A Composition Theorem for Layered Systems*

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

A. Udaya Shankar
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

Abstract

We define interface, module and the meaning of M offers I, where M denotes a module and I an interface. For a module M and disjoint interfaces U and L, the meaning of M using L offers U is also defined. Let N be a module that interacts with module M across interface L. We prove the following composition theorem: If M using L offers U, and N offers L, then M interacting with N offers U. Since the composition of M and N is also a module, the theorem holds for an arbitrary number of modules organized in a linear hierarchy. This theorem provides a theoretical foundation for layered systems (e.g., computer networks) where each layer corresponds to a module in the theorem.

1. Introduction

A module in our theory may be a service provider, a service consumer, or both. Interactions between a module and its environment take place at interfaces. Occurrence of an interface event involves simultaneous participation by both the module and its environment, and is observable from both sides of the interface. The semantics of an interface is defined by a set of allowed sequences of interface events; each such sequence defines an allowed sequence of interactions between the module and its environment. A module is specified by a state transition system (and a set of fairness requirements).

For a module M and an interface I, we define the meaning of M offers I (see Section 2). Our definition is similar to—but not quite the same as—various definitions of M satisfies S in the literature, where S is a specification of M [1,3,5,6,7,9,12,13,14]. Most definitions of M satisfies S have this informal meaning: M satisfies S if every possible observation of M is described by S. Specific definitions, however, differ in many ways: (1) in whether interface events or states are observable, (2) in whether observations are finite or infinite sequences, (3) in the particular formalism for representing these sequences, and (4) in the method of interaction at an interface.

Two modules interacting across an interface are composed to become a single module by hiding the interface between them. In this respect, the composition of two modules in our theory is defined in a manner not unlike the approaches of CSP [5] and I/O automata [14]. There are, however, some basic differences between our theory and the theories of CSP and I/O automata. First, we have an explicit notion of two-sided interfaces. Second, the interaction method between a module and its environment is different in our theory. (See below.) Third, in developing our theory, our vision of how it should be applied is different from those in [5,14]; specifically, we are more interested in decomposing the specification of a complex system (e.g., the protocols of a network) than in composition per se. An elaboration on this point follows.

* The work of Simon S. Lam was supported by National Science Foundation grants no. NCR-8613338 and no. NCR-8990464. The work of A. Udaya Shankar was supported by National Science Foundation grants no. ECS-8502113 and no. NCR-8904350.
Suppose an interface $I$ has been specified through which a system provides services. Instead of designing and implementing a monolithic module $M$ that offers $I$, we would like to implement the system as a collection of smaller modules $\{M_i\}$ such that the composition of $\{M_i\}$ offers $I$. To achieve this objective, the following three-step approach may be used:

**Step 1.** Derive a set of interfaces $\{S_j\}$ from $I$, one for each module in the collection (decomposition step).

**Step 2.** Design modules individually, and prove that $M_i$ offers $S_j$, assuming that the environment of $M_i$ satisfies $S_j$, in some manner.

**Step 3.** Apply an inference rule (composition theorem) to infer from the proofs of Step 2 that the composition of $\{M_i\}$ offers $I$.

The above approach has the following highly-desirable feature: given interfaces $\{S_j\}$, each module can be designed and implemented individually. However, the decomposition step—i.e., deriving the interfaces $\{S_j\}$ from $I$—is not an easy task. Furthermore, to develop the approach into a valid method, the following problem has to be solved: formally, in general, the inference rules required in Step 3 uses circular reasoning, and may not be valid. To see this, consider modules $M$ and $N$ that interact across interface $I$. Each module guarantees some properties of $I$ only if its environment satisfies certain properties of $I$. However, module $M$ is part of the environment of module $N$, and module $N$ is part of the environment of module $M$.

The above problem was considered by Misra and Chandy [16] for processes that communicate by CSP primitives. They gave a proof rule for assumptions and guarantees that are restricted to safety properties. Using different models, Pinzelli [18] provided a proof rule and Abadi and Lamport [2] presented a composition principle that are more general than the rules of Misra and Chandy; in particular, while the class of interface properties is still restricted, it includes progress properties.

In thinking about an interface, we depart from the usual notion that it is an external "cover" that encloses a module. Instead, we think of an interface as being two-sided, namely: there is a user side of the interface, and a user on the other, with both the user's behaviors and the service provider's behaviors constrained by the same set of interface event sequences; in this respect, an interface is symmetric. However, in our definitions of $M$ offers $I$ and $M$ using $L$ offers $U$ (see Section 2), the user and the service provider of each interface have asymmetric obligations. By organizing modules hierarchically and having asymmetric obligations for each interface, circular reasoning is avoided.

\[ \text{user} \]
\[ \underline{\text{module } M} \]
\[ \underline{\text{module } N} \]
\[ \text{lower interface } L \]
\[ \text{upper interface } U \]

Figure 1. Module $M$ and its environment.

For example, consider module $M$ in Figure 1. It provides services to a user through interface $U$ while it uses services offered by another module through interface $L$. We refer to $U$ as the upper interface and $L$ as the lower interface of module $M$. Note that module $M$ is the user of interface $L$ and the service provider of interface $U$. Its environment consists of both the user of $U$ and the module that offers $L$.

Many practical systems have a hierarchical structure. In fact, almost all computer networks have layered protocol architectures. Each protocol layer—e.g., transport, data link—corresponds to a module in our composition theorem. (Note that each such restriction on how these entities are composed is found.

Our composition theorem provides the framework. We are assured that each layer will be designed, implemented, and modified without violating the interface it offers. If $U$ is satisfied, the interfaces of $M$ can be checked for validity.

The balance this paper is organized into two sections. In Section 2, the definition and specification formalism are presented. Upper and lower interfaces of a component are defined together with a proof that it satisfies $M$. In Section 3, the applications of our theory and notation for secure access control and secure access control can be found.

### 2. Theory

We first define some notation for infinite sequence $(e_0, e_1, \ldots)$, where $e_i$ are events, means a sequence $(e_0, e_1, \ldots)$.

**Definition.** An interface $I$ is defined by:

- **Events($I$)**, a set of events that $I$ emits.
- **Inputs($I$)**, a set of input events to $I$.
- **Outputs($I$)**, a set of output events from $I$.
- **AllowedEventSeqs($I$)**, a set of allowed event sequence of $I$.

Output events of $I$ are under the control of the user of $I$. The occurrence of an output event (this requirement will be referred to as simultaneous participation by both the user and the service provider) by one side cannot occur because the interface is asymmetric.

For a given interface $I$, define

\[ \text{SafeEventSeqs}(I) = \{w:\text{true} \} \]

which includes the empty sequence.

**Definition.** A state transition system $A$ is defined by:

- **States($A$)**, a set of states.
- **Initial($A$)**, a subset of States($A$).
- **Events($A$)**, a set of events.
- **Transition($A$)**, a subset of $\text{Events}(A)$ is an ordered pair.

A behavior of $A$ is a sequence $\sigma$ of events such that $\sigma \in \text{Initial}(A)$ and $(\sigma, \varepsilon)\in$ is a transition and Events($A$) may end in a state or an event.
Instead of designing a system as a collection of independent modules, the following steps can be taken:

1. Identification of module interfaces.
3. Development of module algorithms.
4. Integration of modules.

Each module is defined by its interfaces, which are described by a set of input and output events. The composition theorem provides a theoretical foundation for layered systems. With this theorem, we can ensure that each module in the system, say \( M \), with upper interface \( U \) and lower interface \( L \), can be designed, implemented, and modified independently. As long as the interfaces remain the same, \( M \) using \( L \) offers \( U \) is satisfied, the internals of \( M \) can change.

The balance of this paper is organized as follows. In Section 2, we present our theory in a general framework. In Section 3, the definitions and results are specialized to the relational notation [9], which is a specification formalism more suitable for practical application. As an example, we present in Section 4 the upper and lower interfaces of a connection management protocol. A specification of the protocol is given together with a proof that it satisfies \( M \) using \( L \) offers \( U \). Section 5 has some concluding remarks. (Nontrivial applications of our theory and notation to the specification and verification of protocols for concurrency control and secure access control can be found in [8] and [11] respectively.)

2. Theory

We first define a notion of a sequence. A sequence over \( E \), where \( E \) is a set, means a (finite or infinite) sequence \( (e_0, e_1, \ldots) \), where \( e_i \in E \) for all \( i \). A sequence over alternating \( E \) and \( F \), where \( E \) and \( F \) are sets, means a sequence \( (e_0, f_0, e_1, f_1, \ldots) \), where \( e_i \in E \) and \( f_i \in F \) for all \( i \).

Definition. An interface \( I \) is defined by:

- \( \text{Events}(I) \), a set of events that is the union of two disjoint sets, \( \text{Inputs}(I) \), a set of input events, and \( \text{Outputs}(I) \), a set of output events.
- \( \text{AllowedEventSeqs}(I) \), a set of sequences over \( \text{Events}(I) \), each of which is referred to as an allowed event sequence of \( I \).

Output events of \( I \) are under the control of the service provider of \( I \), and input events of \( I \) are under the control of the user of \( I \). The occurrence of an interface event can only be initiated by the side with control. (This requirement will be referred to as unilateral control.) Since the occurrence of an interface event requires simultaneous participation by both the service provider and user of \( I \), it is possible that an interface event initiated by one side cannot occur because the other side refuses to participate.

For a given interface \( I \), define

\[
\text{SafeEventSeqs}(I) = \{ w : w \text{ is a finite prefix of an allowed event sequence of } I \}
\]

which includes the empty sequence.

Definition. A state transition system \( A \) is defined by:

- \( \text{States}(A) \), a set of states.
- \( \text{Initial}(A) \), a subset of \( \text{States}(A) \), referred to as initial states.
- \( \text{Events}(A) \), a set of events.
- \( \text{Transitions}(e) \), a subset of \( \text{States}(A) \times \text{States}(A) \), for every \( e \in \text{Events}(A) \). Each element of \( \text{Transitions}(e) \) is an ordered pair of states \( (s_i, t_i) \) referred to as a transition of \( e \).

A behavior of \( A \) is a sequence \( \sigma = (s_0, e_0, s_1, e_1, \ldots) \) over alternating \( \text{States}(A) \) and \( \text{Events}(A) \) such that \( s_0 \in \text{Initial}(A) \) and \( (s_i, s_{i+1}) \) is a transition of \( e_i \) for all \( i \). A finite sequence \( \sigma \) over alternating \( \text{States}(A) \) and \( \text{Events}(A) \) may end in a state or an event. A finite behavior, on the other hand, ends in a state by definition.
The set of behaviors of $A$ is denoted by $\text{Behaviors}(A)$. The set of finite behaviors of $A$ is denoted by $\text{FiniteBehaviors}(A)$.

For $e \in \text{Events}(A)$, let $\text{enabled}_s(e) = \{ x : \text{for some state } i, (i, x) \in \text{Transition}_s(e) \}$. An event $e$ is said to be enabled in a state $s$ of $A$ iff $s \in \text{enabled}_s(e)$. An event $e$ is said to be disabled in a state $s$ of $A$ iff $s \notin \text{enabled}_s(e)$.

**Notation.** For any sequence $\sigma$ over alternating $\text{States}(A)$ and $\text{Events}(A)$, and for any set $E \subseteq \text{Events}(A)$, $\text{Image}(\sigma, E)$ denotes the sequence of events in $E$ obtained from $\sigma$ by deleting states and deleting events that are not in $E$.

**Definition.** A module $M$ is defined by:

- **Events** ($M$), a set of events that is the union of three disjoint sets:
  - $\text{Inputs}(M)$, a set of input events,
  - $\text{Outputs}(M)$, a set of output events, and
  - $\text{Internals}(M)$, a set of internal events.
- $\text{sts}(M)$, a state transition system with $\text{Events}(\text{sts}(M)) = \text{Events}(M)$.
- **Fairness requirements of $M$**, a finite collection of subsets of $\text{Outputs}(M) \cup \text{Internals}(M)$. Each subset is referred to as a fairness requirement of $M$.

**Convention.** For readability, the notation $\text{sts}(M)$ is abbreviated to $M$ wherever such abbreviation causes no ambiguity, e.g., $\text{States}(\text{sts}(M))$ is abbreviated to $\text{States}(M)$, $\text{enabled}_{\text{int}}(e)$ is abbreviated to $\text{enabled}_e(e)$, etc.

Let $F$ be a fairness requirement of module $M$. $F$ is said to be enabled in a state $s$ of $M$ iff, for some $e \in F$, $e$ is enabled in $s$. $F$ is disabled in a state $s$ iff $F$ is not enabled in $s$. In a behavior $\sigma = (s_0, e_1, s_1, e_2, s_2, \ldots, s_j, e_j, \ldots)$, we say that $F$ occurs in state $s_j$ iff $e_j \in F$. An infinite behavior $\sigma$ of $M$ satisfies $F$ iff $F$ occurs infinitely often or is disabled infinitely often in states of $\sigma$.

For module $M$, a behavior $\sigma$ is an allowed behavior iff for every fairness requirement $F$ of $M$, $\sigma$ is finite and $F$ is not enabled in its last state, or $\sigma$ is infinite and satisfies $F$. Let $\text{AllowedBehaviors}(M)$ denote the set of all allowed behaviors of $M$.

We are now in a position to formalize the notion of a module offers an interface. Consider module $M$ and interface $I$. Let $\sigma$ be a sequence over alternating states and events of module $M$.

**Definition.** $\sigma$ is allowed wrt $I$ iff $\text{Image}(\sigma, \text{Events}(I)) \in \text{AllowedEvents}(I)$.

**Definition.** $\sigma$ is safe wrt $I$ iff one of the following holds:
- $\sigma$ is finite and $\text{Image}(\sigma, \text{Events}(I)) \in \text{SafeEvents}(I)$.
- $\sigma$ is infinite and every finite prefix of $\sigma$ is safe wrt $I$.

In what follows, we use last($\sigma$) to denote the last state in finite behavior $\sigma$, and $\sigma \cdot$ to denote concatenation.

**Definition.** Given a module $M$ and an interface $I$, $M$ offers $I$ iff the following conditions hold:

- **Naming constraints:**
  - $\text{Inputs}(M) = \text{Inputs}(I)$ and $\text{Outputs}(M) = \text{Outputs}(I)$.
- **Safety constraints:**
  - For all $\sigma \in \text{FiniteBehaviors}(M)$, if $\sigma$ is safe wrt $I$, then

\[
\forall e \in \text{Outputs}(M): \text{last}(\sigma) \cdot e \in \text{SafeEvents}(I).
\]

**Progress constraints:**
- For all $\sigma \in \text{AllowedBehaviors}(M)$, $\sigma \cdot e$ is safe wrt $I$.

A module $M$ with upper interface $U$ and lower interface $L$ is consistent with the user of $U$ and the module $M$. In what follows, we use $\sigma^\cdot$ to denote $\sigma \cdot$.

**Definition.** Given module $M$ and interface $I$, let $\sigma$ be:

- **Naming constraints:**
  - $\text{Events}(U) \cap \text{Events}(L) = \emptyset$,
  - $\text{Inputs}(M) = \text{Inputs}(U) \cup \text{Outputs}(M) = \text{Inputs}(U) \cup \text{Outputs}(L)$.
- **Safety constraints:**
  - For all $\sigma \in \text{AllowedBehaviors}(M)$, $\forall e \in \text{Outputs}(M): \sigma \cdot e \in \text{SafeEvents}(I)$.
  - $\forall e \in \text{Inputs}(M): \sigma \cdot e$ is safe wrt $I$.
- **Progress constraints:**
  - For all $\sigma \in \text{AllowedBehaviors}(M)$, $\sigma$ is allowed wrt $I$.

The interface of $M$ using $L$ offers $U$ review the key elements that are common to:
- Module $M$ is required to satisfy requirements of its interface(s), $I$, and the two Safety constraints wrt $M$'s interface(s), the Progress of $M$ violates some safety requirement satisfy $M$ offers $I$ or $M$ using $L$ offers $I$.
- Module $M$ satisfies the safety requirement of $M$ is enabled to occur, the event $\sigma \cdot e$.
- Whenever an input event of $M$ (or event's occurrence.

For an input event whose occurrence occurs or let it occur. (In this respect, $e$ requires an I/O automaton to be always input).

The main difference between the definition. For module $M$ using interface $I$, if $I$ and the module that offers $I$ satisfies the program.

**Definition.** A finite set of modules $\{M_i\}$ as:

- $\text{Internals}(M_i) \cap \text{Events}(M_i) = \emptyset$, and
of $A$ is denoted by $\text{enabled}_A(\sigma)$. An event $e$ is in a state $s$ of $A$ iff $\text{enabled}_A(\sigma) \epsilon \text{enabled}_A(\sigma)$. 

and for any set of events and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and deleting states and delete
Notation. For a set of modules \( \{M_j : j \in J\} \), each state of their composition is a tuple \( s = (t_j : j \in J) \), where \( t_j \in \text{States}(M_j) \). We use image \( \text{image}(s, M_j) \) to denote \( t_j \).

Definition. The composition of a compatible set of modules \( \{M_j : j \in J\} \) is a module \( M \) defined as follows:

- \( \text{Events}(M) \) defined by:
  \[
  \text{Internals}(M) = \bigcup_{j \in J} \text{Internals}(M_j) \cup \bigcup_{j \in J} \text{Outputs}(M_j) \cap \bigcup_{j \in J} \text{Inputs}(M_j)
  \]
  \[
  \text{Outputs}(M) = \bigcup_{j \in J} \text{Outputs}(M_j) - \bigcup_{j \in J} \text{Inputs}(M_j)
  \]
  \[
  \text{Inputs}(M) = \bigcup_{j \in J} \text{Inputs}(M_j) - \bigcup_{j \in J} \text{Outputs}(M_j)
  \]

- \( \text{stx}(M) \) defined by:
  \[
  \text{States}(M) = \prod_{j \in J} \text{States}(M_j)
  \]
  \[
  \text{Initial}(M) = \prod_{j \in J} \text{Initial}(M_j)
  \]
  \[
  \text{Transitions}(e), \text{for all } e \in \text{Events}(M), \text{defined by: } (s, t) \in \text{Transitions}(e) \iff \forall j \in J,
  \text{if } e \in \text{Events}(M_j) \text{ then } (\text{image}(s, M_j), \text{image}(t, M_j)) \in \text{Transitions}(e), \text{and}
  \text{if } e \notin \text{Events}(M_j) \text{ then } \text{image}(s, M_j) = \text{image}(t, M_j).
  \]

- Fairness requirements of \( M = \bigcup_{j \in J} \text{Fairness requirements of } M_j \).

Theorem 1. Let modules, \( M \) and \( N \), and interfaces, \( U \) and \( L \), satisfy the following:

- \( \text{Internals}(M) \cap \text{Internals}(N) = \emptyset \)
- \( M \) using \( L \) offers \( U \)
- \( N \) offers \( L \)

Then, \( M \) and \( N \) are compatible and their composition offers \( U \).

A proof of Theorem 1 can be found in [10]. It is quite long, requiring the proof of several lemmas, and is omitted due to space limitations. Theorem 1 is at the heart of our approach to compose modules hierarchically.

Theorem 2. Let \( M_1, I_1, M_2, I_2, \ldots, M_n, I_n \) be a finite sequence over alternating modules and interfaces, such that the following hold:

- For all \( j, k \), if \( j \neq k \) then \( \text{Events}(I_j) \cap \text{Events}(I_k) = \emptyset \) and \( \text{Internals}(M_j) \cap \text{Internals}(M_k) = \emptyset \).
- \( M_j \) offers \( I_j \).
- For \( j = 2, \ldots, n \), \( M_j \) using \( I_{j-1} \) offers \( I_j \).

Then, modules \( M_1, \ldots, M_n \) are compatible and their composition offers \( I_n \).

Using Theorem 1, a proof of Theorem 2 is straightforward and is omitted. It is also straightforward to generalize Theorem 1 to a set of modules organized as the nodes of a rooted tree; see [10].

3. Relational Specifications

In this section, we give a brief introduction to the specification of state transition systems, modules and interfaces in the relational notation. Some of the definitions and results in Section 2 are recast in this notation. For a complete treatment, see [8] and also [9].

The state space of a state transition system \( \text{stx} \) of a state transition system \( A \), the set of variables \( \text{domain}(v) \) of allowed values. By definition, \( \text{domain} \) represents a tuple of values, \( (d_v : v \in V) \).

We use state formulas to represent a set of states that are true or false when \( \text{Variables} \) represents the set of states for which \( \text{formula} \) evaluates to true for \( s \).

We use event formulas to specify \( \text{Variables}(A) \cup \text{Variables}(A') \), where \( V \) is the ordered pair \((s, t) \in \text{States}(A) \times \text{States}(A') \) for each formula, that is, the formula evaluates to true for \( s \).

Definition. A state transition system \( A \) is a set of events.

- \( \text{Events}(A) \), a set of events.
- \( \text{Variables}(A) \), a set of state variables.
- \( \text{Initial}(A) \), a state formula specifying the initial state.
- For every event \( e \in \text{Events}(A) \), a formula \( \text{Enabled}(e) \).

Note that for each event \( e \), we have \( \text{Enabled}(e) \equiv \exists \text{Variables}(A) \).

which is a state formula representing the set of variables \( \text{Variables}(A) \).

Definition. A module \( M \) is specified by:

- Disjoint sets of events, \( \text{Inputs}(M) \) and \( \text{Outputs}(M) \).
- \( \text{stx}(M) \), a state transition system.
- Fairness requirements of \( M \), a finite sequence over alternating events.

To specify an interface in the relational notation, assert the following assertions. Invariant assertions are sequences over alternating states and satisfy \( \text{Invariants} \) is a finite sequence over alternating states.

We use leads-to assertions, the finite sequence \((s_0, e_0, s_1, e_1, \ldots) \) over alternating states, where there exists \( j, 2j \) such that \( s_j \) satisfies \( G \).

Invariant and leads-to assertions are for a formalization that is either atomic assertion or infinite quantifiers. Let \( \sigma \) denote a sequence over \( \exists ! \text{variables} \).

We use formulae to express a well-formed formula.\(^1\)

\(^1\) leads-to is the only temporal connective we use.
The state space of a state transition system is specified by a set of variables, called state variables. For a state transition system $A$, the set of variables is denoted by $\text{Variables}(A)$. For each variable $v$, there is a set $\text{domain}(v)$ of allowed values. By definition, $\text{States}(A) = \prod_{v \in \text{Variables}(A)} \text{domain}(v)$. Each state $s \in \text{States}(A)$ is represented by a tuple of values, $(d_v : v \in \text{Variables}(A))$, where $d_v \in \text{domain}(v)$.

We use state formulas to represent subsets of $\text{States}(A)$. A state formula is a formula in $\text{Variables}(A)$ that evaluates to true or false when $\text{Variables}(A)$ is assigned $s$, for every state $s \in \text{States}(A)$. A state formula represents the set of states for which it evaluates to true. For state $s$ and state formula $P$, $s$ satisfies $P$ iff $P$ evaluates to true for $s$.

We use event formulas to specify the transitions of events. An event formula is a formula in $\text{Variables}(A) \cup \text{Variables}(A)'$, where $\text{Variables}(A)' = \{v' : v \in \text{Variables}(A)\}$ and $\text{domain}(v') = \text{domain}(v)$. The ordered pair $(s, t) \in \text{States}(A) \times \text{States}(A)$ is a transition specified by an event formula iff $(s, t)$ satisfies the formula, that is, the formula evaluates to true when $\text{Variables}(A)$ is assigned $s$ and $\text{Variables}(A)'$ is assigned $t$.

**Definition.** A state transition system $A$ is specified in the relational notation by:

- $\text{Events}(A)$, a set of events.
- $\text{Variables}(A)$, a set of state variables, and their domains.
- $\text{Initials}_A$, a state formula specifying the initial states.
- For every event $e \in \text{Events}(A)$, an event formula $\text{formula}_A(e)$ specifying the transitions of $e$.

Note that for each event $e$, we have

$$\text{enabled}_A(e) = \exists \mathcal{V} \text{Variables}(A) : \text{formula}_A(e)$$

which is a state formula representing the set of states where $e$ is enabled.

**Definition.** A module $M$ is specified in the relational notation by:

- Disjoint sets of events, $\text{Inputs}(M)$, $\text{Outputs}(M)$, and $\text{Internals}(M)$, with $\text{Events}(M)$ being their union.
- $\text{sts}(M)$, a state transition system with $\text{Events}(	ext{sts}(M)) = \text{Events}(M)$, specified in the relational notation.
- Fairness requirements of $M$, a finite collection of subsets of $\text{Outputs}(M) \cup \text{Internals}(M)$.

To specify an interface in the relational notation, we use a state transition system together with invariant and progress assertions. Invariant assertions are of the form: $\text{Invariant } P$, where $P$ is a state formula. A finite sequence over alternating states and events satisfies $\text{invariant } P$ iff every state in the sequence satisfies $P$. An infinite sequence over alternating states and events satisfies $\text{invariant } P$ iff every finite prefix of the sequence satisfies $\text{invariant } P$.

We use leads-to assertions of the form: $P$ leads-to $Q$, where $P$ and $Q$ are state formulas. A sequence $(s_0, s_1, s_2, \cdots)$ over alternating states and events satisfies $P$ leads-to $Q$ iff for all $i$: if $s_i$ satisfies $P$ then there exists $j$, $j \geq i$, such that $s_j$ satisfies $Q$.

Invariant and leads-to assertions are collectively referred to as atomic assertions. In what follows, an assertion is either an atomic assertion or one constructed from atomic assertions using logical connectives and quantifiers. Let $\sigma$ denote a sequence over alternating states and events. An assertion is true for $\sigma$ iff $\sigma$ satisfies

---

1 We use formula to mean a well-formed formula in the language of predicate logic.
2 leads-to is the only temporal connective we use.
the assertion. For a given σ, to evaluate the truth value of an assertion, say \textit{Assert}, we first evaluate for σ the truth value of every atomic assertion within \textit{Assert}. For example, σ satisfies the assertion \(X \land Y \Rightarrow Z\), where \(X, Y\), and \(Z\) are atomic assertions, iff (σ satisfies \(X\)) \(\land\) (σ satisfies \(Y\)) \(\Rightarrow\) (σ satisfies \(Z\)).

A safety assertion is an assertion constructed from invariant assertions only. A state transition system satisfies a safety assertion iff every finite behavior of the state transition system satisfies the safety assertion. A \textit{progress assertion} is an assertion constructed from atomic assertions that include at least one leads-to assertion.

A module satisfies a progress assertion iff every allowed behavior of the module satisfies the progress assertion.

To use a state transition system, say \(A\), for specifying an interface, we need to exercise care in defining the events of \(A\). First, \(A\) cannot have internal events. Second, the input and output events must be defined such that they have "adequate resolution." A sufficient condition is the following:

**Definition.** A state transition system \(A\) has deterministic events iff

- \(\text{Internals}(A) = \emptyset\),
- \(\text{Init}(A)\) is a single state, and
- for all \(e \in \text{Events}(A)\), \(\text{Transition}_{\text{A}}(e)\) is a partial function, i.e., for all \(s \in \text{States}(A)\), there is at most one state \(s'\) such that \((s, s') \in \text{Transition}_{\text{A}}(e)\).

This condition is easy to satisfy because events in our theory can be regarded as names or labels. (Moreover, event names can be parameterized in the relational notation [9].) The condition implies that each event sequence represents at most one behavior of \(A\) because event occurrences have deterministic effects. Behaviors of \(A\), however, are nondeterministic because more than one event can be enabled in a state.

As an observation, note that the restriction of a single initial state may be circumvented as follows (if needed): Let \(s_0\) denote a state not in \(\text{States}(A)\), and \(\text{Init}(A)\) the desired initial states of \(A\). Define \(\text{Init}(A)\) to be \(\{s_0\}\) and, for all \(s \in \text{Init}(A)\), specify a distinct event for each transition \((s, s')\) in it with \(v^\prime\).

**Notation.** For any state formula \(R\), we use \(R'\) to denote the formula obtained from \(R\) by replacing every state variable \(v\) in it with \(v^\prime\).

**Definition.** An interface \(I\) is specified in the relational notation by:

- Disjoint sets of events, \(\text{Inputs}(I)\) and \(\text{Outputs}(I)\), with \(\text{Events}(I)\) being their union.
- \(\text{sts}(I)\), a state transition system with deterministic events specified in the relational notation such that \(\text{Events}(\text{sts}(I)) = \text{Events}(I)\).
- \(\text{InvAssum}_I\), a conjunction of state formulas referred to as \textit{invariant assumptions} of \(I\), such that \(\text{Initial}(I) = \text{InvAssum}_I\), and
  \(\forall e \in \text{Outputs}(I): \text{InvAssum}_I \land \text{formula}(e) \Rightarrow \text{InvAssum}_I'\)
- \(\text{InvGuar}_I\), a conjunction of state formulas referred to as \textit{invariant guarantees} of \(I\), such that \(\text{Initial}(I) \Rightarrow \text{InvGuar}_I\), and
  \(\forall e \in \text{Inputs}(I): \text{InvGuar}_I \land \text{formula}(e) \Rightarrow \text{InvGuar}_I'\)
- \(\text{ProgReq}_I\), a conjunction of progress assertions, referred to as \textit{progress requirements} of \(I\).

The invariant assumptions and guarantees of interface \(I\) are collectively referred to as \textit{invariant requirements} of interface \(I\). Define

\[\text{InvReq}_I = \text{InvAssum}_I \land \text{InvGuar}_I.\]

Given an interface \(I\) specified in the relational notation, an allowed event sequence of \(I\) is the sequence of events in a behavior of \(\text{sts}(I)\) that satisfies all invariant and progress requirements; more precisely, define

\[\text{AllowedBehaviors}(I) = \{\sigma: \sigma \in B \land \text{InvReq}_I \land \text{ProgReq}_I\}\]

\[\text{AllowedEventSeq}(I) = \{\text{image}(\sigma) : \sigma \in \text{AllowedBehaviors}(I)\}\]

Lastly, for event \(e \in \text{Events}(I)\), define

\[\text{possible}(e) = \text{InvReq}_I \land \text{ProgReq}_I\]

which is a state formula representing the set of \textit{possible} event \(e\).

Note that we have provided two ways of specifying a state transition system, and a set of invariant requirements can be more easily expressed by invariant requirements encoded in a state transition system [8].

For modules and interfaces specified \textit{offers} \(I\) and \(M\) using \textit{uses} \(U\). We first introduce \(B\) such that \(\text{Variables}(A) \supseteq \text{Variables}(B)\) defined as follows: state \(s \in \text{States}(B)\) is \textit{Valid} if \(s(\text{Variables}(B))\) in \(s\) (9,19). States \(\text{States}(A)\) using the projection mapping. A module \(M\) is specified directly over \(\text{States}(A) \supseteq \text{States}(A)\) using

**Definition.** Given state transition system \(B\) assuming \(\text{InvAssum}_B\) iff

- \(\text{Variables}(A) \supseteq \text{Variables}(B)\) and
- \(\text{Init}(A) = \text{Init}(B)\) and
- \(\forall e \in \text{Events}(B): \text{InvGuar}_B \land \text{formula}(e) \Rightarrow \text{InvGuar}_B'\)
- \(\forall e \in \text{Events}(A) \land \text{Events}(B): \text{InvGuar}_B\)
- \(\forall e \in \text{Events}(A) \land \text{Events}(B): \text{InvGuar}_B\)

If \(A\) is a refinement of \(B\) assuming \(\text{InvAssum}_I\) of \(B\) as defined in [9]. In this case, for any satisfies \textit{invariant} \(P\).

Given a module \(M\), an interface \(I\), and \(\text{InvAssum}_I\) of \(I\), as assertions, expressed in the relational notation, any

- \(B_1\ \text{Inputs}(M) \Rightarrow \text{Inputs}(I)\) and \(\text{Outputs}(M) \Rightarrow \text{Outputs}(I)\)
- \(B_2\ \text{sts}(M)\) is a refinement of \(\text{sts}(I)\)
- \(B_3\ \forall e \in \text{Inputs}(I): \text{InvGuar}_B \land \text{possible}(e)\)
- \(B_4\ \forall e \in \text{Outputs}(I): \text{InvGuar}_B \land \text{possible}(e)\)
- \(B_5\ \text{sts}(M)\) satisfies \(\text{InvAssum}_I\)
- \(B_6\ \text{sts}(M)\) satisfies \(\text{InvAssum}_I\).
AllowedBehaviors(\(I\)) = \{ \sigma \in \text{Behaviors}(I) \text{ and } \sigma \text{ satisfies invariant InvReqs and ProgReqs} \}.

AllowedEventSeqs(\(I\)) = \{ \text{image}(\sigma, \text{Events}(I)) : \sigma \in \text{AllowedBehaviors}(I) \}.

Lastly, for event \(e \in \text{Events}(I)\), define
\[
\text{possible}(e) = \text{InvReqs} \land \exists \text{Variables}(I) : \text{formula}(e) \land \text{InvReqs}
\]
which is a state formula representing the set of states in which event \(e\) can occur without violating any invariant requirement of \(I\).

Note that we have provided two ways to specify the safety requirements of an interface: namely, a state transition system, and a set of invariant requirements. It is our experience that some safety requirements are more easily expressed by invariant requirements, while some are more easily expressed by allowed state transitions encoded in a state transition system [8]. Our approach is a flexible one.

For modules and interfaces specified in the relational notation, we provide sufficient conditions for \(M\) offers \(I\) and \(M\) using \(L\) offers \(U\). We first introduce a refinement relation between two state transition systems \(A\) and \(B\) such that Variables(\(A\)) \supseteq Variables(\(B\)). In this case, there is a projection mapping from States(\(A\)) to States(\(B\)) defined as follows: state \(s \in\) States(\(A\)) is mapped to state \(t \in\) States(\(B\)) where \(t\) is defined by the values of Variables(\(B\)) in \(s\) [7,9,19]. State formulas in Variables(\(B\)) can be interpreted directly over States(\(A\)) using the projection mapping. Also, event formulas in Variables(\(B\)) \cup Variables(\(B\)) can be interpreted directly over States(\(A\)) \supseteq States(\(A\)) using the projection mapping.

**Definition.** Given state transition systems \(A\) and \(B\) and state formula Inv\(_{A}\) in Variables(\(A\)), \(A\) is a refinement of \(B\) assuming Inv\(_{A}\) if

- Variables(\(A\)) \supseteq Variables(\(B\)) and Events(\(A\)) \supseteq Events(\(B\))
- \(\text{Init}_{A} \Rightarrow \text{Init}_{B}\)
- \(\forall e \in \text{Events}(B) : \text{Inv}_{A} \land \text{formula}_{A}(e) \Rightarrow \text{formula}_{B}(e)\)
- \(\forall e \in \text{Events}(A) \land \text{Events}(B) : \text{Inv}_{A} \land \text{formula}_{A}(e) \Rightarrow (\forall v \in \text{Variables}(B) : v=v)\)

If \(A\) is a refinement of \(B\) assuming Inv\(_{A}\) and, moreover, \(A\) satisfies invariant Inv\(_{A}\) then \(A\) is a refinement of \(B\) as defined in [9]. In this case, for any state formula \(P\) in Variables(\(B\)), if \(B\) satisfies invariant \(P\), then \(A\) satisfies invariant \(P\).

Given a module \(M\), an interface \(I\), and some state formula Inv\(_{M}\) in Variables(\(M\)), the following conditions, expressed in the relational notation, are sufficient for \(M\) offers \(I\):

- **B1** Inputs(\(M\))=Inputs(\(I\)) and Outputs(\(M\))=Outputs(\(I\))
- **B2** \(\text{sts}(M)\) is a refinement of \(\text{sts}(I)\) assuming Inv\(_{M}\)
- **B3** \(\forall e \in \text{Inputs}(I) : \text{Inv}_{M} \land \text{possible}(e) \Rightarrow \text{enabled}_{M}(e)\)
- **B4** \(\forall e \in \text{Outputs}(I) : \text{Inv}_{M} \land \text{formula}_{M}(e) \Rightarrow \text{InvGuar}_{I}\)
- **B5** \(\text{sts}(M)\) satisfies (InvAssum\(_{M}\) \Rightarrow inv\_variant Inv\(_{M}\))
- **B6** \(M\) satisfies (InvAssum\(_{M}\) \Rightarrow ProgReqs)
Theorem 3. For a module $M$, an interface $I$, and some state formula $\varphi_M$ in Variables($M$), if conditions B1-B6 hold, then

(a) $M$ offers $I$, and

(b) $\forall \sigma \in \text{Behaviors}(M): \sigma$ satisfies invariant $\varphi_M \Rightarrow \sigma$ is safe wrt $I$.

Given an interface $I$, to obtain a module $M$ that offers $I$, we make use of B1-B6 in three stages. First, the events of $\text{events}(M)$ are named such that B1 is satisfied. Second, events of $\text{events}(M)$ are specified such that $\text{events}(M)$ is a refinement of $\text{events}(I)$ (B2 is satisfied), each input event is enabled in states where the event's occurrence would be safe (B3 is satisfied), and $M$ satisfies its invariant guarantees (B4 is satisfied). Initially, $\varphi_M$ is equal to $\text{InvAssum}_M$. But to prove B2-B4, we may have to assume that $\text{events}(M)$ has additional invariant properties, which are used to strengthen $\varphi_M$ and must be proved (so that B5 is satisfied). Third, we try to prove B5 and B6.

For a module $M$, interfaces $U$ and $L$, and some state formula $\varphi_M$ in Variables($M$), the following conditions, expressed in the relational notation, are sufficient for $M$ using $L$ offers $U$:

$$C_1 \quad \text{Events}(U) \cap \text{Events}(L) = \emptyset$$
$$\text{Inputs}(M) = \text{Inputs}(U) \cup \text{Outputs}(L)$$
$$\text{Outputs}(M) = \text{Outputs}(U) \cup \text{Inputs}(L)$$
$$\text{Variables}(U) \cap \text{Variables}(L) = \emptyset$$

$$C_2 \quad \text{events}(M) \text{ is a refinement of } \text{events}(U) \text{ assuming } \varphi_M$$

$$C_3 \quad \text{events}(M) \text{ is a refinement of } \text{events}(L) \text{ assuming } \varphi_M$$

$$C_4 \quad \forall e \in \text{Inputs}(U): \varphi_M \land \text{possible}_L(e) \Rightarrow \text{enabled}_L(e)$$

$$C_5 \quad \forall e \in \text{Outputs}(L): \varphi_M \land \text{possible}_L(e) \Rightarrow \text{enabled}_L(e)$$

$$C_6 \quad \forall e \in \text{Inputs}(L): \varphi_M \land \text{formal}_L(e) \Rightarrow \text{InvAssum}_M$$

$$C_7 \quad \forall e \in \text{Outputs}(U): \varphi_M \land \text{formal}_L(e) \Rightarrow \text{InvGuar}_M$$

$$C_8 \quad \text{events}(M) \text{ satisfies (invariant } (\text{InvAssum}_M \land \text{InvGuar}_M) \Rightarrow \text{invariant } \varphi_M)$$

$$C_9 \quad \varphi_M \text{ satisfies (invariant } (\text{InvAssum}_M \land \text{InvGuar}_M) \land \text{ProgReq}_M \Rightarrow \text{ProgReq}_M)$$

Theorem 4. For a module $M$, interfaces $U$ and $L$, and some state formula $\varphi_M$ in Variables($M$), if conditions C1-C9 hold, then

(a) $M$ using $L$ offers $U$, and

(b) $\forall \sigma \in \text{Behaviors}(M): \sigma$ satisfies invariant $(\text{InvAssum}_M \land \text{InvGuar}_M) \Rightarrow \sigma$ is safe wrt $U$ and $L$.

Theorem 4 indicates that we can set $\varphi_M$ equal to $\text{InvAssum}_M \land \text{InvGuar}_M$, initially. However, to prove C2-C7 for a module $M$, we may have to assume that $\text{events}(M)$ has additional invariant properties, which are used to strengthen $\varphi_M$ and must be proved (so that C8 is satisfied).

Proofs of Theorems 3 and 4 can be found in [10].

For convenience, we employ a couple of conventions when we use the relational notation [9]. They are briefly reviewed below. Recall that an event formula defines a set of state transitions. Some examples of event definitions are shown below:

$$e_1 = v_1 \geq 2 \land v_2 \in [1, 2, 5]$$

$$e_2 = v_1 \geq 2 \land v_1 \geq 5$$

In each definition, the event name is given on the left-hand side of " = " and the event formula is given on the right-hand side.

Consider a state transition system A of this system. Note that $v_1$ does not occur free in $e_2$, not updated by the occurrence of $e_2$. The transition system is a free variable of formula(e), the conjunction.

If a parameter occurs free in an event parameter, for example, consider

$$e(m) = v_1 \geq 2 \land v_1 \geq 5 \land m$$

where $m$ is a parameter with a specified domain. It is used to specify a group of related events.

Lastly, in deriving a state transition system of B as defined above, we further require that the same domain of allowed values.

4. Example—A Connection Manager

We first present an interface $U$ specified by

Input events:

$$\text{ConnReq} = \text{State} \downarrow \text{Closed} \land \text{State} \downarrow \text{Active}$$

$$\text{ConnResp} = \text{State} \downarrow \text{PassiveOpening} \land \text{State} \downarrow \text{ActiveOpening}$$

Output events:

$$\text{ConnInd} = \text{State} \downarrow \text{Closed} \land \text{State} \downarrow \text{Active}$$

$$\text{Collision} = \text{State} \downarrow \text{ActiveOpening} \land \text{State} \downarrow \text{ActiveOpening} \land \text{State} \downarrow \text{ActiveOpening}$$

We show that conditions C1-C9 are satisfied.
Consider a state transition system $A$ with two state variables $v_1$ and $v_2$. Let $e_2$ above be an event of the system. Note that $v_1'$ does not occur free in formula $(e_2)$. By the following convention, it is assumed that $v_1$ is not updated by the occurrence of $e_2$.

**Convention.** Given an event formula, formula $(e)$, for every state variable $v$ in Variables ($A$), if $v'$ is not a free variable of formula $(e)$, the conjunct $v'=v$ is implicit in formula $(e)$.

If a parameter occurs free in an event's formula, then there is an event defined for every allowed value of the parameter. For example, consider

\[ e_2(m) = v_1 > v_2 \land v_1 + v_2 = m \]

where $m$ is a parameter with a specified domain of allowed values. A parameterized event is a convenient way to specify a group of related events.

Finally, in deriving a state transition system $A$ from a state transition system $B$, for $A$ to be a refinement of $B$ as defined above, we further require that every parameter of $B$ be a parameter of $A$ with the same name and same domain of allowed values.

4. Example—A Connection Management Protocol

We first present an interface $U$ specifying a connection management service between two access points, named 1 and 2. Suppose there is a user entity at each access point of interface $U$. Connections are asymmetric in that each connection established "belongs" to the user entity that requested the connection. Call collisions are resolved in favor of the user entity at access point 1. (This example is motivated by the call setup protocol between DTE and DCE in the packet layer of X.25.)

We then present an interface $L$ specifying a reliable message communication service between two access points, also named 1 and 2. (Note that the data link layer of X.25 provides a reliable communication service to the packet layer.)

We then specify a module $M$ that uses $L$ to offer $U$. The module consists of two protocol entities, 1 and 2, such that the events of protocol entity $i$ match the events of $U$ and $L$ at access points named $i$, for $i=1, 2$. We show that conditions C1--C9 are satisfied. Thus, the module satisfies $M$ using $L$ offers $U$.

4.1. Interface $U$ specifying connection management

We specify the state variables, initial condition, and events of interface $U$. The parameter $i$ ranges over 1 and 2. We use parameter $j$ to range over 1 and 2 such that $j\neq i$.

State variables:

\[ \text{State}_i = \{ \text{Closed}, \text{PassiveOpening}, \text{ActiveOpening}, \text{PassiveOpen}, \text{ActiveOpen} \} \] Initially Closed.

Input events:

\[ \text{ConnReq}_i = \text{State}_0 = \text{Closed} \land \text{State}_i = \text{ActiveOpening} \]
\[ \text{ConnResp}_i = \text{State}_0 = \text{PassiveOpening} \land \text{State}_i = \text{PassiveOpen} \]
\[ \text{DiscReq} = \text{State}_i = \text{ActiveOpening} \land \text{State}_i = \text{Closed} \]

Output events:

\[ \text{ConnInd}_i = \text{State}_0 = \text{Closed} \land \text{State}_i = \text{PassiveOpening} \]
\[ \text{Collision}_j = \text{State}_j = \text{ActiveOpening} \land \text{State}_j = \text{PassiveOpening} \]
ConnConf₁ = State₁=ActiveOpening ∧ State₂=ActiveOpening
DisConf₁ = State₁=PassiveOpen ∧ State₂=Closed

Note that the collision event is defined only for access point 2.

Invariant and progress requirements:

InvAssum₁ = true

InvGuar₁ = InvGuar₁₁ ∧ InvGuar₁₂,
where
InvGuar₁₁ = (State₁=ActiveOpen → State₁=PassiveOpen)
∧ (State₁=PassiveOpening → State₁=ActiveOpening)

The first conjunct of InvGuar₁₁ can be falsified only by the event ConnConf₁ (which makes the antecedent true) and the event DisConf₁ (which makes the consequent false). The second conjunct of InvGuar₁₂ can be falsified only by the event ConnConf₁ (which makes the antecedent true), the event ConnConf₂ (which makes the consequent false) and the event Collision₁ (which makes the consequent false for i=1, and the antecedent true for i=2). Note that all these events are output events of U. Input events of U do not falsify InvGuar₁, as required by our definition of a relationally-specified interface.

ProgReq₁ =
((State₁=PassiveOpening leads-to State₁=PassiveOpen)
→ (State₁=ActiveOpening leads-to State₁=ActiveOpen))
∧ ((State₁=PassiveOpening leads-to State₁=PassiveOpen)
→ (State₁=PassiveOpening leads-to State₁=PassiveOpen))

(4.2. Interface L specifying reliable message delivery)

We specify the state variables, initial condition, and events of interface L.

State variables:

Sentᵢ: sequence of messages. Initially the null sequence.
Receivedᵢ: sequence of messages. Initially the null sequence.

Sentᵢ is the sequence of messages that have been sent at access point i since the beginning of execution.
Receivedᵢ is the sequence of messages that have been received at access point i since the beginning of execution. Below, the parameter i ranges over 1 and 2.

Input events:
Sendᵢ(m) = Sentᵢ∪Sentᵢ@(m)

Output events:
Recᵢ(m) = Receivedᵢ∪Receivedᵢ@(m)

Invariant and progress requirements:

InvAssumᵢ = true

InvGuarᵢ = (Receivedᵢ prefix-of Sentᵢ) ∧ (Receivedᵢ prefix-of Sentᵢ)

ProgReqᵢ = ((Sentᵢ)≥k leads-to (Recᵢ))

Note that input events of L do not falsify InvGuarᵢ.

4.3. Module M

The module M consists of two protocol entities and events of the protocol entities because a connect request, disc denoting a disconnect request.

State variables of protocol entity i:

Stateᵢ; <as defined in upper interface U >
Sentᵢ; <as defined in lower interface Sᵢ; null, connsᵢ, connsᵢ', acksᵢ, acksᵢ', discsᵢ, discsᵢ'
Sᵢ=null indicates that protocol entity i does not have to send a conns message. Sᵢ=connsᵢ indicates that entity i must execute an appropriate output event. The conditions for disc (and ack) messages. The disc message followed by a conns message; i.e., the local user entity issued a disconnect request that the other could handle the disconnect request. The value of the message followed by a conns message, for which the equalities are not.

Events of protocol entity i:

We first specify module events that make up L. For an interface event eᵢ, the formula for the interface event formula is g is a formula that the state change to any interface variable is specified by an entity i do not access state variables of protocol

ConnReq₁ = formulai₁(ConnReq₁) ∧ (Sᵢ;)
ConnReq₂ = formulai₂(ConnReq₂) ∧ ((Sᵢ=discᵢ ∧ Sᵢ'=discᵢ ∧ connᵢ;)
ConnRespᵢ = formulaiᵢ(ConnRespᵢ) ∧ (Sᵢ=connsᵢ;)
DiscReqᵢ = formulaiᵢ(DiscReqᵢ) ∧ (Sᵢ=discᵢ;)
ConnIndᵢ = formulaiᵢ(ConnIndᵢ) ∧ (Sᵢ=connsᵢ;
Collisionᵢ = formulaiᵢ(Collisionᵢ) ∧ (Sᵢ=discᵢ;)
ConnConfᵢ₁ = formulaiᵢ(ConnConfᵢ₁) ∧ (Sᵢ;
DiscIndᵢ₁ = formulaiᵢ(DiscIndᵢ₁) ∧ (Sᵢ=disconsᵢ;)
Sendᵢ(connsᵢ) = formulaiᵢ(Sendᵢ(connsᵢ)) ∧ (Sᵢ;
Sendᵢ(ackᵢ) = formulaiᵢ(Sendᵢ(ackᵢ)) ∧ (Sᵢ;
Sendᵢ(discᵢ) = formulaiᵢ(Sendᵢ(discᵢ)) ∧ (Sᵢ;
Recᵢ(connsᵢ) = formulaiᵢ(Recᵢ(connsᵢ)) ∧ (Sᵢ;
Sendᵢ(discᵢ) = formulaiᵢ(Sendᵢ(discᵢ))
\[\text{ProgReq}_L = (|\text{Sent}_1| \geq k \text{ leads-to } |\text{Received}_1| \leq 2k) \land (|\text{Sent}_2| \geq l \text{ leads-to } |\text{Received}_2| \leq 2l)\]

Note that input events of \(L\) do not falsify InvGuar_L.

4.3. Module M

The module \(M\) consists of two protocol entities, named 1 and 2. We specify the state variables, initial condition, and events of the protocol entities below. The protocol uses three types of messages, \texttt{conx} denoting a connect request, \texttt{disc} denoting a disconnect request, and \texttt{ack} denoting an acknowledgement to a connect request.

State variables of protocol entity \(i\):

\[
\begin{align*}
\text{State}_i & : \langle \text{as defined in upper interface } U \rangle. \\
\text{Sent}_i, \text{Received}_i & : \langle \text{as defined in lower interface } L \rangle. \\
S_i & : \{\text{null}, \text{connS}, \text{connR}, \text{ackS}, \text{ackR}, \text{discS}, \text{discR}, \text{discS} \& \text{connS}, \text{discR} \& \text{connR}\}. \text{ Initially null.} \\
S'_i & : \text{null} \text{ indicates that protocol entity } i \text{ does not have any obligation. } S'_i=\text{connS} \text{ indicates that protocol entity } i \text{ has received a conn message for which it must execute an appropriate output event. The values } S'_i=\text{discS} \text{ and } S'_i=\text{discR} \text{ (and } S'_i=\text{ackS} \text{ and } S'_i=\text{ackR}) \text{ indicate similar conditions for } \text{disc} \text{ (and } \text{ack}) \text{ messages. The value } S'_i=\text{discS} \& \text{connS} \text{ indicates that protocol entity } i \text{ must send a } \text{disc} \text{ message followed by a } \text{conn} \text{ message; this can happen if protocol entity } i \text{ was in the } \text{ActiveOpen} \text{ state, and the local user entity issued a disconnect request followed by a connect request before protocol entity } i \text{ could handle the disconnect request. The value } S'_i=\text{discR} \& \text{connR} \text{ indicates that protocol entity } i \text{ has received a } \text{disc} \text{ message followed by a } \text{conn} \text{ message, for which it must execute appropriate output events.}
\end{align*}
\]

Events of protocol entity \(i\):

We first specify module events that match events of \(U\), and then specify module events that match events of \(L\). For an interface event \(e_i\), the formula of the matching module event \(e_j\) has the form \(f \rightarrow g\) where \(f\) is the interface event formula and \(g\) is a formula that has no appearance of any primed interface variable (i.e., no change to any interface variable is specified by \(g\)). The parameter \(i\) ranges over 1 and 2. Events of protocol entity \(i\) do not access state variables of protocol entity \(j\), where \(i \neq j\).

\[
\begin{align*}
\text{ConnReq}_i & : \text{formula}(\text{ConnReq}) \land \langle (S_i=\text{discS} \land S'_i=\text{discS} \& \text{connS}) \land (S_i=\text{discS} \land S'_i=\text{connS}) \rangle \\
\text{ConnResp}_i & : \text{formula}(\text{ConnResp}) \land S'_i=\text{ackS} \\
\text{DiscReq}_i & : \text{formula}(\text{DiscReq}) \land S'_i=\text{discS} \\
\text{ConnInd}_i & : \text{formula}(\text{ConnInd}) \land S'_i=\text{connR} \land S'_i=\text{null} \\
\text{Collison}_i & : \text{formula}(\text{Collison}) \land S'_i=\text{connR} \land S'_i=\text{null} \\
\text{ConnConf}_i & : \text{formula}(\text{ConnConf}) \land S'_i=\text{ackR} \land S'_i=\text{null} \\
\text{DiscInd}_i & : \text{formula}(\text{DiscInd}) \land (S'_i=\text{discR} \land S'_i=\text{null}) \land (S_i=\text{discS} \land S'_i=\text{connR}) \\
\text{Send}(\text{conn}) & : \text{formula}(\text{Send}(\text{conn})) \land S'_i=\text{connS} \land S'_i=\text{null} \\
\text{Send}(\text{ack}) & : \text{formula}(\text{Send}(\text{ack})) \land S'_i=\text{ackS} \land S'_i=\text{null} \\
\text{Send}(\text{disc}) & : \text{formula}(\text{Send}(\text{disc})) \land (S'_i=\text{discS} \land S'_i=\text{null}) \land (S_i=\text{discS} \land S'_i=\text{connS}) \\
\text{Rec}(\text{conn}) & : \text{formula}(\text{Rec}(\text{conn})) \land (\langle \text{State}=\text{Closed} \land S'_i=\text{commR} \rangle \land (S_i=\text{discR} \land S'_i=\text{discS} \& \text{connS}) \land \langle \text{State}=\text{Closed} \land S'_i=\text{discR} \land S'_i=\text{S}_i \rangle)
\end{align*}
\]
input event is enabled in every state of the
our theory is required to be input-enabled
safety requirement of the module’s
module has a choice: it may disable
I/O automaton can execute independently
interface event sequences generate proper-
properties. For example, it cannot be
overflow are blocked. (Blocking is used
control, flow control or congestion.

The model of Abadi and Lamperti-
different from our model and those of [5]
interaction is represented by a change in
occurrence of an interface event involves

A restriction in our model that is
accepted this restriction because we use
complex system rather than the kind of com-

To specify nontrivial examples that
work with state formulas and event-
tant and progress assertions than safe-
allowed sequences of interface events
a set of invariant and progress require-
ments from the allowed behaviors of the state
facilitates our proof that a module of
module states to interface states to pro-
refinement relation. By using auxiliary
possibilities mappings [14].

Conceptually, the use of a state-
implmentor, because only the set of
strained by the assertions, is of inter-
tors of modules that offer the interface.

Acknowledgements: The development
of the University of Maryland, College
Mohamed Gouda and Thomas Woo of
lar, Ken Calvert suggested the condi-

References

Research Center, Palo Alto, CA 9
W. de Bakker, W.-P. de Roever a
Massachusetts, 1988.
input event is enabled in every state of the automaton. In this respect, our model is more general; a module in our theory is required to be input-enabled only when the occurrence of an input event would not violate any safety requirement of the module's interface(s). For an input event whose occurrence would be unsafe, the module has a choice: it may disable the input or let it occur. Because of the input-enabled requirement, each (O) automaton can execute independently because its outputs cannot be blocked by other automata, but the set of interface event sequences generated by the automaton is inadequate for encoding various desirable interface properties. For example, it cannot be used to specify a module with a finite buffer such that inputs causing overflow are blocked. (Blocking is useful in the specification of many communication protocols that enforce input control, flow control or congestion control.)

The model of Abadi and Lamport [2] is state-based, without interface events. It is fundamentally different from our model and those of [5,14] in how a module and its environment interact. Specifically, such an interaction is represented by a change in the observable portion of the module's state, rather than by the occurrence of an interface event involving the simultaneous participation of the module and environment.

A restriction in our model that is uniquely ours is that modules can only be composed hierarchically. We accepted this restriction because we were motivated by our interest in decomposing the specification of a complex system rather than the kind of composition problems of interest in the area of distributed algorithms.

To specify nontrivial examples, we prefer to use the relational notation [9]. We find it more convenient to work with state formulas and event formulas than individual states and transitions, and to reason with invariant and progress assertions than safe and allowed event sequences. In relational specifications, the set of allowed sequences of interface events is not represented directly. Instead, a labeled state transition system and a set of invariant and progress requirements are specified, and the set of allowed event sequences is obtained from the allowed behaviors of the state transition system. Having states represented explicitly in behaviors facilitates our proof that a module offers an interface. Specifically, we make use of a projection mapping from module states to interface states to prove that the state transition systems of the module and interface satisfy a refinement relation. By using auxiliary variables, such projection mappings [9] are as general as multi-valued possibilities mappings [14].

Conceptually, the use of a state transition system in an interface specification should not influence an implementor, because only the set of allowed event sequences, generated by the state transition system and constrained by the assertions, is of interest. In practice, however, the state transition system might bias implementation of modules that offer the interface.

Acknowledgements: The development of our theory has benefited from many discussions with Sandra Murphy of the University of Maryland, College Park. We also thank Leslie Lamport of DEC, and Ken Calvert, Mohamed Gouda and Thomas Woo of the University of Texas at Austin for their helpful comments. In particular, Ken Calvert suggested the condition of deterministic events for relationally-specified interfaces.

References

Stepwise Refinement

Abstract

A processor farm is a distributed system, together with a number of identically configured computers, connected by an arbitrary communication network. An environment is given to the master computer, which turns some answer to the master: the same environment is given to the slaves, and the slaves take care of the tasks between the master and the slaves.

The processor farm paradigm, in which a distributed system is divided into a master and several slaves, has been used to construct distributed systems using parallel computers. The derivation of a distributed system from a master and slave is called a processor farm. The derivation is carried out by a processor farm algorithm.

The action system formalism, introduced by Back and Kurki-Suonio in [9,12], is defined by a distributed system, in which the actions are divided into a master and several slaves. The actions are performed by the slaves, and the master takes care of the tasks between the master and the slaves. The derivation of a distributed system from a master and slave is called an action system. The derivation is carried out by an action system algorithm.

1 Introduction

A processor farm [9,12] is a distributed system, together with an arbitrary communication network. An environment is given to the master computer, which turns some answer to the master: the same environment is given to the slaves, and the slaves take care of the tasks between the master and the slaves.

The processor farm paradigm, in which a distributed system is divided into a master and several slaves, has been used to construct distributed systems using parallel computers. The derivation of a distributed system from a master and slave is called a processor farm. The derivation is carried out by a processor farm algorithm.

The action system formalism, introduced by Back and Kurki-Suonio in [9,12], is defined by a distributed system, in which the actions are divided into a master and several slaves. The actions are performed by the slaves, and the master takes care of the tasks between the master and the slaves. The derivation of a distributed system from a master and slave is called an action system. The derivation is carried out by an action system algorithm.