been several suggestions for a compositional approach to the temporal analysis of reactive programs. Representatives of these suggestions are [BKP84], [NGO85], and [AL90].

In fact, this paper can be described as a direct descendant and elaboration of the approach presented in [BKP84] (and some of its extensions discussed in [BKP85]). However, there are some differences between these two proposals for a compositional approach to temporal reasoning.

The approach presented in this paper can be described as *reluctant* or *coarse granularity* compositionality. By this we mean that certain syntactic components of the program are identified as *modules*. The temporal specification and analysis of each module is done in the traditional global style. But when dealing with composition of such modules, we use compositional reasoning for combining the specification of individual modules to infer properties of the complete system. Thus, the proof methodology we propose can be described as a two level system. On the intra-module level we use global reasoning, while on the inter-module level we use compositional approaches.

This two level approach is not new. One can view the system proposed in [NGO85] as being a two level system. Typical to data-flow networks, a system consists of several nodes with communicating channels. The specification of each individual node is derived and developed by other means, sometimes not even specified in the description of the system. The compositional part only tells us how to combine properties of subsystems, starting with individual nodes, into properties of larger subsystems.

It is also possible to describe the proof system for Unity programs [CM88] as two level. The properties of a single module are derived by a global rule that requires proving a verification condition for each transition contained in the module. Then there are composition rules for deriving the properties of combined modules.

From our point of view, this paper represents a particular didactic approach to the presentation of temporally based proof systems. As a first stage we recommend the introduction of the global closed-system approach through the notions of fair transition systems, their computations and the verification of temporal properties of such systems.

We then recommend a second stage that introduces modular reasoning about systems as an elaboration and refinement of the methods developed in the first stage.

We refer the reader to [dR85] and [HdR86] for additional discussions concerning compositionality and modularity.

The paper is organized as follows: in Section 2 we recall the basic notions of the global approach as presented, for example, in [MP91a]. In Section 3 we introduce the concept of a module and the notions of modular validity, open computations, and the approach for verifying modular validity. In Section 4 we consider the composition of specifications of parallel modules, and approaches to the decomposition of a global specification into several modular specifications. In Section 5 we present a complete example of a global specification and its decomposition into modular parts.

2 Programs, Computations, and Temporal Logic

Our model for concurrent programs is presented in two stages. The first stage presents the generic model of *fair transition systems* [MP83], which is general enough to cover a multitude of concurrent systems with radically different syntactic presentations and communication mechanisms. In the second stage, we 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 briefly mention additional statements that allow communication by *asynchronous* or *synchronous message-passing.*

The basic computational model we use to represent programs is that of a *fair transition system.* A fair transition system $\mathcal{S}$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of *state variables.* Some of these variables represent *data* variables, which are explicitly manipulated by the program text. Other variables are *control* variables, which represent, for example, the location of control in each of the processes in a concurrent program. We assume each variable to be associated with a domain over which it ranges.

- Σ : A set of *states.* Each state $s \in \Sigma$ is an interpretation of V, assigning to each variable $u \in V$ a value over its domain, which we denote by $s[u]$. We denote by Σ^+ the set of all nonempty finite sequences of states.

- $\mathcal{T}$: A finite set of *transitions.* Each transition $\tau \in \mathcal{T}$ is associated with an assertion $\rho_\tau(V, V')$, called the *transition relation*, which may refer to both unprimed and primed versions of the state variables. The purpose of the transition relation ρ_τ is to express a relation between a state s and its successor s'. We use the unprimed version to refer to values in s, and the primed version to refer to values in s'. For example, the assertion $x' = x + 1$ states that the value of x in s' is greater by 1 than its value in s.

- Θ : The *initial condition.* This is an assertion characterizing all the initial states, i.e., states at which the computation of the program can start. A state is defined to be *initial* if it satisfies Θ.

- $\mathcal{J} \subseteq \mathcal{T}$: A set of *just* transitions (also called *weakly fair* transitions). Intuitively, 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). Intuitively, 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.

We define the state s' to be a *τ-successor* of the 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 interprets x' as $s'[x]$. Following this definition, we can view the transition τ as a function $\tau : \Sigma \mapsto 2^{\Sigma}$, defined by:

$$\tau(s) = \{s' \mid s' \text{ is a } \tau\text{-successor of } s\}.$$

We say that the transition τ is *enabled* on the state s if $\tau(s) \neq \phi$. Otherwise, we say that τ is *disabled* on s.

We 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 $\tau_{_I}$ (also called the *stuttering* transition), whose transition relation is $\rho_{_I} : (V = V')$. Thus, s' is a $\tau_{_I}$-successor of s iff $s' = s$.

When considering the computations of a transition system, we need not restrict ourselves to states over the state variables V. Let $\mathcal{V}$ be any *vocabulary* (a finite set of variables) that contains V. A $\mathcal{V}$-state is an interpretation of the variables in $\mathcal{V}$. Clearly, we can evaluate Θ on any $\mathcal{V}$-state, and can evaluate each ρ_τ on any pair of $\mathcal{V}$-states. Consequently, the notions of a $\mathcal{V}$-state being initial (satisfying Θ) and one $\mathcal{V}$-state being the τ-successor of another $\mathcal{V}$-state are well defined.

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 $\sigma : s_0, s_1, s_2, \ldots$, 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 $j = 0, 1, \ldots$, 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 point 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.

For a system $\mathcal{S}$, we denote by $\mathcal{C}omp(\mathcal{S})$ the set of all computations of $\mathcal{S}$.

2.1 A Concrete 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 a list of variables $y_1, \ldots, y_k$ and a list of expressions of corresponding types $e_1, \ldots, e_k$,

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

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,

  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 statements S_1 and S_2 and a boolean expression c,

  if c **then** S_1 **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 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. The first step in the execution of this statement selects a statement S_i, $i = 1, \ldots, k$, that is currently enabled and performs the first step in the execution of S_i. Subsequent steps proceed to execute the rest of the selected substatement. 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 [Dij75], [Dij76]), of the form

  if $c_1 \rightarrow S_1 \;\Box\; c_2 \rightarrow S_2 \;\Box\; \ldots \;\Box\; c_k \rightarrow S_k$ **fi**

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

  [**when** c_1 **do** S_1] **or** [**when** c_2 **do** S_2] **or** $\cdots$ **or** [**when** c_k **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 evaluates to F, execution of the statement terminates. Otherwise, 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. Execution of this statement adds 1 to y. It is equivalent to the assignment $y := y + 1$.

Programs

A program P has the form

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

where $P_1 :: S_1, \ldots, P_m :: S_m$ are *named processes*. Each S_i is a statement called the *body* of the process, and P_i is an 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, ..., variable: type **where** φ.

The mode of each declaration statement may be **in**, **local**, or **out**, identifying variables that are inputs, 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 φ imposes constraints on the initial values of the variables declared in this statement. Assertion φ 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.

Labels and Locations in Programs

Each statement in a program may be labeled. Statements that are bodies of processes are also *post-labeled* as in

$$\ell_i :\ S_i\ : \widehat{\ell}_i.$$

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 labeled. In principle, we consider each program to be fully labeled, and denote the label of statement S by $label(S)$. In actual presentations of programs, we often omit many of the labels.

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 ℓ_3 whose body is assignment ℓ_4.

The second important role of the labels is to serve as possible *sites* of control. Thus, a state in the execution of the program of Fig. 1, in which the next step to be taken will perform statement ℓ_7, can be described by saying that control resides at ℓ_7 in this state.

The problem with interpreting each label as a site of control is that there are too many labels, and we do not necessarily wish to distinguish between them. For example, it is clear that when control is at ℓ_2 it is also at ℓ_3 and at ℓ_5, because either 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:

$$
\begin{array}{l}
\textbf{in} \quad\; x_1,\ x_2 \colon \textbf{integer where } x_1 > 0,\ x_2 > 0 \\
\textbf{local } y_1,\ y_2 : \textbf{integer where } y_1 = x_1,\ y_2 = x_2 \\
\textbf{out} \quad g \quad\;\; : \textbf{integer}
\end{array}
$$

$$
\ell_0\colon \left[\begin{array}{l}
\ell_1\colon \left[\begin{array}{l}
\textbf{while } y_1 \neq y_2 \textbf{ do} \\
\ell_2\colon \left[\begin{array}{l}
\ell_3\colon \textbf{ when } y_1 > y_2 \textbf{ do } [\ell_4\colon\ y_1 := y_1 - y_2] \\
\quad \textbf{or} \\
\ell_5\colon \textbf{ when } y_2 > y_1 \textbf{ do } [\ell_6\colon\ y_2 := y_2 - y_1]
\end{array}\right]
\end{array}\right] \\
\ell_7\colon\ [g := y_1]
\end{array}\right] : \widehat{\ell}_0
$$

Fig. 1. A fully labeled GCD program.

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

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

2.2 A Programming Language: Semantics

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

Consider a program P given by

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

Let L_P denote the set of locations of program P. We will show how to define a transition system $\mathcal{S}_P$ corresponding to P.

State Variables and States

The *state 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 include the input, output, and local variables, and range over their respectively declared data domains. The control variable π ranges over *sets of locations*. The value of π 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

$$
\left[\begin{array}{c}
\textbf{out } x\colon \textbf{ integer where } x = 0 \\
[\ell_1\colon\ x := x + 1 \ : \widehat{\ell}_1] \quad \| \quad [\ell_2\colon\ x := x + 2 \ : \widehat{\ell}_2]
\end{array}\right].
$$

The state variables for this program are $V = \{\pi, x\}$. While there are many possible states, there are only four states that occur in a computation of this program. These are:

$\langle \pi : \{[\ell_1], [\ell_2]\},\ x : 0\rangle \quad \langle \pi : \{[\widehat{\ell}_1], [\ell_2]\},\ x : 1\rangle \quad \langle \pi : \{[\ell_1], [\widehat{\ell}_2]\},\ x : 2\rangle$
$\langle \pi : \{[\widehat{\ell}_1], [\widehat{\ell}_2]\},\ x : 3\rangle$

Transitions

For each statement S in a program, we define a set of locations, denoted by *postloc*(S), which represents the location of control after termination of S. Since every statement has precisely one successor location, *postloc*(S) will always be a singleton set, i.e., a set consisting of a single member. This set of locations is defined inductively in a top-down manner as follows:

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

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

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

ρ_I: $V' = V$.

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 τ_ℓ, with the transition relation

$$\rho_\ell: \quad \left[\begin{array}{c} [\ell] \in \pi \ \wedge \ \pi' = \pi - \{[\ell]\} \cup \mathit{postloc}(\ell) \\ \wedge \\ (y_1' = e_1) \wedge \cdots \wedge (y_k' = e_k) \ \wedge \displaystyle\bigwedge_{u \in Y - \{y_1, \ldots, y_k\}} (u' = u) \end{array}\right]$$

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

- *Await*: With the statement ℓ: **await** c, we associate a transition τ_ℓ, with the transition relation

ρ_ℓ: $[\ell] \in \pi \ \wedge \ c \ \wedge \ \pi' = \pi - \{[\ell]\} \cup \mathit{postloc}(\ell) \ \wedge \ Y' = Y$

The transition τ_ℓ is enabled only when control is at ℓ and the condition c holds. When taken, it moves from ℓ to the location *postloc*(ℓ). The conjunct $Y' = Y$ stands for the conjunction $\bigwedge_{u \in Y} (u' = u)$.

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

ρ_ℓ^{T}: $[\ell] \in \pi \ \wedge \ \ c \ \wedge \ \pi' = \pi - \{[\ell]\} \cup \{[\ell_1]\} \ \wedge \ Y' = Y$
ρ_ℓ^{F}: $[\ell] \in \pi \ \wedge \ \neg c \ \wedge \ \pi' = \pi - \{[\ell]\} \cup \{[\ell_2]\} \ \wedge \ Y' = Y$.

Relation ρ_ℓ^{T} corresponds to the case that c evaluates to true and execution proceeds to ℓ_1, while ρ_ℓ^{F} corresponds to the case that c is false and execution proceeds to ℓ_2.

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

$$\rho_\ell^{\mathrm{T}}: \quad [\ell] \in \pi \;\wedge\; \quad c \;\wedge\; \pi' = \pi - \{[\ell]\} \cup \{[\tilde{\ell}]\} \;\wedge\; Y' = Y$$
$$\rho_\ell^{\mathrm{F}}: \quad [\ell] \in \pi \;\wedge\; \neg c \;\wedge\; \pi' = \pi - \{[\ell]\} \cup \mathit{postloc}(\ell) \;\wedge\; Y' = Y.$$

Note that, according to ρ_ℓ^{T}, when c evaluates to $\mathbf{T}$ control moves from ℓ to $\tilde{\ell}$. According to ρ_ℓ^{F}, when c evaluates to $\mathbf{F}$ control moves from ℓ to $\mathit{postloc}(\ell)$.

- *Request*: With the statement ℓ: **request**(y), we associate a transition τ_ℓ, with the transition relation

$$\rho_\ell: \quad \left[\begin{array}{c} [\ell] \in \pi \;\wedge\; y > 0 \;\wedge\; \pi' = \pi - \{[\ell]\} \cup \mathit{postloc}(\ell) \\ \wedge \\ y' = y - 1 \;\wedge\; \displaystyle\bigwedge_{u \in Y - \{y\}} (u' = u) \end{array}\right]$$

- *Release*: With the statement ℓ: **release**(y), we associate a transition τ_ℓ, with the transition relation

$$\rho_\ell: \quad [\ell] \in \pi \;\wedge\; \pi' = \pi - \{[\ell]\} \cup \mathit{postloc}(\ell) \;\wedge\; y' = y + 1 \;\wedge\; \bigwedge_{u \in Y - \{y\}} (u' = u).$$

The transitions associated with the semaphore statements *request* and *release* are called *semaphore transitions*.

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 pointing to the initial locations of the processes, and the data variables satisfying the data precondition.

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 $\mathcal{T} - \{\tau_I\}$, the set all transitions, excluding the idling transition τ_I.
- *Compassion*: As the *compassion* set, we take all the semaphore transitions.

This concludes the definition of the transition system $\mathcal{S}_P$.

2.3 Extension to Message-Passing

Several new constructs may be introduced to provide the capability of communication by message passing.

- *Channel Declaration*

  Channels are declared at the head of the program by a *channel declaration*:

mode $\alpha_1, \alpha_2, \ldots, \alpha_n$: **channel of** type.

This declaration identifies $\alpha_1, \ldots, \alpha_n$ as channels through which messages of the specified type can be sent and received.

- *Send Statement*

  The *send* statement has the form

  $\alpha \;\Leftarrow\; e,$

  where α is a channel name and e is an expression whose value is to be sent via channel α. The type of the channel and of the expression must match.

- *Receive Statement*

  The *receive* statement has the form

  $\alpha \;\Rightarrow\; y,$

  where α is a channel name and y is a variable. The type of the channel and of the variable must match. Execution of this statement reads and removes the first message (value) currently held in α and stores it in y.

The send and receive statements appear in different syntactic forms in various languages. Some languages present statements

$\alpha \;\Leftarrow\; e$ and $\alpha \;\Rightarrow\; y$

as

send e **to** α and **receive** y **from** α.

A prevalent notation, introduced in the language CSP, represents these statements as

$\alpha!e$ and $\alpha?y$.

We refer to *send* and *receive* as *communication statements*.

Buffering Capability

Channels may be endowed with a buffering capability. This means that they are able to hold some messages that have been sent but not yet received. Messages are kept in the order sent. We may therefore view buffering channels as queues (or lists) of pending messages and the send and receive statements as append and remove operations on the queues.

The declaration of a channel specifies its buffering capacity. We distinguish between *buffering* and *non-buffering* channels. The transition relations associated with the send and receive statements depend on the buffering capability of the channel and are separately discussed for each of the two cases.

Buffering Channels

Channels with buffering capability are declared by

α: **channel** [1..] **of** type.

For each such declaration, we add to the set of state variables V a new variable α, which is considered to be a data variable belonging to Y. The domain of this variable consists of lists of elements of the declared type. Thus, if channel α is declared as

α: **channel** [1..] **of integer**,

the new state variable α ranges over lists of integers.

List α represents the messages that have been written to channel α by some send statements and have not yet been read by a receive statement.

- With a send statement

  $\ell\colon\ \alpha \Leftarrow e,$

  we associate a transition τ_ℓ, whose transition relation is given by

  $$\rho_\ell\colon\quad [\ell] \in \pi \ \wedge\ \pi' = \pi - \{[\ell]\} \cup \mathit{postloc}(\ell) \ \wedge\ \alpha' = \alpha \bullet e \ \wedge \bigwedge_{u \in Y - \{\alpha\}} (u' = u).$$

  The data part of the assertion ρ_ℓ describes the new value of α as being obtained by appending the value of e to the end of the old value of α.

- With a receive statement

  $\ell\colon\ \alpha \Rightarrow y,$

  we associate a transition τ_ℓ, whose transition relation is given by

  $$\rho_\ell \ :\ \left[\begin{array}{c} [\ell] \in \pi \ \wedge\ |\alpha| > 0 \ \wedge\ \pi' = \pi - \{[\ell]\} \cup \mathit{postloc}(\ell) \\ \wedge \\ y' = hd(\alpha) \ \wedge\ \alpha' = tl(\alpha) \ \wedge \displaystyle\bigwedge_{u \in Y - \{\alpha, y\}} (u' = u) \end{array}\right]$$

  The data part of ρ_ℓ states that transition τ_ℓ is enabled only if channel α is currently nonempty and, when executed, its effect is to deposit the first element (head) of α in y and to remove this element (retaining the tail) from α.

These definitions imply that the send statement $\alpha \Leftarrow e$ is always enabled when control is at the statement. In contrast, for the receive statement $\alpha \Rightarrow y$ to be enabled it is also required that channel α be nonempty.

Variable α, added to the state variables, is similar to the control variable π in being an auxiliary variable. This means that it is required for the representation of a message-passing program as a transition system, but it cannot be explicitly referenced as a variable by statements in the program.

To the initial condition Θ, we add the clause $(\alpha = \Lambda)$, for each declared channel α.

We refer to communication along buffering channels as *asynchronous communication*.

No Buffering

Channels with no buffering capacity are declared simply by

α: **channel of** type.

Such a channel cannot hold messages even for a short duration. Therefore, communication along such a channel allows no delay between the sending and receiving of a message. Consequently, send and receive statements can only be performed simultaneously. This implies that any process reaching either a send or a receive statement delays its execution until another process is ready to perform the matching statement. Two statements belonging to different processes are considered to be *matching* if they form an $\alpha \Leftarrow e$, $\alpha \Rightarrow y$ pair for some e and y and for the same channel α. When two matching statements are jointly ready to execute, their execution is atomic and simultaneous and the effect is equivalent to the assignment

$y \ := \ e.$

To express this in the semantic framework we have been using, we associate with each pair of matching send and receive statements

$\ell_1\colon\ \alpha \Leftarrow e \qquad \ell_2\colon\ \alpha \Rightarrow y,$

a corresponding joint transition $\tau_{\langle \ell_1, \ell_2 \rangle}$, whose transition relation is given by

$$\rho_{\langle \ell_1, \ell_2 \rangle} \ : \ \left[\begin{array}{c} \{[\ell_1], [\ell_2]\} \subseteq \pi \ \wedge \ \pi' = \pi - \{[\ell_1], [\ell_2]\} \cup \mathit{postloc}(\ell_1) \cup \mathit{postloc}(\ell_2) \\ \wedge \\ y' = e \ \wedge \ \bigwedge\limits_{u \in Y - \{y\}} (u' = u) \end{array} \right]$$

Thus, transition $\tau_{\langle \ell_1, \ell_2 \rangle}$ is enabled only if control is both at ℓ_1 and at ℓ_2 at the same time. This is tested by the inclusion $\{[\ell_1], [\ell_2]\} \subseteq \pi$. When executed, the transition causes joint progress in the two processes containing the communication statements, simultaneously replacing $[\ell_1]$ by the post-location of ℓ_1, and $[\ell_2]$ by the post-location of ℓ_2. It also performs the communication itself, assigning to y the value of the expression e.

Note that state variable α is not needed in π for the case of no buffering and therefore no clause is added to the initial condition for non-buffering channels.

We refer to communication along non-buffering channels as *synchronous communication.*

Additions to the Compassion Set

To the compassion set we add all the transitions τ_ℓ associated with asynchronous send and receive statements and all joint transitions $\tau_{\langle \ell_1, \ell_2 \rangle}$ associated with pairs of matching synchronous communication statements.

2.4 Temporal logic

We assume an underlying assertion language $\mathcal{L}$ which contains the predicate calculus and interpreted symbols for expressing the standard operations and relations over some concrete domains. For the sake of completeness, we require that one of the domains is that of the integers, or another domain with similar expressive power.

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 to which we apply the boolean operators $\neg$ and $\vee$ (the other boolean operators can be defined from these), and the following basic temporal operators:

$\bigcirc$ – Next $\quad \odot$ – Previous

$\mathcal{U}$ – Until $\quad \mathcal{S}$ – Since

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 variables mentioned in p.

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

- For a state formula p,
  $(\sigma, j) \models p \iff s_j \models p$
  That is, we evaluate p locally, using the interpretation given by s_j.
- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \iff (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ for some $k \geq j, (\sigma, k) \models q$, and for every i such that $j \leq i < k, (\sigma, i) \models p$
- $(\sigma, j) \models \odot p \iff j > 0$ and $(\sigma, j-1) \models p$
- $(\sigma, j) \models p \mathcal{S} q \iff$ for some $k \leq j, (\sigma, k) \models q$, and for every i such that $j \geq i > k, (\sigma, i) \models p$

Additional temporal operators can be defined as follows:

$\diamond p = \mathrm{T} \mathcal{U} p$	– Eventually
$\square p = \neg \diamond \neg p$	– Henceforth
$p \mathcal{W} q = \square p \vee (p \mathcal{U} q)$	– Waiting-for, Unless, Weak Until
$\diamondminus p = \mathrm{T} \mathcal{S} p$	– Sometimes in the past
$\boxminus p = \neg \diamondminus \neg p$	– Always in the past
$p \mathcal{B} q = \boxminus p \vee (p \mathcal{S} q)$	– Back to, Weak Since

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

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

We refer to $\bigcirc$, $\mathcal{U}$, $\diamond$, $\square$, and $\mathcal{W}$ as *future operators* and to $\odot$, $\mathcal{S}$, $\diamondminus$, $\boxminus$, and $\mathcal{B}$ as *past operators*.

A formula that contains no future operators is called a *past* formula. A formula that contains no past operators is called a *future* formula. Note that a state formula is both a past and a future formula. We refer to a past formula [future formula] that is not also a state formula as a *strict-past* [*strict-future*, respectively] formula.

We refer to the set of variables that occur in a formula p as the *vocabulary of* p.

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 *assertionally valid* (*$\mathcal{A}$-valid* for short).

For a temporal formula p and a position $j \geq 0$ such that $(\sigma, j) \models p$, we say that j is a *p-position* (in σ). Note that the satisfaction of a past formula at position $j \geq 0$ depends only on the finite prefix $s_0, \ldots, s_j$.

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

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

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., $\mathcal{C}omp(P) = \mathcal{C}omp(\mathcal{S}_P)$. This leads to the notion of $\mathcal{P}$-validity, by which p is *P-valid* if it holds over all the computations of P. Similarly, we obtain the notions of P-satisfiability and P-equivalence.

2.5 Program Constructs Predicates

To use temporal logic for specifying properties of programs presented in the programming language described previously, we introduce several predicates that are specific to the language and to its communication mechanisms.

For a term t we introduce the notation t^- denoting the *previous value* of t. At position j, $j > 0$, the value of t^- is defined to be the value of t at position $j-1$. At position 0, the value of t^- is taken to be the value of t at 0.

Location Predicates

We introduce state predicates that identify the location of control in a state of a computation. For a statement $\ell: S$, predicate

$$at\_\ell: \quad [\ell] \in \pi$$

holds at a state if control is currently in front of the statement S. The definition refers to the control variable π, which ranges over sets of locations, where each location is an equivalence class of labels.

Expressing the Enabledness of a Transition

For a transition τ with transition relation $\rho_\tau(V, V')$, we define the formula

$$enabled(\tau) \quad : \quad \exists V' \; \rho_\tau(V, V').$$

This formula states that there exist possible values for the primed version of the variables V' such that the relation ρ_τ holds between the current values of V and the existing values for V'. It is not difficult to see that the transition τ is enabled at position j of a computation iff s_j, the state at that position, satisfies $enabled(\tau)$.

Transition Predicates

Define the formula

$$first \quad : \quad \neg \circleddash \mathrm{T}.$$

Obviously *first* characterizes the first position of a computation as a position that has no predecessor.

For a transition τ, we define the predicate

$$last\text{-}taken(\tau): \quad \neg first \;\wedge\; \rho_\tau(V^-, V).$$

Clearly $last\text{-}taken(\tau)$ holds at position j of a computation iff $j > 0$ and s_j is a τ-successor of s_{j-1}, i.e., τ was taken at position $j-1$.

Communication Predicates

For a language that allows communication statements, we need the ability to observe that a communication has taken place in the transition leading to the current state. We use two transition predicates for that purpose. One of them observes the output of a message to a channel, and the other observes the input of a message from a channel.

We consider separately the cases of asynchronous and synchronous communication.

Asynchronous Communication

In the case of asynchronous communication, we can detect the events of sending and receiving messages from a channel α by the changes to the corresponding state variable α. Consequently, we define

$$[\alpha \lessdot v]: \quad \alpha = \alpha^- \bullet v.$$

This predicate identifies a *sending event*. It holds at position $j > 0$ if the transition leading from s_{j-1} to s_j sent the value v to channel α.

In a similar manner, we define the *receiving event*

$$[\alpha \gtrdot v]: \quad v \bullet \alpha = \alpha^-.$$

This predicate holds at position j iff $j > 0$ and the current value of α equals the previous value α^- minus its first element, which equals v. Clearly, $[\alpha \gtrdot v]$ holds precisely at the states in which v has just been read from channel α.

Synchronous Communication

In the case of synchronous communication, there are no state variables that represent the list of pending messages for each channel. Therefore, to observe that a synchronous communication has taken place, we should look for the activation of the joint transitions that perform synchronous communication.

Let $\tau_{\langle \ell_1, \ell_2 \rangle}$ be a communication transition associated with the pair of matching statements

$$\ell_1\colon\ \alpha \Leftarrow e \qquad \text{and} \qquad \ell_2\colon\ \alpha \Rightarrow y\ .$$

We define

$$comm(\ell_1, \ell_2, v): \quad \textit{last-taken}(\tau_{\langle \ell_1, \ell_2 \rangle}) \ \wedge\ v = e^-.$$

This formula expresses the fact that the joint transition $\tau_{\langle \ell_1, \ell_2 \rangle}$ has just been taken and that the value communicated has been $e^- = v$.

We may now take a disjunction over all matching $\langle \ell_1, \ell_2 \rangle$ pairs that refer to channel α and define

$$[\alpha \lessdot v]: \bigvee_{\langle \ell_1, \ell_2 \rangle} comm(\ell_1, \ell_2, v).$$

Thus, the sending event $[\alpha \lessdot v]$ is defined to have taken place iff the last transition taken is a synchronous communication of value v over channel α.

Note that synchronous communication consists of a receiving event together with a sending event. Consequently, in the specification of such systems we will only use the output predicate $[\alpha \lessdot v]$.

For both synchronous and asynchronous communication, we sometimes need to specify that an input or output on channel α has taken place, without specifying the value communicated. We define the predicates

$$[\alpha \gtrdot\ ] \qquad \text{and} \qquad [\alpha \lessdot\ ]$$

to hold whenever $[\alpha \gtrdot v]$ and $[\alpha \lessdot v]$ hold for some v, respectively.

2.6 Stuttering and Projections

An important concept that leads to greater abstraction in reasoning about programs is that of *stuttering* [Lam83].

We say that the infinite sequence of states $\tilde{\sigma}$ *stretches* the state sequence

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

if there exists an infinite sequence of positive integers $n_0, n_1, n_2, \ldots$, such that $\widetilde{\sigma}$ can be presented as

$$\widetilde{\sigma}: \quad \underbrace{s_0, \ldots, s_0}_{n_0}, \underbrace{s_1, \ldots, s_1}_{n_1}, \underbrace{s_2, \ldots, s_2}_{n_2}, \ldots$$

That is, $\widetilde{\sigma}$ can be constructed by listing $n_0 > 0$ copies of s_0, followed by $n_1 > 0$ copies of s_1, etc.

Two sequences $\widetilde{\sigma}$ and $\widehat{\sigma}$ are called *stuttering equivalent* if there exists a sequence σ such that both $\widetilde{\sigma}$ and $\widehat{\sigma}$ stretch σ.

For example, the sequences

$$\widetilde{\sigma}: \quad s_0, s_0, s_1, s_2, s_2, s_3, \ldots$$

and

$$\widehat{\sigma}: \quad s_0, s_1, s_1, s_2, s_3, s_3, \ldots$$

are stuttering equivalent because they both stretch

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

A formula φ is said to be *stuttering robust* if for every two stuttering equivalent sequences of states, $\widetilde{\sigma}$ and $\widehat{\sigma}$,

$$\widetilde{\sigma} \models \varphi \qquad \textit{iff} \qquad \widehat{\sigma} \models \varphi.$$

It seems that most of the interesting properties of reactive programs are expressible by stuttering robust formulas. Consequently, from now on, we will consider only stuttering robust formulas.

A sufficient condition for stuttering robustness is that the formula does not contain the immediate operators $\bigcirc$ or $\odot$. However, we will not restrict ourselves to such formulas and will also allow formulas that do contain immediate operators, provided they are stuttering robust.

Projections

Let V be a vocabulary and $\widetilde{V}$ a *subvocabulary* (i.e., a subset) of V. For a V-state s, we denote by $s \Downarrow_{\widetilde{V}}$ the $\widetilde{V}$-state obtained by omitting from s the interpretations of all variables in $V - \widetilde{V}$. We refer to $s \Downarrow_{\widetilde{V}}$ as the *projection* of s on the subvocabulary $\widetilde{V}$.

Projections can also be applied to infinite state sequences. Let

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

be an infinite sequence of V-states. The *projected sequence* $\sigma \Downarrow_{\widetilde{V}}$ is given by

$$\sigma \Downarrow_{\widetilde{V}}: \quad s_0 \Downarrow_{\widetilde{V}}, s_1 \Downarrow_{\widetilde{V}}, s_2 \Downarrow_{\widetilde{V}}, \ldots .$$

Let σ be a sequence of V-states and p be a formula whose vocabulary is contained in $\widetilde{V}$. Then

$$\sigma \models p \qquad \textit{iff} \qquad \sigma \Downarrow_{\widetilde{V}} \models p.$$

Consider a formula of the form

$$\varphi: \quad p_1 \ \wedge \ \cdots \ \wedge \ p_k.$$

We define the *projected formula* $\varphi \downarrow_{\widetilde{V}}$ to be the formula obtained by removing from φ every conjunct p_i whose vocabulary contains variables not in $\widetilde{V}$. Clearly, if φ holds over σ, then $\varphi \downarrow_{\widetilde{V}}$ holds over $\sigma \downarrow_{\widetilde{V}}$.

3 Specification of Reactive Modules

The global approach to specification and verification by temporal logic is concerned with the specification of entire systems. Clearly, a large system is constructed of several components (modules) that are usually developed by different teams. It is therefore of paramount interest to be able to provide a separate specification for each component that serves as the task definition for the implementors of the component.

In this section we introduce the notions necessary for *modular specification*, i.e., specifying the desired behavior of a module within a system.

3.1 The Module Statement

We extend our programming syntax by introducing the *module* statement, whose form is

M :: [**module**; interface specification; body].

The *interface specification* is a list of declaration statements, each of the form

modes $x_1, \ \ldots, \ x_m$: type **where** φ

where *modes* is a list of one or more modes, each of which may be **local**, **in**, **out**, or **external**. The list $x_1, \ldots, x_m$ consists of names of variables or channels that are declared by this statement. The *type* part specifies the type of the declared variables. The optional assertion φ restricts the initial values of the declared variables. The last component of a module statement, *body*, is a statement.

Statements within the body B of module M may refer only to variables and channels that are declared within the interface specification of M.

Let y be a variable declared by the interface specification. References to y within statements of B and statements parallel to M are restricted as follows:

- A statement in B may have a reading reference to y only if y is declared to be of the modes **local** or **in**.
- A statement in B may have a writing reference to y only if y is declared to be of the modes **local** or **out**.
- A statement in a module parallel to M may have a writing reference to y only if y is declared (in M) to be of the mode **external**.

Similar restrictions apply to channels, where a receive statement counts as a reading reference and a send statement counts as a writing reference. Note, however, that a channel α declared as *out* but not *external* is still modifiable by the environment since reading from an asynchronous channel removes the first message from variable α.

Thus, the purpose of the interface specification is to identify the variables and channels that are used for communication between module M and other modules. It also determines the modes in which these variables may be used, i.e., whether they can be written or only read by the module.

Consider, for example, program PING-PONG presented in Fig. 1. In this program, M_2 serves as the environment of M_1 and M_1 as the environment of M_2.

$$
M_1 :: \left[\begin{array}{l}
\textbf{module} \\
\begin{array}{lll}
\textbf{external in} & y & : \textbf{integer} \\
\textbf{out} & x,\ z & : \textbf{integer where } x = 0,\ z = 0
\end{array} \\
\qquad \ell_0 : \left[\begin{array}{l}
\ell_1\text{: } x := 1 \\
\ell_2\text{: } \textbf{await } (y = 1) \\
\ell_3\text{: } z := 1
\end{array}\right] : \widehat{\ell_0}
\end{array}\right]
$$

$$\|$$

$$
M_2 :: \left[\begin{array}{l}
\textbf{module} \\
\begin{array}{lll}
\textbf{external in} & x & : \textbf{integer} \\
\textbf{out} & y & : \textbf{integer where } y = 0
\end{array} \\
\qquad m_0 : \left[\begin{array}{l}
m_1\text{: } \textbf{await } (x = 1) \\
m_2\text{: } y := 1
\end{array}\right] : \widehat{m_0}
\end{array}\right]
$$

Figure 1: Program PING-PONG.

The interface specification of module M_1 declares x and z as writable integer variables, whose initial value is 0. It also forbids these variables (by not declaring them as external) to be written by the environment. Variable y is declared as an integer variable that may be read by M_1 and modified externally. In a complementary way, the interface specification for M_2 identifies x as an integer variable that is read-only for M_2 and writable by the environment. Variable y is identified as writable by M_2 and read-only for the environment of M_2. Note that the bodies of the two modules respect these restrictions.

The program behaves as follows. Initially, the values of x, y, and z are 0. Module M_1 starts a protocol of communication between the two modules by setting x to 1. This is sensed by M_2, which responds by setting y to 1. Module M_1 senses this change and sets z to 1.

3.2 Modular Specification

Let M_1 and M_2 be two modules. We say that M_1 and M_2 are *interface compatible* (also that M_1 [respectively M_2] is interface compatible with M_2 [respectively M_1]) if the declarations for any variable x that is declared in both M_1 and M_2 are consistent (i.e., satisfiable). By that we mean the following:

- The types specified in both declarations are identical.
- If the two declarations contain where clauses which specify constraints φ_1 and φ_2 on the initial value of the declared variable, then $\varphi_1 \wedge \varphi_2$ is consistent.
- If one of the declarations specifies an out mode, the other specifies an external mode.

If M_1 and M_2 appear as parallel statements in a program, we require that M_1 and M_2 be interface compatible.

Modules M_1 and M_2 are said to be *interface complete* if:

- Each variable declared as external in one of the modules is declared as an out variable in the other module.

To simplify the presentation, we base all our definitions on the case of a program P that has the form

$$P :: \quad [M_1 \| M_2],$$

where M_1 and M_2 are interface compatible and interface complete. Consequently, from now on, when we say that two modules are interface compatible, we also require interface completeness.

The transitions associated with a program that contains a module statement are obtained by treating the module statements as the processes of the program.

For a module M, let Y_M denote the set of all data variables that are declared at the head of the module, and let X_M be the subset of these variables that are declared as external to M. The variables of M, V_M, consist of the data variables Y_M and the control variable π.

To simplify the notation, we write $s_{\Downarrow_M}$, $\sigma_{\Downarrow_M}$, and $\varphi_{\Downarrow_M}$ for the projections $s_{\Downarrow_{V_M}}$, $\sigma_{\Downarrow_{V_M}}$, and $\varphi_{\Downarrow_{V_M}}$, respectively.

A formula φ is called a *formula over* M if all of its free (unquantified) variables are in $V_M = Y_M \cup \{\pi\}$.

We define φ, a temporal formula over M, to be *modularly valid* for module M if

$$[M \| \widetilde{M}] \models \varphi,$$

for every module $\widetilde{M}$ that is interface compatible with M. Thus, modular validity requires that M satisfies φ independently of the form and behavior of its partner in the program, module $\widetilde{M}$, as long as the partner respects the constraints implied by the interface specification of M. That is, $\widetilde{M}$ may modify variables known to M (variables in V_M) only if they are declared by the interface specification of M to be external.

Consider, for example, program PING-PONG presented in Fig. 1. The specification

$$(x = 0)\ \mathcal{W}\ \Box(x = 1)$$

states that initially $x = 0$ and x will remain 0 unless it changes to 1 and remains 1 forever. We claim that this specification is modularly valid for M_1. It certainly is valid over the given program. However, it is also valid over any program obtained by replacing module M_2 by an arbitrary module that respects the interface specification of M_1. This is because the actions of M_1 maintain the property, and since x is not declared as external in the interface list of M_1, no other module is allowed to modify it.

On the other hand, consider the formula

$$(y = 0)\ \mathcal{W}\ \Box(y = 1),$$

which states a similar property for variable y. While this formula is valid over program PING-PONG, it is not modularly valid for module M_1. This is because if we replace module M_2 with the module

$$\widetilde{M}_2 :: \quad [\textbf{module};\ \textbf{out}\ y\text{: }\textbf{integer where}\ y = 0;\ \ y := 2],$$

the resulting program $M_1 \| \widetilde{M}_2$ does not satisfy $(y = 0)\,\mathcal{W}\,\Box(y = 1)$.

The formula $(y = 0)\,\mathcal{W}\,\Box(y = 1)$ is, however, modularly valid for the module M_2. No matter which module $\widetilde{M}_1$ we use to replace M_1, the program $\widetilde{M}_1 \| M_2$ satisfies $(y = 0)\,\mathcal{W}\,\Box(y = 1)$.

3.3 The Task of Implementing a Module

Consider the task assigned to an implementing team that is charged with the construction of a module M. We may describe the task definition as consisting of two specification elements:

- An interface specification. This part identifies the types of the variables and channels that are used for communication between the module and its environment. It also identifies the mode of communication for each of the interface variables and channels, i.e., whether they can be written or only read by the module and its environment.
- A behavioral specification. This part consists of a temporal formula, specifying the expected behavior of the module.

Assume that the implementation team for module M_2 is handed an interface specification $inter_2$ and a behavioral specification φ_2. Their task can be described as

construct a body B_2 (a statement), such that

$$[\widetilde{M}_1 \,\|\, M_2 :: [\textbf{module};\ inter_2;\ B_2]] \ \models\ \varphi_2,$$

for every module $\widetilde{M}_1$, interface compatible with M_2.

We may conceive, for example, of a programming team receiving the following specification for module M_2:

$$\begin{array}{lrl} \text{Interface specification:} & \textbf{external in} & x\text{: } \textbf{integer} \\ & \textbf{out} & y\text{: } \textbf{integer where } y = 0. \\ \text{Behavioral specification:} & \multicolumn{2}{l}{(y = 0)\,\mathcal{W}\,\square(y = 1) \qquad \wedge} \\ & \multicolumn{2}{l}{(y = 1) \ \Rightarrow\ \diamondminus\,(x = 1) \qquad \wedge} \\ & \multicolumn{2}{l}{\square(x = 1) \ \Rightarrow\ \diamond\,(y = 1).} \end{array}$$

The behavioral specification consists of two safety clauses and one response clause. The first safety clause states that y maintains the value 0 either forever or until it changes to 1 and remains 1 forever. The second clause states that y may change to 1 only after observing that x equals 1. The response clause states that, if from a certain point x continuously equals 1, then eventually y will become 1.

Handed such a specification, the programming team may construct the following statement as the body B_2 of a module satisfying the given behavioral specification under the constraints implied by the given interface specification:

$$m_0\text{: } [m_1 :\ \textbf{await}\ (x = 1);\ m_2\text{: } y := 1]\ \ : \widehat{m}_0.$$

The reader may wonder at this point why we use such a strong antecedent, namely, $\square(x = 1)$, in the response formula. Is it not simpler to consider the more natural formulation $(x = 1) \Rightarrow \diamond\,(y = 1)$?

The answer to this question is that, while $(x = 1) \Rightarrow \diamond\,(y = 1)$ is a simpler specification, it is more difficult, in fact impossible, to implement in a non-trivial way. A trivial implementation will be one that sets y to 1 independently of the value of x.

To support the statement that $(x = 1) \Rightarrow \diamond\,(y = 1)$ has only trivial implementations, consider the module M_2 appearing in program PING-PONG of Fig. 1, whose body is the statement suggested earlier. We claim that $(x = 1) \Rightarrow \diamond\,(y = 1)$ is not modularly valid for M_2.

For a proof of this claim, consider the module

$$\widetilde{M}_1 :: \left[\begin{array}{c} \textbf{module out}\ x\text{: } \textbf{integer where}\ x = 0 \\ \ell_0 :\ [\ell_1\text{: } x := 1;\ \ell_2\text{: } x := 0]\ \ : \widehat{\ell}_0 \end{array}\right].$$

We claim that the program $\widetilde{M}_1 \| M_2$ does not satisfy the formula $(x = 1) \Rightarrow \diamond(y = 1)$. To see this, consider the following computation of $\widetilde{M}_1 \| M_2$ (listing values of π, x, y):

$$\langle\{[\ell_0], [m_0]\},\ 0,\ 0\rangle,\ \ \langle\{[\ell_2], [m_0]\},\ 1,\ 0\rangle,\ \ \langle\{[\widehat{\ell}_0], [m_0]\},\ 0,\ 0\rangle,\ \ \langle\{[\widehat{\ell}_0], [m_0]\},\ 0,\ 0\rangle,\ \ \ldots$$

This computation contains one state in which $x = 1$ but no state in which $y = 1$. Clearly, the problem is that M_2 was not fast enough (technically, was not scheduled at the right moment) to detect the instant at which x equals 1.

While not being a complete proof of the statement that $(x = 1) \Rightarrow \diamond(y = 1)$ has only trivial implementations under the given interface specification **external in** x, the preceding argument certainly provides strong evidence that this is the case.

3.4 Obstacles to Modular Implementation

The preceding example points to an important question. Having composed a specification that we believe captures our intuition of the desired behavior of a system or a component of a system (a module), how do we check that the specification is implementable?

Imagine that we are professional specifiers whose job is to compose specifications that are later delivered to a programming team. We had better make sure, to the best of our ability, that all specifications we construct are implementable.

When we consider specifications of entire systems, to which we refer as *global specifications*, the major obstacle to implementability is inconsistency of the specification. For example, it is obvious that the specification

$$\square(x = 0)\ \wedge\ \diamond(x = 1)$$

cannot be implemented. Checking the consistency of a specification φ is a well-defined problem in logic (albeit not an easy one). We only have to confirm that φ is satisfiable.

However, once we consider modular specifications, we encounter additional obstacles to implementability. In general, a modular specification is unimplementable if it requires the module to maintain a property in a way that is inconsistent with its interface specification. Thus, in some sense, we may still attribute unimplementability to inconsistencies in the specification. In the case of global specifications, inconsistencies can only arise between different parts of the behavioral specification, such as the conjuncts $\square(x = 0)$ and $\diamond(x = 1)$. In the case of modular specifications, we may have additional conflicts between the behavioral and interface specifications.

Consider, for example, the combination of the interface specification **external in** x: **integer** with the behavioral specification $\diamond(x = 1)$. The behavioral part requires that the module causes x to become 1. The interface part requires that the module never modifies x. Clearly, the two requirements are in conflict.

As another example, the interface specification **external out** x: **integer** is in conflict with the behavioral specification $\square(x = 0)$. This is because, try as it may to keep x at 0, the module may still lose to the environment, which is also allowed to modify x and may set x to a nonzero value.

3.5 Computations of a Module

A formula φ has been defined to be modularly valid for a module M if φ is valid over the program $M \| \widetilde{M}$ for any module $\widetilde{M}$, interface compatible with M. This definition seems to imply that, in order to check whether φ is modularly valid for M, we somehow should consider all the infinitely many interface compatible modules $\widetilde{M}$.

Here we present a more effective approach to the notion of modular validity. With a module M we associate a transition system $\mathcal{S}_M$ and consider the computations of $\mathcal{S}_M$. We will first consider the case of the shared-variables model.

Computations of a Shared-Variables Module

Let M be a shared-variables module, i.e., there are no communication statements in the body of M, and let L_M denote the set of locations in M. The transition system $\mathcal{S}_M$ consists of the following components:

- *State variables*: These are given by $V_M = \{\pi\} \cup Y_M$. Variable π ranges over sets of locations that may also contain non-L_M locations.
- *States*: All possible interpretations of a vocabulary containing V_M consistent with the types of the variables.
- *Transitions*: These consist of the *idling transition* τ_I, all the transitions associated with statements of M to which we refer as the *internal transitions*, and an *environment transition* τ_E.

The transition relation for the idling transition is given by $\rho_I: \ V_M' = V_M$. It ensures the preservation of all the state variables of M.

For each transition τ associated with a statement of M, the transition relation ρ_τ is defined as though M was the only process in the program. In particular, it only refers to the variables in V_M.

Transition τ_E is intended to represent all the possible interference of the environment with the operation of the module. Recall that $X_M \subseteq Y_M$ is the set of variables declared as external in the interface specification of M. The transition relation for τ_E is given by

$$\rho_{\tau_E}: \quad \pi' \cap L_M = \pi \cap L_M \ \wedge \bigwedge_{u \in Y_M - X_M} (u' = u) \ \wedge \ \left(X_M' \neq X_M \rightarrow \pi' \neq \pi\right).$$

Thus, transition τ_E pledges to preserve the values of all the nonexternal data variables and the set of M-locations within π. By omission, it may arbitrarily change values of external data variables and modify the set of non-M locations contained in π. The last conjunct of the formula requires that if there is a change in one of the external variables ($X_M' \neq X_M$ stands for the disjunction $\bigvee_{x \in X_M} (x' \neq x)$), then there is also a change in π. Due to the conjunct $\pi' \cap L_M = \pi \cap L_M$, this change can involve only the non-L_M part of π.

- *Initial condition*: Let φ_M be the conjunction of all the where parts of declarations in M, and ℓ_0 the initial location of the module. Then the initial condition is

$$\Theta: \quad \pi \cap L_M = \{[\ell_0]\} \ \wedge \ \varphi_M.$$

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

Note that the main difference between $\mathcal{S}_M$ and the transition system we would have constructed if M were the entire program is the presence of the environment transition τ_E in $\mathcal{S}_M$.

It is also important to understand the difference between the global transition relation corresponding to a statement in the transition system $\mathcal{S}_P$ for the entire program, and the modular transition relation for the same statement in the modular transition system $\mathcal{S}_M$ of the module to which the statement belongs.

Consider, for example, statement m_2 in program PING-PONG of Fig. 1. The transition relation corresponding to this statement in the global transition system $\mathcal{S}_P$ is given by

$$\rho_{m_2}: \quad [m_2] \in \pi \ \wedge \ \pi' = \pi - \{[m_2]\} \cup \{[\widehat{m}_0]\} \ \wedge \ y' = 1 \ \wedge \ x' = x \ \wedge \ z' = z.$$

In comparison, the modular transition relation for the same statement within the transition system $\mathcal{S}_{M_2}$ is given by

$$\rho_{m_2 \ M_2}: \quad [m_2] \in \pi \ \wedge \ \pi' = \pi - \{[m_2]\} \cup \{[\widehat{m}_0]\} \ \wedge \ y' = 1 \ \wedge \ x' = x.$$

To distinguish between the two, we often denote the modular transition relation by $\rho_{\tau \ M}$, as shown above.

Note that ρ_{m_2} and $\rho_{m_2 \ M_2}$ differ only in the conjunct $z' = z$ which refers to the variable z that is a state variable of the complete program but not of module M_2.

We define the *computations of module M* to be the computations of system $\mathcal{S}_M$.

Example

Consider the module:

$$M_1 :: \left[\begin{array}{c} \textbf{module};\ \textbf{external in out}\ x\text{: }\textbf{integer where}\ x = 0 \\ \ell_0\text{: }\ x := x + 1 : \widehat{\ell}_0 \end{array} \right]$$

Based on the construction of the transition system $\mathcal{S}_{M_1}$, this module has the following computation (listing values of π, and x and displaying the transitions responsible for each step):

$$\langle \{[\ell_0], [m_0]\}, 0\rangle \xrightarrow{\tau_E} \langle \{[\ell_0], [m_1]\}, 5\rangle \xrightarrow{\tau_{\ell_0}} \langle \{[\widehat{\ell}_0], [m_1]\}, 6\rangle \xrightarrow{\tau_E} \langle \{[\widehat{\ell}_0], [m_2]\}, 0\rangle \rightarrow \cdots$$

Note that the interpretation of π in this computation also contains locations that are not in M_1. This is allowed by the definition. In the presented computation the non-M_1 part of π is modified by the environment transition τ_E. ▟

From now on, we will refer to computations of complete programs as *closed computations* and to computations of modules as *open computations*. The rationale for this is that computations of a module are open to interference from the environment, while computations of a complete program are not.

The approach to compositionality presented in [BKP84] introduced a special (control) state variable Π that is intended to identify, at each step of an open computation, whether the step corresponds to an internal transition or to to the environment transition. Here, this information is encoded in the changes to the control variable π. A change in $\pi \cap L_M$ in a step implies that an internal transition is taken. A change in $\pi - L_M$ identifies the environment transition, while a step that involves no change in π identifies the idling transition.

Let σ be an infinite sequence of states that interpret the variable π, and L a set of locations. We say that σ is *L-controlled* if $\pi \subseteq L$ in all states of σ.

The precise relation between the closed computations of a program and the open computations of the modules it contains is stated by the following proposition.

Proposition 1 *Let $P :: [M_1 \| M_2]$ be a program. An infinite state sequence σ is a closed computation of P iff σ is an L_P-controlled open computation of both M_1 and M_2.*

Proof:

The proposition compares computations of the three transition systems: $\mathcal{S}_P$, $\mathcal{S}_{M_1}$, and $\mathcal{S}_{M_2}$. For brevity, we refer to the last two as $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. To distinguish between the components belonging to different systems, we use the subscripts P, 1, and 2, respectively.

For $i = 1, 2$, let φ_i denote the data preconditions of module M_i, L_i denote the set of locations in M_i, Y_i denote the set of data variables declared in M_i, and X_i denote the subset of variables declared in M_i as external. Let ℓ_0^i denote the initial location of M_i.

Let

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

be a closed computation of P. By the definition of closed computations, it is obvious that σ is L_P-controlled. We will show that it is also an open computation of M_1 (i.e., a computation of $\mathcal{S}_1$). The proof that it is also an open computation of M_2 follows identical lines.

We will consider in turn each of the requirements a computation of $\mathcal{S}_1$ must satisfy and show that σ satisfies all of them.

Initiation: State s_0 satisfies Θ_P the initial condition of program P. This condition is given by

$$\Theta_P : \quad \pi = \{[\ell_0^1],\ [\ell_0^2]\}\ \wedge\ \varphi_1\ \wedge\ \varphi_2.$$

Under the assumption that the range of π is subsets of L_P, Θ_P is equivalent to the conjunction $\Theta_1 \wedge \Theta_2$. Consequently, s_0 satisfies Θ_1 and is initial also for $\mathcal{S}_1$.

Consecution: For every $i \geq 0$, there exists a transition $\tau \in \mathcal{T}_P$ such that s_{i+1} is a τ-successor of s_i. By considering the different types of τ, we will show that there exists a corresponding transition $\tau_1 \in \mathcal{T}_1$ such that s_{i+1} is a τ_1-successor of s_i.

- τ corresponds to a statement S of M_1:
  In this case we take τ_1 to be $\tau\ _{M_1}$, i.e., the transition whose transition relation $\rho_{\tau_1} = \rho_{\tau\ M_1}$ is obtained by removing from ρ_τ any conjuncts that refer to non-V_1 variables. Obviously ρ_{τ_1} is implied by ρ_τ. Consequently, since the joint interpretation $\langle s_i, s_{i+1}\rangle$ satisfies ρ_τ, it also satisfies ρ_{τ_1} and therefore s_{i+1} is also a τ_1-successor of s_i.

- τ corresponds to a statement of M_2:
  In this case s_i and s_{i+1} must agree on the value of $\pi \cap L_1$ and the values of all variables in $Y_1 - X_1$, i.e., the data variables of M_1 that are not declared as external. Consequently, the joint interpretation $\langle s_i, s_{i+1}\rangle$ satisfies ρ_{τ_E}, and we can take τ_1 to be τ_E the environment transition.

- τ is the idling transition of P:
  In this case s_i and s_{i+1} must agree on the values of all the state variables of P. In particular, they agree on the values of variables in V_1. Consequently, we take τ_1 to be the idling transition of M_1.

Justice: Let τ_1 be an internal transition of M_1 that is continuously enabled beyond position j in σ. We will show that it is taken at least once beyond j.

Being internal, τ_1 corresponds to a statement S belonging to M_1. Obviously, we can always identify a transition $\tau \in \mathcal{T}_P$ that corresponds to S in the program P and such that ρ_{τ_1} is equivalent to $\rho_{\tau}\ {}_{M_1}$. Recall that the enabling conditions for τ and τ_1 are given, respectively, by

$$\exists V'_P\ \rho_\tau(V_P, V'_P) \qquad \text{and} \qquad \exists V'_1\ \rho_\tau(V_1, V'_1)\ {}_{M_1}.$$

It can be checked that $\rho_\tau(V_P, V'_P)$ and $\rho_\tau(V_1, V'_1)\ {}_{M_1}$ differ only by conjuncts of the form $u' = u$ for a variable $u \in V_P - V_1$, which appears in $\rho_\tau(V_P, V'_P)$ only in this conjunct.

In general, let p be an assertion containing no occurrences of the variables u or u'. Then it is always true that

$$\exists u'\ (p\ \wedge\ u' = u) \quad \text{is equivalent to} \quad p.$$

Using this general observation, we can show that assertions *enabled*(τ) and *enabled*(τ_1) are equivalent, which implies that whenever τ_1 is enabled so is τ.

Consequently, τ is a transition of P which is continuously enabled beyond position j in σ. As σ is a computation of P, it is just with respect to the transition τ which must therefore be taken beyond j, say at position $k \geq j$. This means that s_{k+1} is a τ-successor of s_k. Since ρ_τ implies ρ_{τ_1}, s_{k+1} is also a τ_1-successor of s_k, and τ_1 can be considered as taken at position k.

Compassion: Let τ_1 be a transition belonging to the compassion set $\mathcal{C}_1$ which is enabled infinitely many times in σ. This means that τ_1 corresponds to a semaphore statement in M_1. Let $\tau \in \mathcal{T}_P$ be the P-transition corresponding to the same statement. By the arguments presented above, τ is enabled whenever τ_1 is enabled. It follows that the semaphore transition τ is enabled infinitely many times in σ, and is therefore taken infinitely many times. At each of these positions we can consider τ_1 also to be taken.

Consider now the other direction of the proposition.

Assume that σ is an L_P-controlled open computation of both M_1 and M_2. We will show that it is a closed computation of P.

The first state of σ, s_0, satisfies both Θ_1 and Θ_2. It therefore satisfies their conjunction which is equivalent (under the assumption $\pi \subseteq L_P$) to Θ_P.

Consider a pair of consecutive states, s_j and s_{j+1}, in σ. Each of the modules must justify the step of going from s_j to s_{j+1} by some transition that can either correspond to a statement in the module or be the environment transition. For $i = 1, 2$, let $\tau_i \in \mathcal{T}_i$ be the transition by which M_i justifies the step. We will show that there exists a transition $\tau \in \mathcal{T}_P$ which can justify the step for P.

We observe that if τ_i is a transition internal to M_i (i.e., corresponding to a statement in M_i), then it changes the value of $\pi \cap L_i$ and preserves the value of $\pi - L_i$. Consequently, τ_1 and τ_2 cannot both be internal to their respective modules, and at least one of them must be an environment transition. Let us consider the different cases.

- Case: τ_1 is internal and τ_2 is the environment transition for M_2.

  Let $\tau \in \mathcal{T}_P$ the P-transition corresponding to the same statement that τ_1 does. As previously shown, ρ_{τ_1} is equivalent to $\rho_{\tau}\ {}_{M_1}$. Obviously, ρ_τ and ρ_{τ_1} agree on the relation between the values of the V_1-variables in s_i and in s_{i+1}. In addition, τ_2

ensures that the values of variables belonging to $V_2 - V_1$ are preserved from s_i to s_{i+1}. This guarantees the validity of the conjuncts of the form $u' = u$ for these variables that occur in ρ_τ but are absent from ρ_{τ_1}. Consequently, the full assertion ρ_τ holds on the joint interpretation $\langle s_i, s_{i+1}\rangle$, which establishes that s_{i+1} is a τ-successor of s_i.

- Case: τ_2 is internal and τ_1 is the environment transition for M_1.

  This is handled in a way symmetric to the the previous case.

- Case: both τ_1 and τ_2 are the environment transitions of their respective modules.

  In this case τ_1 ensures the preservation of $\pi \cap L_1$ and of the values of all variables in $Y_1 - X_1$, and τ_2 ensures the preservation of $\pi \cap L_2$ and of the values of all variables in $Y_2 - X_2$. Since, by assumption $\pi \subseteq L_P$, this implies that the complete value of π is preserved. Consider a data variable $y \in Y_i$, for some $i = 1, 2$. If $y \in Y_i - X_i$, then the conjunct $\bigwedge\limits_{u \in Y_M - X_M} (u' = u)$ in ρ_{τ_i} ensures that y is preserved. If $y \in X_i$, then the last conjunct of ρ_{τ_i} allows y to change only if π does, and we have already established that π is preserved. This shows that s_i and s_{i+1} agree on the values of all variables in $\{\pi\} \cup Y_1 \cup Y_2 = V_P$. Consequently, the idling transition of P can justify the step.

- Case: either τ_1 and τ_2 are the environment transitions of their respective modules.

  Without loss of generality, assume that τ_1 is the idling transition of M_1. This transition requires that the full value of π is preserved in the current step. Consequently, τ_2 must be the idling transition of module M_2. Between them they guarantee the preservation of the variables in $V_1 \cup V_2 = V_P$. Consequently, we can take τ to be the idling transition of P.

Justice: Let $\tau \in \mathcal{T}_P$ be a non-idling transition that is continuously enabled beyond position j in σ. We will show that it is taken at least once beyond j.

Obviously, τ corresponds to some statement that belongs to module M_i for some $i = 1, 2$. Let $\tau_i \in \mathcal{T}_i$ be the M_i-transition that corresponds to the same statement. Clearly, ρ_τ implies ρ_{τ_i}, and therefore τ_i is also continuously enabled beyond position j. Since σ is an open computation of M_i, it is just with respect to τ_i, which means that τ_i is taken at some position $k \geq 0$. We will show that τ can also be considered as taken at k.

The main difference between ρ_τ and $\rho_{\tau_i} = \rho_{\tau}{}_{\;M_i}$ is that ρ_τ also ensures that all variables in $V_P - V_i$ are preserved. Let $\bar{i}$ denote the index of the module that is parallel to M_i in P. That is, if $i = 1$ then $\bar{i} = 2$ and vice versa. As σ is also a computation of $M_{\bar{i}}$, there is a transition $\tau_{\bar{i}} \in \mathcal{T}_{\bar{i}}$ which is taken at k. It is not difficult to see that $\tau_{\bar{i}}$ can only be the environment transition of $M_{\bar{i}}$ which guarantees preservation of all variables in $Y_{\bar{i}} - X_{\bar{i}}$. By the requirement of interface completeness, $X_{\bar{i}} \subseteq Y_i$. Since π also belongs to V_i, it follows that $(V_P - V_i) \subseteq (Y_{\bar{i}} - X_{\bar{i}})$, and therefore all variables in $V_P - V_i$ are preserved between s_k and s_{k+1}. We conclude that transition τ can be considered as taken at position k.

The requirement of compassion is treated in a similar way.

This concludes the proof of the proposition. ▟

Example

Consider module M_1 defined as

$$M_1 :: \left[\begin{array}{c} \textbf{module};\ \textbf{external in out}\ x\text{: } \textbf{integer where}\ x = 0 \\ \ell_0\text{: } x := x + 1 : \widehat{\ell}_0 \end{array}\right]$$

and module M_2 defined as

$$M_2 :: \left[\begin{array}{c} \textbf{module};\ \textbf{external in out}\ x\text{: } \textbf{integer} \\ m_0\text{: } \left[\begin{array}{l} m_1 :\ x := x + 5 \\ m_2 :\ x := 0 \end{array}\right] : \widehat{m}_0 \end{array}\right]$$

The following sequence is a closed computation of $P = M_1 \| M_2$:

$$\sigma\text{: } \langle\{\ell_0, m_0\},\ 0\rangle,\ \langle\{\ell_0, m_1\},\ 5\rangle,\ \langle\{\widehat{\ell}_0, m_1\},\ 6\rangle,\ \langle\{\widehat{\ell}_0, m_2\},\ 0\rangle,\ \cdots$$

The same sequence can be presented as an open computation of M_1

$$\langle\{[\ell_0],\ [m_0]\},\ 0\rangle \xrightarrow{\tau_E} \langle\{[\ell_0],\ [m_1]\},\ 5\rangle \xrightarrow{\tau_{\ell_0}} \langle\{[\widehat{\ell}_0],\ [m_1]\},\ 6\rangle \xrightarrow{\tau_E} \langle\{[\widehat{\ell}_0],\ [m_2]\},\ 0\rangle \rightarrow \cdots$$

and also as an open computation of M_2

$$\langle\{[\ell_0],\ [m_0]\},\ 0\rangle \xrightarrow{m_1} \langle\{[\ell_0],\ [m_2]\},\ 5\rangle \xrightarrow{\tau_E} \langle\{[\widehat{\ell}_0],\ [m_2]\},\ 6\rangle \xrightarrow{m_2} \langle\{[\widehat{\ell}_0],\ [\widehat{m_0}]\},\ 0\rangle \longrightarrow \cdots$$

Note how each module interprets a transition performed by its partner as an environment transition. ▪

3.6 From Open to Closed Computations

Proposition 1 states that any sequence σ, which is a closed computation of the program $M \| \widetilde{M}$ for some module $\widetilde{M}$, is also a computation of the module M (i.e., a computation of $\mathcal{S}_M$).

Here we are interested in the converse statement. Ideally, the converse statement should claim that for each σ, an open computation of M, there exists a module $\widetilde{M}$, such that σ is a closed computation of $M \| \widetilde{M}$. There are several reasons why this statement cannot hold in this strong form. Instead, we will show the existence of $\widetilde{M}$ and $\widehat{\sigma}$ a closed computation of $M \| \widetilde{M}$, such that σ and $\widehat{\sigma}$ are closely related.

We will give a more precise definition of the relation expected to hold between σ and $\widehat{\sigma}$. A sequence σ is said to have an *double M-change* at position $i \geq 0$ if, for two distinct variables $u_1, u_2 \in V_M$ (one of which may be π), s_i and s_{i+1} disagree on the values of both u_1 and u_2.

Two sequences, σ and $\widehat{\sigma}$, are defined to be *elementarily M-similar* if they satisfy the following two requirements:

1. For every $i = 0, 1, \ldots$, states s_i and $\widehat{s}_i$ agree on the values of all the variables in Y_M, and on the value of $\pi \cap L_M$.

2. For every $i = 0, 1, \ldots$, sequence σ has a double M-change at position i iff $\widehat{\sigma}$ does.

Note that requirement 2 adds more information to requirement 1 only for the case that one of u_1, u_2 is π. This is because requirement 1 already ensures that s_i and s_{i+1} disagree on precisely the same Y_M variables as $\widehat{s}_i$ and $\widehat{s}_{i+1}$.

The sequences σ and $\widehat{\sigma}$ are defined to be *M-similar* if there exists a list of sequences

$$\sigma = \sigma_1,\ \sigma_2,\ \ldots,\ \sigma_{k-1},\ \sigma_k = \widehat{\sigma}$$

for some $k > 0$, such that for every $i = 1, \ldots, k-1$, sequence σ_{i+1} is either elementarily M-similar to σ_i or is stuttering equivalent to σ_i.

Example

Consider the module:

$$M :: \left[\begin{array}{l} \textbf{module} \\ \textbf{out} \qquad\quad y:\ \textbf{integer where } y = 0 \\ \textbf{external} \ \ x:\ \textbf{integer} \\ \qquad \ell_0: \left[\begin{array}{l} \ell_1:\ \textbf{while T do} \\ \quad \ell_2:\ y := y+1 \end{array}\right] : \widehat{\ell}_0 \end{array}\right]$$

This module has the following open computation (listing values of π, x, and y):

$$\begin{array}{ll} \sigma: & \langle\{[\ell_0], [m_0]\},\ 0,\ 0\rangle \xrightarrow{\ell_1} \langle\{[\ell_2], [m_0]\},\ 0,\ 0\rangle \xrightarrow{\ell_2} \langle\{[\ell_1], [m_0]\},\ 0,\ 1\rangle \xrightarrow{\tau_E} \\ & \langle\{[\ell_1], [m_1]\},\ 1,\ 1\rangle \xrightarrow{\ell_1} \langle\{[\ell_2], [m_1]\},\ 1,\ 1\rangle \xrightarrow{\ell_2} \langle\{[\ell_1], [m_1]\},\ 1,\ 2\rangle \xrightarrow{\tau_E} \\ & \langle\{[\ell_1], [m_2]\},\ 2,\ 2\rangle \xrightarrow{\ell_1} \langle\{[\ell_2], [m_2]\},\ 2,\ 2\rangle \xrightarrow{\ell_2} \langle\{[\ell_1], [m_2]\},\ 2,\ 3\rangle \xrightarrow{\tau_E} \\ & \langle\{[\ell_1], [m_3]\},\ 3,\ 3\rangle \xrightarrow{\ell_1} \cdots \end{array}$$

This computation alternates between taking two internal transitions which increment y followed by an environment transition which increments x and modifies the non-L_M part of π in one step. Examination of the types of programs that can be written in our programming language shows that there is no module $\widetilde{M}$ that can increment x in every step of its execution, while performing infinitely many steps.

The best that can be done is represented by the following module:

$$\widetilde{M} :: \left[\begin{array}{l} \textbf{module} \\ \textbf{out} \qquad\quad x:\ \textbf{integer where } x = 0 \\ \textbf{external} \ \ y:\ \textbf{integer} \\ \qquad k_0: \left[\begin{array}{l} k_1:\ \textbf{while T do} \\ \quad k_2:\ x := x+1 \end{array}\right] : \widehat{k}_0 \end{array}\right]$$

The program $M \| \widetilde{M}$ has the closed computation

$$\begin{array}{ll} \widehat{\sigma}: & \langle\{[\ell_0], [k_0]\},\ 0,\ 0\rangle \xrightarrow{\ell_1} \langle\{[\ell_2], [k_0]\},\ 0,\ 0\rangle \xrightarrow{\ell_2} \\ & \langle\{[\ell_1], [k_0]\},\ 0,\ 1\rangle \xrightarrow{k_1} \langle\{[\ell_1], [k_2]\},\ 0,\ 1\rangle \xrightarrow{k_2} \\ & \langle\{[\ell_1], [k_1]\},\ 1,\ 1\rangle \xrightarrow{\ell_1} \langle\{[\ell_2], [k_1]\},\ 1,\ 1\rangle \xrightarrow{\ell_2} \\ & \langle\{[\ell_1], [k_1]\},\ 1,\ 2\rangle \xrightarrow{k_1} \langle\{[\ell_1], [k_2]\},\ 1,\ 2\rangle \xrightarrow{k_2} \\ & \langle\{[\ell_1], [k_1]\},\ 2,\ 2\rangle \xrightarrow{\ell_1} \langle\{[\ell_2], [k_1]\},\ 2,\ 2\rangle \xrightarrow{\ell_2} \\ & \langle\{[\ell_1], [k_1]\},\ 2,\ 3\rangle \xrightarrow{k_1} \langle\{[\ell_1], [k_2]\},\ 2,\ 3\rangle \xrightarrow{k_2} \\ & \langle\{[\ell_1], [k_1]\},\ 3,\ 3\rangle \xrightarrow{\ell_1} \cdots \end{array}$$

While $\widehat{\sigma}$ is certainly different from σ it is M-similar to it. To show this, we first copy $\widehat{\sigma}$, omitting the transitions and renaming the sequence. This yields

$$\begin{array}{ll}\sigma_1: & \langle\{[\ell_0],[k_0]\},\ 0,\ 0\rangle\ ,\ \langle\{[\ell_2],[k_0]\},\ 0,\ 0\rangle\ ,\ \langle\{[\ell_1],[k_0]\},\ 0,\ 1\rangle\ ,\ \langle\{[\ell_1],[k_2]\},\ 0,\ 1\rangle\ , \\ & \langle\{[\ell_1],[k_1]\},\ 1,\ 1\rangle\ ,\ \langle\{[\ell_2],[k_1]\},\ 1,\ 1\rangle\ ,\ \langle\{[\ell_1],[k_1]\},\ 1,\ 2\rangle\ ,\ \langle\{[\ell_1],[k_2]\},\ 1,\ 2\rangle\ , \\ & \langle\{[\ell_1],[k_1]\},\ 2,\ 2\rangle\ ,\ \langle\{[\ell_2],[k_1]\},\ 2,\ 2\rangle\ ,\ \langle\{[\ell_1],[k_1]\},\ 2,\ 3\rangle\ ,\ \langle\{[\ell_1],[k_2]\},\ 2,\ 3\rangle\ , \\ & \langle\{[\ell_1],[k_1]\},\ 3,\ 3\rangle\ ,\ \cdots\end{array}$$

Next, consider the sequence

$$\begin{array}{ll}\sigma_2: & \langle\{[\ell_0],[m_0]\},\ 0,\ 0\rangle\ ,\ \langle\{[\ell_2],[m_0]\},\ 0,\ 0\rangle\ ,\ \langle\{[\ell_1],[m_0]\},\ 0,\ 1\rangle\ ,\ \langle\{[\ell_1],[m_0]\},\ 0,\ 1\rangle\ , \\ & \langle\{[\ell_1],[m_1]\},\ 1,\ 1\rangle\ ,\ \langle\{[\ell_2],[m_1]\},\ 1,\ 1\rangle\ ,\ \langle\{[\ell_1],[m_1]\},\ 1,\ 2\rangle\ ,\ \langle\{[\ell_1],[m_1]\},\ 1,\ 2\rangle\ , \\ & \langle\{[\ell_1],[m_2]\},\ 2,\ 2\rangle\ ,\ \langle\{[\ell_2],[m_2]\},\ 2,\ 2\rangle\ ,\ \langle\{[\ell_1],[m_2]\},\ 2,\ 3\rangle\ ,\ \langle\{[\ell_1],[m_2]\},\ 2,\ 3\rangle\ , \\ & \langle\{[\ell_1],[m_3]\},\ 3,\ 3\rangle\ ,\ \cdots\ .\end{array}$$

We claim that σ_2 is elementarily M-similar to σ_1. They certainly agree on the values of x, y, and $\pi \cap \{[\ell_0],[\ell_1],[\ell_2],[\widehat{\ell}_0]\}$. Also, wherever σ_1 has a double M-change, so does σ_2. Sequence σ_2 is stuttering equivalent to

$$\begin{array}{ll}\sigma_3: & \langle\{[\ell_0],[m_0]\},\ 0,\ 0\rangle\ ,\ \langle\{[\ell_2],[m_0]\},\ 0,\ 0\rangle\ ,\ \langle\{[\ell_1],[m_0]\},\ 0,\ 1\rangle\ , \\ & \langle\{[\ell_1],[m_1]\},\ 1,\ 1\rangle\ ,\ \langle\{[\ell_2],[m_1]\},\ 1,\ 1\rangle\ ,\ \langle\{[\ell_1],[m_1]\},\ 1,\ 2\rangle\ , \\ & \langle\{[\ell_1],[m_2]\},\ 2,\ 2\rangle\ ,\ \langle\{[\ell_2],[m_2]\},\ 2,\ 2\rangle\ ,\ \langle\{[\ell_1],[m_2]\},\ 2,\ 3\rangle\ , \\ & \langle\{[\ell_1],[m_3]\},\ 3,\ 3\rangle\ ,\ \cdots\ ,\end{array}$$

which is identical to σ. Thus, $M \| \widehat{M}$ has a closed computation ($\widehat{\sigma}$) which is M-similar to σ. ▟

The following proposition provides the converse to Proposition 1.

Proposition 2 *If σ is an open computation of M, then there exists a module $\widetilde{M}$, interface compatible with M, and $\widehat{\sigma}$ a closed computation of $M \| \widetilde{M}$ that is M-similar to σ.*

Proof:

For simplicity, we consider the case that module M has a single external data variable x which ranges over the natural numbers.

We will present a module $\widetilde{M}$ such that, for every σ an open computation of M, there exists $\widehat{\sigma}$ a closed computation of $M \| \widetilde{M}$, such that $\widehat{\sigma}$ is M-similar to σ. Module $\widetilde{M}$ is given by

$$\widetilde{M} :: \left[\begin{array}{l}\textbf{module} \\ \textbf{local}\quad t,\ c\ :\ \textbf{natural} \\ \textbf{out}\quad\ \ x\quad :\ \textbf{natural} \\ k_0 : \left[\begin{array}{l} k_1 : \ \textbf{while}\ \mathrm{T}\ \textbf{do} \\ \quad \left[\begin{array}{l} k_2 :\ (t,c) := (0,0) \\ k_3 :\ \textbf{while}\ t = 0\ \textbf{do} \\ \quad \left[\begin{array}{l} k_4 :\ t := 1 \\ \quad \textbf{or} \\ k_5 :\ c := c+1 \end{array}\right] \\ k_6 :\ x := c \end{array}\right] \end{array}\right] : \widehat{k}_0 \end{array}\right]$$

This module has a while loop with a nondeterministic body that can generate in c any natural number n and assign it to x. This is done by a finite computation segment σ^n of the form (listing values of π (partial value), t, c, and x, and assuming that at the beginning of the segment $\widetilde{M}$ is at k_1 and $x = x_0$):

$$
\begin{array}{ll}
\sigma^n: & \langle\{..., [k_1]\},\ -,\ -,\ x_0, ...\rangle\ ,\ \langle\{..., [k_2]\},\ -,\ -,\ x_0, ...\rangle\ , \\
& n+1 \times \left\{ \begin{array}{lcl}
\langle\{..., [k_3]\},\ 0,\ 0,\ x_0, ...\rangle & , & \langle\{..., [k_5]\},\ 0,\ 0,\ x_0, ...\rangle \\
\langle\{..., [k_3]\},\ 0,\ 1,\ x_0, ...\rangle & , & \langle\{..., [k_5]\},\ 0,\ 1,\ x_0, ...\rangle \\
\quad\quad\cdot & , & \quad\quad\cdot \\
\quad\quad\cdot & , & \quad\quad\cdot \\
\quad\quad\cdot & , & \quad\quad\cdot \\
\langle\{..., [k_3]\},\ 0,\ n,\ x_0, ...\rangle & , & \langle\{..., [k_4]\},\ 0,\ n,\ x_0, ...\rangle
\end{array} \right. \\
& \langle\{..., [k_3]\},\ 1,\ n,\ x_0, ...\rangle\ ,\ \langle\{..., [k_6]\},\ 1,\ n,\ x_0, ...\rangle\ ,\ \langle\{..., [k_1]\},\ 1,\ n,\ n, ...\rangle\ .
\end{array}
$$

Let $\sigma:\ s_0, s_1, \ldots,$ be an open computation of M. We will show how to construct a sequence $\widehat{\sigma}$ that will be a closed computation of $M \| \widetilde{M}$ which is M-similar to σ. The construction begins by putting s_0, the initial state of σ, in $\widehat{\sigma}$. We proceed by scanning the states of σ and adding to $\widehat{\sigma}$ either a single state of σ or a segment that ends in a state of σ. Assume that the last added state was s_i. We consider three cases:

- The transition taken in σ to obtain s_{i+1} from s_i is either an internal transition or the idling transition of M. In this case, add s_{i+1} and repeat the process.
- The transition from s_i to s_{i+1} is an environment transition that assigns to x in s_{i+1} the value $n \neq s_i[x]$. In this case, s_i and s_{i+1} must agree on the M-part of π and must disagree on the non-M part of π. We add to $\widehat{\sigma}$ the computation segment σ^n as defined above.
- The transition from s_i to s_{i+1} is an environment transition that assigns to x in s_{i+1} the value $n = s_i[x]$. As in the first case, we only add add s_{i+1} to $\widehat{\sigma}$.

Applying to $\widehat{\sigma}$ transformations, similar to the ones performed in the preceding example, we can show that $\widehat{\sigma}$ is M-similar to σ.

The more general case, in which M has several external variables of more general types than just natural numbers (but still ranging over countable domains) can be treated by a generalization of the special case considered here. ▪

We are now ready to state the main result concerning modular validity. A formula φ is said to be *M-robust* if for every two M-similar sequences of states, σ and $\widehat{\sigma}$,

$$\sigma \models \varphi \qquad \textit{iff} \qquad \widehat{\sigma} \models \varphi.$$

It seems very plausible that most of the formulas we will ever want to use as specifications for module M will be M-robust. They should certainly be insensitive to stuttering and to the particular values of the non-M part of π. Consequently, restricting our attention to M-robust formulas does not seem a too severe restriction.

The following proposition relates modular validity to validity over open computations.

Proposition 3 *An M-robust formula φ is modularly valid for a module M iff φ holds over all open computations of M.*

Proof:

Assume that φ is an M-robust formula modularly valid for a module M, and let σ be an open computation of M. We will show that φ holds over σ. By Proposition 2, there exists a module $\widetilde{M}$ and $\widehat{\sigma}$, a closed computation of $M \| \widetilde{M}$, such that $\widehat{\sigma}$ is M-similar to σ. As φ is modularly valid for M it holds over $\widehat{\sigma}$. Since φ is M-robust and $\widehat{\sigma}$ is M-similar to σ, φ also holds over σ.

In the other direction, assume that φ holds over all open computations of M. We will show that it also holds over all closed computations of $M \| \widetilde{M}$, for every module $\widetilde{M}$, thus showing φ to be modularly valid for M.

Let $\widetilde{M}$ be an aribtrary module and σ any closed computation of $M \| \widetilde{M}$. By Proposition 1, σ is also an open computation of M, and therefore satisfies φ. ▪

3.7 Computations of Message-Passing Modules

For the case of asynchronously communicating modules, a single environment transition is not sufficient for representing all possible interactions with the environment.

As in the case of shared-variables modules, the transition system $\mathcal{S}_M$ is very similar to the transition system that would have been constructed if M were a complete program. For each channel α declared as external-in, $\mathcal{S}_M$ has a state variable α and a receiving transition τ_ℓ for each receive statement ℓ: $\alpha \Rightarrow y$. For each channel β declared as out, $\mathcal{S}_M$ has a state variable β and a sending transition τ_m for each send statement m: $\beta \Leftarrow e$. The system also has an idling transition and an internal transition for each non-communicating statement in the module.

In addition, $\mathcal{S}_M$ contains the following environment transitions:

- For each external-in channel α, we define an *environment sending* transition $\tau_\alpha^{\mathrm{ES}}$, whose transition relation is

  $$\rho_\alpha^{\mathrm{ES}}:\quad \exists v\ (\alpha' = \alpha \bullet v)\ \wedge\ \pi' \cap L_M = \pi \cap L_M\ \wedge \bigwedge_{u \in Y_M - \{\alpha\}} (u' = u)\ \wedge\ \pi' \neq \pi.$$

  This implies that the environment chooses an arbitrary value v to be added to the end of the message buffer α. In addition, it is required that the M-part of π and the values of all the data variables of M, except α are preserved. The last clause ensures that the non-M part of π changes to signal that this is an environment transition.

- For each out channel β, we define an *environment receiving* transition τ_β^{ER}, whose transition relation is

  $$\rho_\beta^{\mathrm{ER}}:\quad |\beta| > 0\ \wedge\ \beta' = tl(\beta)\ \wedge\ \pi' \cap L_M = \pi \cap L_M\ \wedge \bigwedge_{u \in Y_M - \{\beta\}} (u' = u)\ \wedge\ \pi' \neq \pi.$$

The initial condition is given by

$$\Theta:\quad \pi \cap L_M = \{\ell_0\}\ \wedge\ \varphi_M\ \wedge \bigwedge_{\alpha \in C_M} (\alpha = \Lambda),$$

where φ_M is the conjunction of the *where* parts of the declarations in M, and C_M is the set of channels declared in M

No fairness requirements are associated with the idling transition or the environment transitions τ_α^{ES} and τ_β^{ER}.

Example

Consider the asynchronously communicating module

$$M :: \left[\begin{array}{lll} \textbf{module} & & \\ \textbf{external in} & \alpha & : \textbf{channel of integer} \\ \textbf{local} & x,\ y & : \textbf{where } x = 0,\ y = 0 \\ \multicolumn{3}{c}{\ell_0 : \left[\begin{array}{l} \ell_1\colon\ \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_2\colon\ \alpha \Rightarrow y \\ \ell_3\colon\ x := x + y \end{array}\right] \end{array}\right] : \widehat{\ell}_0} \end{array}\right]$$

The following is an open computation associated with this module, listing for each state the values of π, α, x, and y.

$$\begin{array}{l} \langle\{[\ell_0],[m_0]\},\ \langle\ \rangle,\ 0,\ 0\rangle \xrightarrow{\tau_\alpha^{\text{ES}}} \langle\{[\ell_0],[m_1]\},\ \langle\ 2\ \rangle,\ 0,\ 0\rangle \xrightarrow{\ell_1} \\ \langle\{[\ell_2],[m_1]\},\ \langle 2\rangle,\ 0,\ 0\rangle \xrightarrow{\tau_\alpha^{\text{ES}}} \langle\{[\ell_2],[m_2]\},\ \langle 2,\ 3\rangle,\ 0,\ 0\rangle \xrightarrow{\ell_2} \\ \langle\{[\ell_3],[m_2]\},\ \langle 3\rangle,\ 0,\ 2\rangle \xrightarrow{\ell_3} \langle\{[\ell_1],[m_2]\},\ \langle\ 3\ \rangle,\ 2,\ 2\rangle \longrightarrow \cdots \longrightarrow \\ \langle\{[\ell_1],[m_1]\},\ \langle\ \rangle,\ 5,\ 3\rangle \longrightarrow \cdots . \end{array}$$

We define the open computations of an asynchronously communicating module M to be the computations of the transition system $\mathcal{S}_M$. It can be shown that the environment transitions included in $\mathcal{S}_M$ represent all the possible interactions M may have with its environment.

Consequently, the preceding propositions for shared-variables modules are also valid for asynchronously communicating modules.

Synchronously Communicating Modules

For the case of synchronously communicating modules, we can no longer model the interaction with the environment by separate environment transitions. Transition system $\mathcal{S}_M$ should contain transitions that can perform synchronous communication between the environment and statements of M for each channel declared as external or out.

This can be accomplished by adding

- a transition $\tau_{\langle\ell,\text{E}\rangle}$ for each statement ℓ: $\alpha \Leftarrow e$ addressing a channel α that is declared as out, and
- a transition $\tau_{\langle\text{E},\ell\rangle}$ for each statement ℓ: $\alpha \Rightarrow y$ addressing a channel α that is declared as external-in.

With the transition system $\mathcal{S}_M$ constructed in this way and the resulting notion of computations of a synchronously communicating module, the preceding propositions are also true for synchronously communicating modules.

4 Composing Modular Specifications

Having defined the notions of modules and their specification, we explore the relation between modular validity and validity over the entire program.

4.1 A Compositional rule

For a module M and formula φ, we denote by $M \models_{\mathcal{M}} \varphi$ the fact that φ is modularly valid for M. Consider a program $M :: [M_1 \| M_2]$ and formulas φ_1, φ_2, and φ. We propose rule Comp for the compositional verification that φ is valid over P.

$$
\begin{array}{lll}
\textsc{Comp} & \text{C1.} & M_1 \models_{\mathcal{M}} \varphi_1 \\
 & \text{C2.} & M_2 \models_{\mathcal{M}} \varphi_2 \\
 & \text{C3.} & \big(\Box(\pi \subseteq L_P) \;\wedge\; \varphi_1 \;\wedge\; \varphi_2\big) \quad \rightarrow \quad \varphi \\
\hline
 & & P \models \varphi
\end{array}
$$

Premises C1 and C2 require that φ_1 and φ_2 are modularly valid for M_1 and M_2, respectively. Premise C3 requires that every L_P-controlled sequence that satisfies φ_1 and φ_2 also satisfies φ. The conclusion of the rule is that φ is valid over P.

Proposition 4 *Rule* Comp *is sound and complete.*

Proof:
To prove soundness, assume that premises C1, C2, and C3 are all true. Consider σ an arbitrary closed computation of P. We will show that σ satisfies φ. Obviously σ is L_P-controlled and therefore satisfies $\Box(\pi \subseteq L_P)$. As φ_1 is modularly valid for M_1 it holds over all closed computations of programs of the form $M_1 \| \widetilde{M}$, for any $\widetilde{M}$, in particular for $\widetilde{M} = M_2$. It follows that σ satisfies φ_1. By a similar argument, σ satisfies φ_2. By premise C3, since σ satisfies $\Box(\pi \subseteq L_P) \wedge \varphi_1 \wedge \varphi_2$, it also satisfies φ.

To prove completeness, assume that φ is valid over P. We will identify two formulas, φ_i, for $i = 1, 2$, that satisfies the premises of the rule. For the formula φ_i, we take the *temporal semantics* of module M_i. This is a formula ψ_i that precisely characterizes the open computations of M_i. That is, an arbitrary sequence σ satisfies $\psi - i$ iff it is an open computation of M_i.

Assume that the components of the transition system S_{M_i} are Θ_i, $\mathcal{T}_i$, $\mathcal{J}_i$, and $\mathcal{C}_i$. The temporal semantics formula for M_i is given by

$$\psi_i : \quad \Theta_i \;\wedge\; \bigcirc\Box\Big(\bigvee_{\tau \in \mathcal{T}_i} \mathit{last\text{-}taken}(\tau)\Big) \;\wedge\; \mathit{Just}_i \;\wedge\; \mathit{Compassionate}_i.$$

The first conjunct in this formula requires that the first state in σ is initial, i.e., satisifies Θ_i.

The second conjunct requires that, starting from the second state on, every state in σ is a τ-successor of its predecessor for some $\tau \in \mathcal{T}_i$.

The third conjunct requires that σ be just with respect to the justice set $\mathcal{J}_i$. The formula $Just_i$ stands for

$$Just_i: \quad \bigwedge_{\tau \in \mathcal{J}_i} \Big(\Diamond\, \Box enabled(\tau) \rightarrow \Box \Diamond\, last\text{-}taken(\tau) \Big).$$

This formula requires that every transition $\tau \in \mathcal{J}_i$ that is continuously enabled beyond some position be taken infinitely many times within σ, which is the requirement of justice for τ.

The last conjunct of ψ_i requires that σ be compassionate with respect to the compassion set $\mathcal{C}_i$. The formula $Compassionate_i$ stands for

$$Compassionate_i: \quad \bigwedge_{\tau \in \mathcal{J}_i} \Big(\Box \Diamond\, enabled(\tau) \rightarrow \Box \Diamond\, last\text{-}taken(\tau) \Big).$$

This formula requires that every transition $\tau \in \mathcal{C}_i$ that is enabled infinitely many times be taken infinitely many times within σ, which is the requirement of Compassion for τ.

It is not difficult to check that the formula ψ is indeed a precise characterization of the open computations of M_i. Another fact that can be verified is that ψ_i is M_i-robust. We will proceed to show that taking φ_i to be ψ_i makes the three premises true.

Clearly, all open computations of M_i satisfy ψ_i. By Proposition 3, it follows that ψ_i is modularly valid for M_i. This establishes premises C1 and C2.

Consider now premise C3. Let σ be an L_P-controlled sequence that satisfies both ψ_1 and ψ_2. We will show that it necessarily satisfies φ. Since ψ_i precisely characterizes the open computations of M_i, this means that σ is an L_P-controlled sequence which is an open computation of both M_1 and M_2. According to Proposition 1, σ is a closed computation of P. At the beginning of the completeness proof we explicitly made the assumption that φ is valid over P, namely, that all closed computations for P satisfy φ. It follows that σ satisfies φ. ▟

Rule Comp leads to a useful verification strategy that can be formulated as follows:

Compositional Verification Strategy

To verify that the program $M_1 \| M_2$ satisfies specification φ, find formulas φ_1 and φ_2 that satisfy

$$\Big(\Box(\pi \subseteq L_P) \;\wedge\; \varphi_1 \;\wedge\; \varphi_2\Big) \;\rightarrow\; \varphi,$$

and verify that φ_1 and φ_2 are modularly valid for M_1 and M_2, respectively.

The same rule also suggests an important methodology for the formal development of reactive systems by top-down decomposition.

Development by Decomposition

To develop a reactive program P satisfying φ,

- design compatible interface specifications $inter_1$ and $inter_2$,
- design behavioral specifications φ_1 and φ_2 such that $((\Box(\pi \subseteq L_P) \wedge \varphi_1 \wedge \varphi_2) \rightarrow \varphi$,
- develop bodies B_1 and B_2 such that φ_1 and φ_2 are modularly valid for

  $M_1 ::$ [**module**; $inter_1$; B_1] and

  $M_2 ::$ [**module**; $inter_2$; B_2], respectively.

The required program is given by $M_1 \| M_2$.

In many cases, the interface specifications $inter_1$ and $inter_2$ or parts of them are already given as part of the problem, and we have only to determine the decomposition of φ into φ_1 and φ_2 and to develop the bodies B_1 and B_2.

A key element common to both the development by decomposition and the compositional verification strategies is the decomposition of a global specification φ into two modular specifications φ_1 and φ_2, such that $((\Box(\pi \subseteq L_P) \land \varphi_1 \land \varphi_2) \rightarrow \varphi$. In many cases, the modular specifications satisfy the simpler implication $\varphi_1 \land \varphi_2) \rightarrow \varphi$ which simplifies the search and the verification.

In the rest of this section we will illustrate several cases of such decompositions.

4.2 Layered Decomposition

Consider again program PING-PONG in Fig. 1. As a global specification for this program we take

$$\varphi: \quad \Diamond(z = 1),$$

which states that eventually z will be set to 1. We look for a modular decomposition of this global specification.

In the previous discussion of this program, we presented for module M_2 the modular specification

$$(y = 0)\,\mathcal{W}\,\Box(y = 1) \quad \land \quad (y = 1) \Rightarrow \diamondminus(x = 1) \quad \land \quad \Box(x = 1) \Rightarrow \Diamond(y = 1).$$

In our current analysis we try to be more goal-directed, and include in the modular specifications φ_1 and φ_2 only the minimal part necessary to establish the global property $\Diamond(z = 1)$.

As a general methodology for the construction of modular specifications, we often follow a layered process. As the first layer we put into φ_1 and φ_2 some unconditional properties of the modules M_1 and M_2, respectively. These properties are the effects of actions that are performed independently of the behavior of the environment. At the next layer we put into φ_1 and φ_2 conditional properties of the general form "if the environment guarantees p, I will guarantee q." In most cases, such properties can simply be expressed by the implication $p \rightarrow q$, but there are cases where the dependence of q on p is more complicated. In the layered methodology we put into φ_1, say, a property of the form "q depending on p," only if p has already been included in φ_2 (which is the environment of φ_1) as part of a previous layer.

Let us illustrate this methodology with the case we consider. Clearly, the evolution toward the goal $z = 1$ consists of three events: x becomes 1, y becomes 1, and finally z becomes 1. The first event occurs unconditionally. Each of the other two is conditional on the occurrence of its predecessor. Consequently, the layered methodology recommends the following three steps:

Step 1: Put $\Diamond\Box(x = 1)$ into φ_1. Unconditionally x becomes 1 and stays 1 forever.

Step 2: Put $\Diamond\Box(x = 1) \rightarrow \Diamond\Box(y = 1)$ into φ_2. If x eventually becomes permanently 1, then so does y.

Step 3: Put $\Diamond\Box(y = 1) \rightarrow \Diamond(z = 1)$ into φ_1. If y eventually becomes permanently 1, then z eventually becomes 1.

Thus we wind up with the two modular specifications

$$\varphi_1\colon\quad \Diamond\,\Box(x=1)\ \wedge\ (\Diamond\,\Box(y=1)\ \rightarrow\ \Diamond\,(z=1))$$
$$\varphi_2\colon\quad \Diamond\,\Box(x=1)\ \rightarrow\ \Diamond\,\Box(y=1).$$

It is not difficult to see that $\varphi_1 \wedge \varphi_2$ implies $\Diamond\,(z=1)$.

It is most important to realize that putting a property $p \rightarrow q$ into a modular specification for a module M, when we know that p is guaranteed by the environment, does not contradict our definition that the specification should hold independently of what the environment does. The property $p \rightarrow q$ indeed holds for all environments, even those that do not maintain p. When p is guaranteed we can use this fact to infer q, as we did in the preceding example. It is the validity of q that depends on the behavior of the environment and not the validity of $p \rightarrow q$, which is always guaranteed.

4.3 Variables that Change More than Once

The previous example has been particularly easy due to the fact that each variable changed its value at most once. This enabled us to represent the setting of y that is dependent on the setting of x by the implication

$$\Diamond\,\Box(x=1)\ \rightarrow\ \Diamond\,\Box(y=1).$$

A more complicated situation is presented in program PING-PONG-PING in Fig. 2.

$$M_1 :: \left[\begin{array}{l} \textbf{module} \\ \textbf{external in}\ \ y\ \ \colon \textbf{integer} \\ \textbf{out}\ \ x\ \ \colon \textbf{integer where}\ x=0 \\ \qquad \ell_0 : \left[\begin{array}{l} \ell_1\colon\ x := 1 \\ \ell_2\colon\ \textbf{await}\ (y=1) \\ \ell_3\colon\ x := 0 \end{array}\right] : \widehat{\ell_0} \end{array}\right]$$

$$\|$$

$$M_2 :: \left[\begin{array}{l} \textbf{module} \\ \textbf{external in}\ \ x\ \ \colon \textbf{integer} \\ \textbf{out}\ \ y,\ z\ \ \colon \textbf{integer where}\ y=0,\ z=0 \\ \qquad m_0 : \left[\begin{array}{l} m_1\colon\ \textbf{await}\ (x=1) \\ m_2\colon\ y := 1 \\ m_3\colon\ \textbf{await}\ (x=0) \\ m_4\colon\ z := 1 \end{array}\right] : \widehat{m_0} \end{array}\right]$$

Figure 2: Program PING-PONG-PING.

We can describe the behavior of this program by saying that M_1 signals M_2 twice. The first signal is generated by setting x to 1, and the second signal by resetting x once more to 0. Module M_2 signals M_1 once by setting y to 1. Let us consider the incremental construction of the modular specifications.

The first fact we wish to put in φ_1 is that, unconditionally, eventually x becomes 1 and remains 1 long enough to be detected by M_2. In the previous example we simply stated that x becomes 1 and remains 1 forever. Here we can only promise that x remains 1 until y becomes 1, which allows M_1 to go to ℓ_3 and reset x to 0. Consequently, we put into φ_1 the clause

$$\diamond\left((x=1)\ \mathcal{W}\ (y=1)\right).$$

Next, we express in φ_2 the property that, if x becomes 1 and stays 1 long enough, then eventually y will become 1 and stay 1 forever. This property can be expressed by the formula

$$\diamond\left((x=1)\ \mathcal{W}\ (y=1)\right)\ \rightarrow\ \diamond\,\square(y=1).$$

The formula states that if x is eventually set to 1 and kept at 1 until M_2 responds (by setting y to 1), then indeed M_2 will respond. Furthermore, when y is set to 1 it will remain 1 permanently.

Next, we add to φ_1 the requirement

$$\diamond\,\square(y=1)\ \rightarrow\ \diamond\,\square(x=0),$$

which states that if eventually y becomes permanently 1 then x will become permanently 0.

Finally, we should add to φ_2 a requirement stating that if x becomes permanently 0 then z will become eventually 1. A first candidate for expressing this requirement is the formula

$$\diamond\,\square(x=0)\ \rightarrow\ \diamond\,(z=1).$$

However, this formula is not modularly valid for module M_2. Consider, for example, a variant of M_1, call it M_1', which does nothing, i.e., assigns no values to variables. In the computations of the program $M_1'\|M_2$, x is permanently 0 from the beginning, yet z is never set to 1. This is because if x is always 0, M_2 is not able to go past the await statement m_1.

In view of this, we use the formula

$$[\diamond\,\square(x=0)\ \wedge\ \diamond\,\square(y=1)]\ \rightarrow\ \diamond\,(z=1)$$

to capture the intuition that M_2 commits to set z to 1 in response to x staying at 0 sufficiently long, only if M_2 has gone past m_1 and m_2 and has already set y to 1.

Collecting the various clauses, we obtain the two modular specifications:

$$\begin{aligned}
\varphi_1:&\quad \diamond\left((x=1)\,\mathcal{W}\,(y=1)\right)\ \wedge\ \left(\diamond\,\square(y=1)\rightarrow\diamond\,\square(x=0)\right)\\
\varphi_2:&\quad \left(\diamond\left((x=1)\,\mathcal{W}\,(y=1)\right)\rightarrow\diamond\,\square(y=1)\right)\ \wedge\\
&\qquad\qquad\qquad \left([\diamond\,\square(x=0)\ \wedge\ \diamond\,\square(y=1)]\rightarrow\diamond\,(z=1)\right).
\end{aligned}$$

It can be shown that φ_1 and φ_2 are modularly valid for M_1 and M_2, respectively, and that $\varphi_1\wedge\varphi_2$ implies $\diamond\,(z=1)$.

4.4 Decomposing Safety Specifications

Our previous examples were concerned with the decomposition of global progress specifications of the form $\diamond\,(z=1)$. The decomposition of this global specification showed how this global goal is achieved by M_1 performing some operation (setting x to 1), to which M_2 responded by performing another operation that influences M_1, etc.

Here we consider a global safety property that is an invariant and show how to identify two modular specifications that together ensure the preservation of the invariant.

Consider program KEEPING-UP in Fig. 3. Module M_1 in this program repeatedly increments x, provided x does not exceed $y+1$. Module M_2, similarly, repeatedly increments y, provided y does not exceed $x+1$. For the combined program we state the global safety property

$$
M_1 :: \left[\begin{array}{l}
\textbf{module} \\
\textbf{external in} \quad y \quad : \textbf{integer} \\
\textbf{out} \qquad\qquad x \quad : \textbf{integer where } x = 0 \\
\ell_0\colon \left[\begin{array}{l}
\ell_1 : \ \textbf{while } \mathrm{T} \textbf{ do} \\
\qquad \left[\begin{array}{l} \ell_2\colon \ \textbf{await } (x < y+1) \\ \ell_3\colon \ x := x+1 \end{array}\right]
\end{array}\right] : \widehat{\ell}_0
\end{array}\right]
$$

$$\|$$

$$
M_2 :: \left[\begin{array}{l}
\textbf{module} \\
\textbf{external in} \quad x \quad : \textbf{integer} \\
\textbf{out} \qquad\qquad y \quad : \textbf{integer where } y = 0 \\
m_0\colon \left[\begin{array}{l}
m_1 : \ \textbf{while } \mathrm{T} \textbf{ do} \\
\qquad \left[\begin{array}{l} m_2\colon \ \textbf{await } (y < x+1) \\ m_3\colon \ y := y+1 \end{array}\right]
\end{array}\right] : \widehat{m}_0
\end{array}\right]
$$

Figure 3: Program KEEPING-UP.

$$\Box(|x-y| \leq 1),$$

claiming that the difference between x and y never exceeds 1.

To construct appropriate modular specifications that guarantee this property, we again adopt the strategy of layered construction.

At the first step we put into φ_1 and φ_2, respectively, the following two unconditional formulas

$$\Box(x \geq x^-) \qquad \text{and} \qquad \Box(y \geq y^-).$$

The first formula states that M_1 can only increase the value of x. The second formula states similarly that the value of y can only increase.

Next, we add to φ_1 and φ_2, respectively, the two conditional formulas

$$\Box(y \geq y^-) \ \rightarrow \ \Box(x \leq y+1)$$
$$\Box(x \geq x^-) \ \rightarrow \ \Box(y \leq x+1).$$

The first formula states that if y never decreases, then M_1 can guarantee that x will never exceed $y+1$. This is because M_1 increases x only at location ℓ_3 to which it gets only if in some previous state $x < y+1$. Since from that state until now x retained the same value and y did not decrease, we know that when ℓ_3 is performed x is still smaller than $y+1$.

Thus, the modular specifications we propose are

$$\varphi_1\colon \quad \Box(x \geq x^-) \ \wedge \ \big(\Box(y \geq y^-) \rightarrow \Box(x \leq y+1)\big)$$
$$\varphi_2\colon \quad \Box(y \geq y^-) \ \wedge \ \big(\Box(x \geq x^-) \rightarrow \Box(y \leq x+1)\big).$$

It is easy to see that from the conjunction $\varphi_1 \wedge \varphi_2$ we can infer

$\square(x \leq y+1 \ \wedge \ y \leq x+1)$,
which implies $\square(|x-y| \leq 1)$.

4.5 Asynchronously Communicating Modules

Program ASYNC-PING-PONG in Fig. 4 is a variation of program PING-PONG in Fig. 1 where, instead of communicating via shared variables x, y, and z, the modules communicate via asynchronous channels α, β, and γ.

$$M_1 :: \left[\begin{array}{lll} \textbf{module} & & \\ \textbf{external in} & \beta & : \textbf{channel } [1..] \textbf{ of integer} \\ \textbf{out} & \alpha,\ \gamma & : \textbf{channel } [1..] \textbf{ of integer} \\ \textbf{local} & u & : \textbf{integer} \\ & \ell_0 : \left[\begin{array}{l} \ell_1\text{: } \alpha \Leftarrow 1 \\ \ell_2\text{: } \beta \Rightarrow u \\ \ell_3\text{: } \gamma \Leftarrow 1 \end{array}\right] : \widehat{\ell_0} & \end{array}\right]$$

$$\|$$

$$M_2 :: \left[\begin{array}{lll} \textbf{module} & & \\ \textbf{external in} & \alpha & : \textbf{channel } [1..] \textbf{ of integer} \\ \textbf{out} & \beta & : \textbf{channel } [1..] \textbf{ of integer} \\ \textbf{local} & v & : \textbf{integer} \\ & m_0 : \left[\begin{array}{l} m_1\text{: } \alpha \Rightarrow v \\ m_2\text{: } \beta \Leftarrow 1 \end{array}\right] : \widehat{m_0} & \end{array}\right]$$

Figure 4: Program ASYNC-PING-PONG.

In addition to these channels, the two modules also use two local variables, u and v, respectively. Apart from the change of communication medium, the logical structure of this program is identical to that of the program in Fig. 1.

Communication starts by M_1 sending a message (the value 1 is unimportant) on channel α. Module M_2 waits for the message to arrive. When the message arrives, M_2 sends a message on channel β. When this message is received by M_1, the module proceeds to send a message on channel γ.

A global property of this program is given by

$\diamond [\gamma \lessdot\ ]$,

stating that eventually a message will be sent on channel γ.

Let us consider how we can decompose this global specification into two modular specifications. It is not difficult to see how our modular specifications for the shared-variables case can be translated into corresponding communication-based specifications. These are given by:

φ_1: $\diamond [\alpha \lessdot\ ] \ \wedge \ (\diamond [\beta \lessdot\ ] \rightarrow \diamond [\gamma \lessdot\ ])$

φ_2: $\diamond [\alpha \lessdot\ ] \ \rightarrow \ \diamond [\beta \lessdot\ ]$.

The conjunction $\varphi_1 \wedge \varphi_2$ clearly implies $\diamondsuit [\gamma \lessdot\ ]$.

It is interesting to compare the clause $\diamondsuit [\alpha \lessdot\ ] \rightarrow \diamondsuit [\beta \lessdot\ ]$ to its shared-variables counterpart $\diamondsuit \square (x = 1) \rightarrow \diamondsuit \square (y = 1)$. One may wonder why we need the stronger commitment expressed by the $\diamondsuit \square$ combination in the case of shared variables and manage with the simpler $\diamondsuit$ operator in the case of asynchronous communication.

In the case of shared variables, the $\diamondsuit \square$ is necessary to ensure that the signal generated by M_1 setting x to 1 is not prematurely revoked before it has been sensed by M_2. In the asynchronous message-passing model, signals cannot be revoked, since once a message has been sent by M_1 there is no way for M_1 to change its mind and withdraw or cancel the message.

Beacuse of this and closely related reasons, of the three models considered in this paper (shared-variables, asynchronous, and synchronous communication), asynchronous communication is the most appropriate and easiest model for modular specification and compositional verification.

4.6 Synchronously Communicating Modules

Next, we consider the case of synchronously communicating modules. In Fig. 5, we present program SYNC-PING-PONG, which is similar to program ASYNC-PING-PONG of Fig. 4, except that the channels are synchronous.

$$M_1 :: \left[\begin{array}{lll} \textbf{module} & & \\ \textbf{external in} & \beta & : \textbf{channel of integer} \\ \textbf{out} & \alpha,\ \gamma & : \textbf{channel of integer} \\ \textbf{local} & u & : \textbf{integer} \\ \multicolumn{3}{c}{\ell_0 : \left[\begin{array}{l} \ell_1\!:\ \alpha \Leftarrow 1 \\ \ell_2\!:\ \beta \Rightarrow u \end{array}\right] : \widehat{\ell}_0} \end{array}\right]$$

$$\|$$

$$M_2 :: \left[\begin{array}{lll} \textbf{module} & & \\ \textbf{external in} & \alpha & : \textbf{channel of integer} \\ \textbf{out} & \beta & : \textbf{channel of integer} \\ \textbf{local} & v & : \textbf{integer} \\ \multicolumn{3}{c}{m_0 : \left[\begin{array}{l} m_1\!:\ \alpha \Rightarrow v \\ m_2\!:\ \beta \Leftarrow 1 \end{array}\right] : \widehat{m}_0} \end{array}\right]$$

Figure 5: Program SYNC-PING-PONG.

Another difference between the two programs is that module M_1 terminates after reading the message from channel β and does not send a message on channel γ. This modification is necessary because, in the synchronous communication model, the statement $\gamma \Leftarrow 1$ in module M_1 cannot be executed without a matching receive statement in M_2. Consequently, the global property we consider for this program is

$\Diamond(u=1)$.

When we consider synchronous communication, we face a new problem. Consider, for example, the first statement ℓ_0: $\alpha \Leftarrow 1$ in module M_1. In the asynchronous case, seeing such a first statement, we can immediately write down the formula $\Diamond[\alpha \lessdot\,]$ as a modularly valid property for module M_1. This property states that a message will be sent on channel α independently of the behavior of other modules. This is because in the asynchronous case the operation of sending a message is autonomous and requires no cooperation from the environment.

In the synchronous case, on the other hand, every communication requires cooperation. Module M_1 can send a message on α only if its partner M_2 is ready to participate in a joint transition. To represent this situation we introduce two new state predicates

$$\mathit{ready}([\alpha \lessdot\,]) \qquad \text{and} \qquad \mathit{ready}([\alpha \gtrdot\,]).$$

The state predicate $\mathit{ready}([\alpha \lessdot\,])$ (ready to send on α) is true in a state of a computation of a program if control is in front of a statement of the form

$$\alpha \;\Leftarrow\; e.$$

Similarly, $\mathit{ready}([\alpha \gtrdot\,])$ (ready to receive from α) is true in a state if control is in front of a statement

$$\alpha \;\Rightarrow\; u.$$

Note that, while we have agreed to describe synchronous communication events by the single output predicate $[\alpha \lessdot\,]$ (or $[\alpha \lessdot v]$), we distinguish between the two ready predicates: $\mathit{ready}([\alpha \lessdot\,])$ and $\mathit{ready}([\alpha \gtrdot\,])$. This is because it is important to know whether the module is waiting for input or output.

With these predicates, the property that corresponds to the statement ℓ_0: $\alpha \Leftarrow 1$ in module M_1 can be expressed by

$$\Diamond[\alpha \lessdot\,] \;\vee\; \Box\big(\mathit{ready}([\alpha \lessdot\,]) \;\wedge\; \neg[\alpha \lessdot\,]\big).$$

This formula states that either communication on α will take place or M_1 will remain forever ready to send, vainly waiting for a matching communication partner that will agree to receive.

The conjunction of waiting and not communicating is frequently used. Consequently, we define

$$\mathit{yearn}([\alpha \lessdot\,]): \quad \mathit{ready}([\alpha \lessdot\,]) \;\wedge\; \neg[\alpha \lessdot\,]$$
$$\mathit{yearn}([\alpha \gtrdot\,]): \quad \mathit{ready}([\alpha \gtrdot\,]) \;\wedge\; \neg[\alpha \lessdot\,].$$

Using this abbreviation, we can represent the formula corresponding to the statement ℓ_0: $\alpha \Leftarrow 1$ as

$$\Diamond[\alpha \lessdot\,] \;\vee\; \Box\,\mathit{yearn}([\alpha \lessdot\,]).$$

With this understanding, we may proceed to construct modular specifications for M_1 and M_2. A systematic construction of specifications for synchronously communicating modules can be based on the construction of a separate clause for each location in the program. Consequently we have

$$\varphi_1: \quad \varphi_{1.0} \;\wedge\; \varphi_{1.1} \qquad \text{and} \qquad \varphi_2: \quad \varphi_{2.0} \;\wedge\; \varphi_{2.1},$$

where

$$\varphi_{1.0}: \quad \Diamond[\alpha \lessdot\,] \;\vee\; \Box\,\mathit{yearn}([\alpha \lessdot\,])$$

$$\varphi_{1.1}: \quad \Big[\Diamond[\alpha \lessdot\,] \;\rightarrow\; \big(\Diamond[\beta \lessdot\,] \;\vee\; \Diamond\Box\,\mathit{yearn}[\beta \gtrdot\,]\big)\Big] \;\wedge$$
$$([\beta \lessdot 1] \;\Rightarrow\; (u=1))$$

and

$$\varphi_{2.0}\!: \quad \Diamond[\alpha \lessdot\,] \;\vee\; \Box yearn([\alpha \gtrdot\,])$$
$$\varphi_{2.1}\!: \quad \Big[\Diamond[\alpha \lessdot\,] \;\rightarrow\; \big(\Diamond[\beta \lessdot\,] \;\vee\; \Diamond\Box yearn([\beta \lessdot\,])\big)\Big] \;\wedge$$
$$([\beta \lessdot\,] \;\Rightarrow\; [\beta \lessdot 1]).$$

Clause $\varphi_{1.0}$ claims that either communication over α will occur or some statement (of M_1) will remain ready for output on α forever.

Clause $\varphi_{1.1}$ states two properties. The first is that if there will be an α-communication (after which M_1 must be at ℓ_1), then either there will be a β-communication or M_1 will stay ready for input from β forever. The second property stated by $\varphi_{1.1}$ is a safety property claiming that in every state following a communication of the value 1 over channel β, the value of u equals 1.

Clause $\varphi_{2.0}$ claims that either an α-communication will occur or some statement (of M_2) will remain ready for input from α forever.

Clause $\varphi_{2.1}$ claims that if there will be an α-communication, then either there will be a β-communication of 1, or M_2 will stay ready for output to β forever. It also states that the only value ever communicated on channel β is 1.

Let us consider the conjunction $\varphi_1 \wedge \varphi_2$. In the preceding cases we showed that it immediately implied the global property ($\Diamond(u = 1)$ in our case). Here we can only infer from it the weaker disjunction

$$\Box\Big(yearn([\alpha \lessdot\,]) \;\wedge\; yearn([\alpha \gtrdot\,])\Big) \;\vee$$
$$\Diamond\Box\Big(yearn([\beta \lessdot\,]) \;\wedge\; yearn([\beta \gtrdot\,])\Big) \;\vee\; \Diamond(u = 1).$$

This formula states that either M_1 remains ready for α-output and M_2 remains ready for α-input forever, or M_1 remains (from a certain point on) ready for β-input and M_2 remains ready for β-output forever, or $u = 1$ is achieved.

The inference of this disjunction can be based on the following case analysis: If $[\alpha \lessdot\,]$ does not occur then from $\varphi_{1.0}$ and $\varphi_{2.0}$, we obtain $\Box yearn([\alpha \lessdot\,]) \wedge \Box yearn([\alpha \gtrdot\,])$, which yields the first disjunct. If $[\alpha \lessdot\,]$ does occur, then by $\varphi_{1.1}$ and $\varphi_{2.1}$, we obtain $\Diamond[\beta \lessdot\,] \vee \Diamond\Box yearn[\beta \gtrdot\,]$ and $\Diamond[\beta \lessdot\,] \vee \Diamond\Box yearn[\beta \lessdot\,]$. If $[\beta \lessdot\,]$ does not occur then it follows that $\Diamond\Box yearn[\beta \gtrdot\,] \wedge \Diamond\Box yearn[\beta \lessdot\,]$ holds, which implies the second disjunct. If $[\beta \lessdot\,]$ does occur, then by $\varphi_{2.1}$, $[\beta \lessdot 1]$ occurs, which implies, by $\varphi_{1.1}$, the third disjunct $\Diamond(u = 1)$.

Here we introduce the following *eventual communication axiom*

$$\Box\Diamond\Big(ready([\gamma \lessdot\,]) \;\wedge\; ready([\gamma \gtrdot\,])\Big) \;\rightarrow\; \Box\Diamond[\gamma \lessdot\,]$$

stated for each channel γ in the system. This axiom reflects the compassion requirement by which, if a sender to γ and a receiver from γ are simultaneously ready infinitely many times, then there should be infinitely many γ-communications. We can use this axiom to derive the following *matchmaker postulate*

$$\neg\Diamond\Box\Big(yearn[\gamma \lessdot\,] \;\wedge\; yearn[\gamma \gtrdot\,]\Big),$$

which disallows computations in which a sender and receiver continually yearn for one another (but never communicate) from a certain point on. If we apply this axiom with respect to α and β, it rules out the first two disjuncts,

$$\Box\Big(yearn([\alpha \lessdot\,]) \;\wedge\; yearn([\alpha \gtrdot\,])\Big) \quad \text{and}$$
$$\Diamond\Box\Big(yearn([\beta \lessdot\,]) \;\wedge\; yearn([\beta \gtrdot\,])\Big),$$

in the disjunction.

Thus, with the help of the matchmaker postulate, which is always valid, we may infer $\diamond(u = 1)$ from the conjunction $\varphi_1 \wedge \varphi_2$.

5 Example: A Resource Allocator

As our final example of a decomposition of a global specification into modular ones, we consider a system that manages the allocation of a single resource among several competing processes.

We first present a global specification of this system

5.1 Resource Allocator: A Global Specification

The Resource Allocator system presented in Fig. 6 consists of an allocator (or arbiter) process A and m customer processes $C_1, \ldots, C_m$.

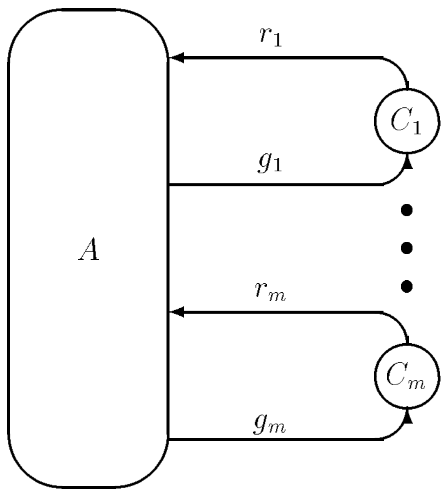

Figure 6: Resource Allocator system.

The resource can represent, for example, a shared disk or a printer. In the simple case we consider here, there is only one indivisible resource, but more general cases can be specified as well.

Each customer C_i communicates with allocator A via two shared boolean variables r_i ("request") and g_i ("grant"). Representing them in the diagram as directed edges implies that C_i may modify r_i but may only read g_i, while A may modify g_i but only read r_i. We assume that the initial value of all these boolean variables is F.

The expected protocol between C_i and A is the following:

C_i sets r_i to T, signaling a request for the resource.
A sets g_i to T, signaling that the resource has been granted to C_i.
C_i resets r_i to F, signaling a release of the resource.
A resets g_i to F, acknowledging the release and permitting C_i to submit subsequent requests.

It is possible to define events corresponding to the rising and falling of boolean variables. For a boolean variable b, we define

$$b\uparrow \; = \; b \;\wedge\; \circleddash(\neg b)$$
$$b\downarrow \; = \; (\neg b) \;\wedge\; \circleddash b.$$

The event of b rising occurs in a position $j > 0$, such that $b = \text{F}$ at $j-1$ and $b = \text{T}$ at j. Similarly, b falls at position $j > 0$, if $b = \text{T}$ at $j-1$ and $b = \text{F}$ at j.

5.2 Safety

The safety part of the specification consists of the following requirements:

- **Mutual Exclusion**

This property states that, at any instant, the resource is given to at most one customer.

One way of specifying this property is by the formula

$$\Box\Big(\bigwedge_{i\neq j}\neg(g_i \wedge g_j)\Big),$$

which claims that it is impossible to have both g_i and g_j, for $i \neq j$, true at the same state.

An alternative description utilizes *arithmetization* of boolean values, by which T is interpreted as 1 and F as 0. With this interpretation, we can state the mutual exclusion property as

$$S_1 \;:\; \Box(\textstyle\sum g_i \leq 1),$$

which states that at most one g_i can be true (equal 1).

- **Conformance with the Protocol**

The next list of requirements ensures that both the allocator and the customers adhere to the protocol. This means that the changes in the variables r_i, g_i, always occur in the cyclic order:

$$r_i\uparrow \qquad g_i\uparrow \qquad r_i\downarrow \qquad g_i\downarrow,$$

where the first event is $r_i\uparrow$.

The expected behavior of variables r_i and g_i is presented in Fig. 7. The diagram contains four states corresponding to the different possible values of the communication variables r_i and g_i. The transitions drawn between these states represent the possible sequence of changes. We have labeled each transition by the rising or falling event corresponding to the variable that changes.

We now present four formulas that characterize the changes in the values of the communication variables allowed by the protocol. To characterize the next change allowed from a state $(\neg r_i \wedge \neg g_i)$ we can use the formula

$$(\neg r_i \;\wedge\; \neg g_i) \;\Rightarrow\; (\neg r_i \;\wedge\; \neg g_i)\;\mathcal{W}\;(r_i \;\wedge\; \neg g_i).$$

However, a shorter equivalent formula is given by

$$S_2 \;:\; \neg g_i \;\Rightarrow\; (\neg g_i)\;\mathcal{W}\;(r_i \;\wedge\; \neg g_i).$$

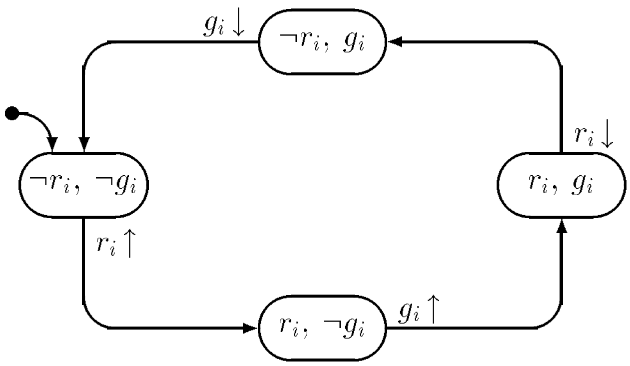

Figure 7: Expected protocol for C_i.

This formula states that from the time $g_i = \text{F}$, i.e., the resource is not granted to C_i, g_i will not change to T unless C_i makes a request by turning r_i to T first. This is another example of an absence of unsolicited response property.

Similar formulas complete the specification of conformance with the protocol:

$$S_3 \ : \ r_i \ \Rightarrow \ r_i \,\mathcal{W}\, (r_i \ \wedge \ g_i).$$

The formula states that, from the time C_i makes a request, the request is kept up at least until A responds.

$$S_4 \ : \ g_i \ \Rightarrow \ g_i \,\mathcal{W}\, (\neg r_i \ \wedge \ g_i).$$

This formula states that the resource will not be prematurely withdrawn, i.e., A will not reset g_i to F until after C_i releases the resource by resetting r_i to F.

$$S_5 \ : \ \neg r_i \ \Rightarrow \ (\neg r_i) \,\mathcal{W}\, (\neg r_i \ \wedge \ \neg g_i).$$

The formula states that from the time C_i releases the resource, it will not make another request (by setting r_i to T) until after A acknowledges the release by resetting g_i to F.

- **Bounded Overtaking**

It is not an essential part of the specification, but we may want to impose the requirement of 1-bounded overtaking. This property states that, from the time C_i makes a request, C_j may be granted the resource ahead of C_i at most once, for any $j \neq i$.

This property can be specified by the formula

$$r_i \ \Rightarrow \ (\neg g_j)\,\mathcal{W}\,g_j\,\mathcal{W}\,(\neg g_j)\,\mathcal{W}\,g_i \qquad \text{for every } i,\ j,\ j \neq i.$$

5.3 Progress

The progress part of the specification forces both the allocator and the customers to take some vital actions. As usual, all the safety properties can be satisfied by a system in which neither the allocator nor the customers ever modify a variable. They are also satisfied by a system in which the customers make requests, consistent with the protocol, but the allocator never responds. The role of the progress part is to exclude such trivial solutions.

- **Response to Requests**

This property ensures that every request for the resource is eventually honored. It is expressed by the formula

$$R_1 \;:\; r_i \;\Rightarrow\; \diamond g_i,$$

which states that every r_i request must be followed by a g_i grant.

- **Release of Resource**

The previous requirement is an obligation of the allocator to ensure good service. However, some cooperation from the customers is also required. Some of this cooperation was already requested in the safety part, mainly, in the requirement of conformance with the protocol. The progress requirement for the customers is that a customer that has the resource will eventually release it. Clearly, if a customer C_i appropriates the resource and never releases it, the allocator cannot guarantee service to another customer C_j, $j \neq i$, without violating the mutual exclusion requirement. The customer's progress obligation is specified by the formula

$$R_2 \;:\; g_i \;\Rightarrow\; \diamond \neg r_i.$$

This formula states that once granted the resource, the customer must eventually release it.

- **Response to Releases**

An equally important allocator responsibility is the assurance of response to notification of release. Due to the safety requirements, customer C_i cannot make a next request until its previous release was acknowledged by the allocator resetting g_i to F. Consequently, a devious way for the allocator to withhold service is to avoid acknowledging a release. The following requirement disallows such behavior:

$$R_3 \;:\; \neg r_i \;\Rightarrow\; \diamond \neg g_i.$$

This formula states that once the customer notifies a release, the allocator must respond by resetting g_i to F.

A Single Requirement

The previous three progress properties can be unified into a single requirement. This requirement identifies a state as *quiescent*, relative to C_i, if it satisfies $\neg r_i \wedge \neg g_i$. A single requirement that can replace requirements $R_1 - R_3$ states that the computation visits quiescent states infinitely many times, relative to each C_i. This can be stated by

$$R \;:\; \square\diamond(\neg r_i \;\wedge\; \neg g_i).$$

This formula requires, for each i, that there are infinitely many states in which C_i is quiescent, i.e., there are no pending requests and no grants to C_i.

Formula R alone is not equivalent to the conjunction $R_1 \wedge R_2 \wedge R_3$. However, under the assumption $S_2 \wedge S_3 \wedge S_4 \wedge S_5$, which is part of the safety component of the specification, R and $R_1 \wedge R_2 \wedge R_3$ are equivalent.

5.4 Modular Decomposition for the Resource Allocator

We can view the system as consisting of several modules as depicted in Fig. 8. Module A represents the allocator, while the array of modules $C_1, \ldots, C_m$ represents the customers.

$$A :: \left[\begin{array}{lll} \textbf{module} & & \\ \textbf{external in} & r & : \textbf{array } [1..m] \textbf{ of boolean} \\ \textbf{out} & g & : \textbf{array } [1..m] \textbf{ of boolean where } g = \mathrm{F} \\ & & \qquad\quad \text{Body}_A \end{array}\right]$$

$$\|$$

$$\overset{m}{\underset{i=1}{\|}} \; C[i] :: \left[\begin{array}{lll} \textbf{module} & & \\ \textbf{external in} & g_i & : \textbf{boolean} \\ \textbf{out} & r_i & : \textbf{boolean where } r_i = \mathrm{F} \\ & & \quad \text{Body}_{C_i} \end{array}\right]$$

Figure 8: Decomposition of the system into modules.

We proceed to present modular specifications for the customers and allocator that, together, will imply the global specification represented earlier.

Modular Specification for Customer C_i

Let us consider customer C_i. In Fig. 9 we present a possible implementation of a customer module C_i. This implementation is not intended as a binding representation but rather as a yardstick against which we can measure the feasibility of the modular specification we are about to develop.

The program contains a noncritical section in which activity that does not need the resource is carried out and a critical section in which activity that needs the resource is performed. The communication protocol involving r_i and g_i is self-explanatory and obviously maintains the cyclic order of the events

$$r_i \uparrow \qquad g_i \uparrow \qquad \text{Critical-activity} \qquad r_i \downarrow \qquad g_i \downarrow .$$

A possible methodology for constructing a modular specification, for a case in which the global specification is given as a list of simple requirements, is to inspect the requirements one by one and try to determine whether the considered module is the one responsible for maintaining this requirement. There is no simple recipe for detecting which module is responsible for maintaining a given property.

A very useful heuristic is available when we can view the property φ as constraining the behavior of a single variable, say, x. In this case the module responsible for maintaining φ is the one that can write x (assuming there is only one such module). This heuristic can be generalized to the case that φ constrains the behavior of several variables, all of which can be written only by a single module.

$$C_i :: \left[\begin{array}{l} \textbf{module} \\ \textbf{external in} \quad g_i \ : \textbf{boolean} \\ \textbf{out} \qquad\quad r_i \ : \textbf{boolean where } r_i = \text{F} \\ \ell_0 : \left[\begin{array}{l} \ell_1\text{: } \textbf{while T do} \\ \quad \left[\begin{array}{l} \ell_2\text{: } \textbf{noncritical} \\ \ell_3\text{: } r_i := \text{T} \\ \ell_4\text{: } \textbf{await } g_i \\ \ell_5\text{: } \textbf{critical} \\ \ell_6\text{: } r_i := \text{F} \\ \ell_7\text{: } \textbf{await } \neg g_i \end{array}\right] \end{array}\right] : \widehat{\ell}_0 \end{array}\right]$$

Figure 9: A possible implementation of a customer.

A simple example of this heuristic is to determine that it is the responsibility of the allocator A to maintain the mutual exclusion property $\Box\left(\left(\sum_{i=1}^{m} g_i\right) \leq 1\right)$. This is because module A is the one that owns (can exclusively write to) the variables $g_1, \ldots, g_m$.

Once a clause is identified as the responsibility of the module we are considering, we may have to modify it. This modification usually makes explicit the cooperation by the other modules that is necessary in order to maintain the property.

The first global requirement that seems to fall under the responsibility of C_i is

$$r_i \;\Rightarrow\; r_i \,\mathcal{W}\, (r_i \;\wedge\; g_i).$$

This property states that if r_i is true (equals T) at some point, it remains true at least until g_i is true (if ever). This is obviously a constraint on the execution of statements that set r_i to F and therefore is one of the liabilities of C_i.

Unfortunately, as stated, the only way to implement this modular specification is never to set r_i to F. Consider, for example, the following segment of a module that attempts to implement the specification and yet set r_i to F.

external in g_i: **boolean**

$\vdots$

m_3: **await** g_i

m_4: $r_i := \text{F}$

m_5:

Clearly this segment does its best not to set r_i to F before it senses a true g_i. Nevertheless, this module also has the following computation (listing values of π, r_i and g_i):

$$\cdots \;\; s_j\text{: } \langle\{..., [m_3]\},\ \text{T},\ \text{T}\rangle \;\xrightarrow{m_3}\; s_{j+1}\text{: } \langle\{..., [m_4]\},\ \text{T},\ \text{T}\rangle \;\xrightarrow{\tau_E}$$

$$s_{j+2}\text{: } \langle\{..., [m_4]\},\ \text{T},\ \text{F}\rangle \;\xrightarrow{m_4}\; s_{j+3}\text{: } \langle\{..., [m_5]\},\ \text{F},\ \text{F}\rangle \;\rightarrow\; \cdots$$

Observe that the formula $r_i \rightarrow r_i \mathcal{W} (r_i \wedge g_i)$ does not hold at position $j+2$ of the computation.

Obviously, the intended meaning of the property we are discussing is to make sure that once C_i sets r_i to T, it maintains the value T at least until module A responds by setting g_i to T. Thus, it is sufficient to state this property only at the exact points at which r_i has just changed from F to T.

Consequently, we include in the modular specification of C_i the weaker requirement

$$\varphi_1[i] \;\;:\;\; r_i \uparrow \;\Rightarrow\; r_i \; \mathcal{W} \; (r_i \;\wedge\; g_i).$$

This formula requires the maintenance of r_i at T at least until g_i is true only from the points at which r_i has just changed from F to T.

An alternative equivalent formulation of this requirement is given by the nested unless formula

$$\neg r_i \;\Rightarrow\; (\neg r_i) \; \mathcal{W} \; r_i \; \mathcal{W} \; (r_i \;\wedge\; g_i).$$

This formula traces the behavior of r_i from a position at which it is false. It claims that, following such a position, r_i will remain false for a while (perhaps forever), but when it changes to true, it will remain true at least until g_i becomes true. It is not difficult to see that this formula is equivalent to $\varphi_1[i]$.

The next (and last) safety requirement that constrains a variable owned by C_i is

$$\neg r_i \;\Rightarrow\; (\neg r_i) \; \mathcal{W} \; (\neg r_i \;\wedge\; \neg g_i).$$

This property is complementary to the previous one. It requires that if r_i is currently false, it should remain false at least until g_i is (becomes) false. As before, the modular version of this property is stated only at points of falling r_i, i.e., points at which r_i has just changed from T to F

$$\varphi_2[i] \;\;:\;\; r_i \downarrow \;\Rightarrow\; (\neg r_i) \; \mathcal{W} \; (\neg r_i \;\wedge\; \neg g_i).$$

Next we inspect the list of global progress properties looking for properties that constrain variables owned by C_i. The only such property is

$$g_i \;\Rightarrow\; \Diamond \neg r_i,$$

claiming that C_i always releases the resource some time after it has been granted to it.

To make this into a modular specification, we observe that it is impossible to guarantee a response to g_i if it is not held true sufficiently long. One way of specifying this conditional response is by the formula

$$g_i \, \mathcal{W} (\neg r_i) \;\Rightarrow\; \Diamond \neg r_i.$$

This formula guarantees a response (resetting r_i to F) to a true g_i only if g_i remains true at least until the response is generated.

An equivalent formula, which is the one we add to the modular specification of C_i is

$$\varphi_3[i] \;\;:\;\; \Box \Diamond \neg (r_i \wedge g_i).$$

This formula is best interpreted as stating that it is the responsibility of process C_i to make sure that the system does not remain in a state satisfying $r_i \wedge g_i$ forever. There are two ways of getting out of such a state. The allocator may change g_i to F and then C_i is not required to do anything. However, if the allocator keeps g_i true long enough, which is its expected good behavior, then process C_i is obliged to eventually reset r_i to F in order to guarantee that the state $r_i \wedge g_i$ does not last forever.

The complete modular specification for C_i is given by

$$\varphi[i] \;:\; \neg r_i \;\wedge\; \varphi_1[i] \;\wedge\; \varphi_2[i] \;\wedge\; \varphi_3[i],$$

where the first conjunct $\neg r_i$ identifies the initial value of r_i as F.

Modular Specification for the Allocator

Next we construct a modular specification ψ for the allocator module A. Again we scan the list of global properties and include in ψ modified versions of properties that constrain variables owned by A. It is an encouraging sign if we collect into ψ modified versions of all the properties that have not been collected into $\varphi[i]$.

Clearly, the safety property of mutual exclusion is completely under the responsibility of module A since it refers only to variables owned by A. We therefore include in ψ the clause

$$\psi_1 \;:\; \Box\left(\left(\sum_{i=1}^{m} g_i\right) \;\leq\; 1\right).$$

From the remaining safety properties, the relevant ones for A are those that govern the changes of g_i. These are

$$\begin{aligned} \neg g_i \;&\Rightarrow\; (\neg g_i)\; \mathcal{W}\; (r_i \;\wedge\; \neg g_i) \\ g_i \;&\Rightarrow\; (\;\, g_i)\; \mathcal{W}\; (\neg r_i \;\wedge\; g_i). \end{aligned}$$

As discussed before, the corresponding modular specification should be stated only from the points of change. Consequently, we include in ψ the clauses

$$\begin{aligned} \psi_2[i] \;&:\; (\mathit{first} \;\vee\; g_i \downarrow) \;\Rightarrow\; (\neg g_i)\; \mathcal{W}\; (\;\; r_i \;\wedge\; \neg g_i) \\ \psi_3[i] \;&:\; \qquad\qquad\quad g_i \uparrow \;\;\Rightarrow\; (\;\; g_i)\; \mathcal{W}\; (\neg r_i \;\wedge\; \;\; g_i). \end{aligned}$$

Note that in ψ_2 we have also included the first position of the computation as a position from which g_i is required to stay false at least until r_i becomes true. This represents a requirement of absence of unsolicited response.

Going over to the progress properties, there are two global progress properties that can be viewed as constraining the behavior of g_i. They are

$$\begin{aligned} \neg r_i \;&\Rightarrow\; \Diamond \neg g_i \\ r_i \;&\Rightarrow\; \Diamond \;\; g_i. \end{aligned}$$

The first property requires that g_i is eventually reset to F in response to r_i being false. Taking into account the possibility that r_i may not stay false long enough to be sensed, we add to the specification the clause

$$\psi_4[i] \;:\; \Box\Diamond \neg(\neg r_i \wedge g_i).$$

This formula states that it is the responsibility of the allocator to guarantee that the system does not remain in a state satisfying $\neg r_i \wedge g_i$ forever. It implies that either customer C_i resets r_i to T prematurely, and then the allocator has no further obligation for this round, or C_i leaves r_i at F long enough, and then the allocator must respond by resetting g_i to F.

The situation is more complex for the second property, requiring that the resource is granted to C_i, i.e., g_i set to T, in response to a request from C_i, i.e., $r_i = $ T. Since the modular specification is supposed to also hold for misbehaving customers, consider the following situation. Assume that, some time before a request from C_i, customer C_j, for $j \neq i$, has acquired the resource and not released it yet. Consequently, currently $r_j = g_j = $ T. Assume that C_i has made a request, i.e., r_i is T, but the rebellious C_j refuses to release the resource. What can the allocator do? It cannot reset g_j to F without violating ψ_3. It cannot set g_i to T without violating ψ_1. It seems that under such a situation we should relieve module A from the obligation to grant the resource to C_i.

There are several equivalent ways to specify this conditional requirement. One candidate formula is

$$r_i \, \mathcal{W} \, g_i \quad \Rightarrow \quad \left(\Diamond \, g_i \;\vee\; \bigvee_{j \neq i} \Diamond \, \Box g_j \right).$$

This formula states that if C_i has made a sustained request, then either it will be granted the resource, or we can identify a rebellious C_j that holds the resource and refuses to release it forever. The latter case can be identified by g_j remaining true forever. Note that, due to ψ_4, the option of leaving g_j true forever cannot be misused by the allocator by preferring this option to granting the resource to C_i. This is because, by $\psi_4[j]$, the allocator can keep g_j true forever only if C_j does not reset r_j to F and hold it at F long enough for the allocator to respond.

An alternative formula for specifying this property is

$$\psi_5[i] \quad : \quad \Box \Diamond \, (\neg r_i \wedge g_i) \;\vee\; \bigvee_{j \neq i} \Diamond \, \Box g_j.$$

This formula, which is the one we adopt, states that it is the responsibility of the allocator to guarantee that a state in which C_i is requesting ($r_i = $ T) while the resource is free ($g_i = $ F for all $i = 1, \ldots, m$) does not last forever, i.e., there are infinitely many states in which either r_i is false or g_i is true. This can be guaranteed by either C_i withdrawing its request prematurely, or some process C_j hogging the resource and not releasing it, or if neither of these happen, by the allocator eventually granting the resource to C_i.

The complete specification for the allocator module is given by the conjunction

$$\psi \quad : \quad \psi_1 \;\wedge\; \bigwedge_{i=1}^{m} (\neg g_i \;\wedge\; \psi_2[i] \;\wedge\; \psi_3[i] \;\wedge\; \psi_4[i] \;\wedge\; \psi_5[i]).$$

The conjunct $\neg g_i$ identifies the initial value of g_i as F for every $i = 1, \ldots, m$.

It can now be confirmed that the conjunction of the modular specifications

$$\psi \;\wedge\; \bigwedge_{i=1}^{m} \varphi[i]$$

implies each of the global properties that were listed earlier.

Global Properties Should Sometimes be Decomposed

When we considered the global specification of the resource allocator system, we commented that it is possible to replace the three progress requirements R_1–R_3 with the single requirement

$$R \;:\; \Box\Diamond(\neg r_i \;\wedge\; \neg g_i).$$

While this provides a more succinct representation of the global specification it makes the decomposition into modular specifications more difficult. This is because a property such as R constrains at the same time variables owned by C_i and variables owned by A.

Indeed, if faced with a global specification whose only progress part is given by R, the first recommended step is to break R into the three requirements R_1–R_3, each of which constrains only one variable. Then we could follow the partitioning policy and decide which of the three should be allocated to C_i and which to A.

References

[AL90] M. Abadi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, Lecture Notes in Computer Science 430. Springer-Verlag, 1990.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. 16th ACM Symp. Theory of Comp.*, pages 51–63, 1984.

[BKP85] H. Barringer, R. Kuiper, and A. Pnueli. A compositional temporal approach to a *csp*-like language. In E.J. Neuhold and G. Chroust, editors, *Formal Models of Programming*, pages 207–227. IFIP, North Holland, 1985.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminancy, and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, New Jersey, 1976.

[dR85] W.-P. de Roever. The quest for compositionality — a survey of assertion-based proof systems for concurrent programs, part i: Concurrency based on shared variables. In E.J. Neuhold and G. Chroust, editors, *The Role of Abstract Models in Computer Science*, pages 181–206. IFIP, North Holland, 1985.

[HdR86] J. Hooman and W.-P. de Roever. The quest goes on: A survey of proof systems for partial correctness of CSP. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 343–395. Lec. Notes in Comp. Sci. 224, Springer-Verlag, 1986.

[Lam83] L. Lamport. What good is temporal logic. In R.E.A. Mason, editor, *Proc. IFIP 9th World Congress*, pages 657–668. North-Holland, 1983.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1981.

[MP83] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proc. 10th ACM Symp. Princ. of Prog. Lang.*, pages 141–154, 1983.

[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.

[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.