Protocol Specification, Testing, and Verification, XI
B. Jonsson, J. Parrow and B. Pehrson (Editors)
Elsevier Science Publishers B.V. (North-Holland)

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-9004464. The work of A. Udaya Shankar was supported by National Science Foundation grants no. ECS-8502113 and no. NCR-890450.

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 $\{S_i\}$ from I, one for each module in the collection (*decomposition* step).

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

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

The above approach has the following highly-desirable feature: given interfaces $\{S_i\}$, each module can be designed and implemented individually. However, the decomposition step—i.e., deriving the interfaces $\{S_i\}$ from I—is not an easy task. Furthermore, to develop the approach into a valid method, the following problem has to be solved, namely: In general, the inference rule 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, Pnueli [18] presented a proof rule and Abadi and Lamport [2] presented a composition principle, that are more general than the rule 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 service provider on one 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.

user

____________________ upper interface U

module M

____________________ lower interface L

module N

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 protocol layer is composed of a set of entities [17,19,20]. We place no restriction on how these entities are composed.)

Our composition theorem provides a theoretical foundation for layered systems. With the composition theorem, we are assured that each layer in the system, say M with upper interface U and lower interface L, can be designed, implemented and modified individually. As long as the interfaces remain the same and *M using L offers U* is satisfied, the internals of M can change.

The balance this paper is organized as follows. In Section 2, we present our theory in a general semantic 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 some notation for sequences. A *sequence over E*, where E is a set, means a (finite or infinite) sequence $(e_0, e_1, \cdots)$, where $e_i \in E$ for all i. A *sequence over alternating E and F*, where E and F are sets, means a sequence $(e_0, f_0, e_1, f_1, \cdots)$, where $e_i \in E$ and $f_i \in F$ for all i.

Definition. An interface I is defined by:

- *Events* (I), a set of events that is the union of two disjoint sets,

  Inputs (I), a set of input events, and

  Outputs (I), a set of output events.

- *AllowedEventSeqs* (I), a set of sequences over *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

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

which includes the empty sequence.

Definition. A state transition system A is defined by:

- *States* (A), a set of states.
- *Initial* (A), a subset of *States* (A), referred to as initial states.
- *Events* (A), a set of events.
- $Transitions_A(e)$, a subset of $States(A) \times States(A)$, for every $e \in Events(A)$. Each element of $Transitions_A(e)$ is an ordered pair of states (s, t) referred to as a transition of e.

A *behavior* of A is a sequence $\sigma = (s_0, e_0, s_1, e_1, \cdots)$ over alternating *States* (A) and *Events* (A) such that $s_0 \in Initial(A)$ and (s_i, s_{i+1}) is a transition of e_i for all i. A finite sequence σ over alternating *States* (A) and *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 *Behaviors*(A). The set of finite behaviors of A is denoted by *FiniteBehaviors*(A).

For $e \in Events(A)$, let $enabled_A(e) \equiv \{s\text{: for some state } t, (s,t) \in Transitions_A(e)\}$. An event e is said to be enabled in a state s of A iff $s \in enabled_A(e)$. An event e is said to be disabled in a state s of A iff $s \notin enabled_A(e)$.

Notation. For any sequence σ over alternating $States(A)$ and $Events(A)$, and for any set $E \subseteq Events(A)$, $image(\sigma, E)$ denotes the sequence of events in E obtained from σ 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:

  Inputs(M), a set of input events,

  Outputs(M), a set of output events, and

  Internals(M), a set of internal events.

- *sts*(M), a state transition system with $Events(sts(M))=Events(M)$.
- *Fairness requirements of* M, a finite collection of subsets of $Outputs(M) \cup Internals(M)$. Each subset is referred to as a fairness requirement of M.

Convention. For readability, the notation $sts(M)$ is abbreviated to M wherever such abbreviation causes no ambiguity, e.g., $States(sts(M))$ is abbreviated to $States(M)$, $enabled_{sts(M)}(e)$ is abbreviated to $enabled_M(e)$, 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 $\sigma=(s_0, e_0, s_1, e_1, \cdots, s_j, e_j, \cdots)$, we say that F occurs in state s_j iff $e_j \in F$. An infinite behavior σ of M satisfies F iff F occurs infinitely often or is disabled infinitely often in states of σ.

For module M, a behavior σ is an *allowed behavior* iff for every fairness requirement F of M: σ is finite and F is not enabled in its last state, or σ is infinite and satisfies F. Let *AllowedBehaviors*(M) denote the set of 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 σ be a sequence over alternating states and events of module M.

Definition. σ is allowed wrt I iff $image(\sigma, Events(I)) \in AllowedEventSeqs(I)$.

Definition. σ is safe wrt I iff one of the following holds:

- σ is finite and $image(\sigma, Events(I)) \in SafeEventSeqs(I)$.
- σ is infinite and every finite prefix of σ is safe wrt I.

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

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

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

$$\forall e \in Outputs(M)\colon\ last(\sigma) \in enabled_M(e) \ \Rightarrow\ \sigma@e \text{ is safe wrt } I, \text{ and}$$

$$\forall e \in Inputs(M)\colon\ \sigma@e \text{ is safe wrt } I \ \Rightarrow\ last(\sigma) \in enabled_M(e).$$

- Progress constraints:
  For all $\sigma \in AllowedBehaviors(M)$, if σ is safe wrt I, then σ is allowed wrt I.

A module M with upper interface U and lower interface L is illustrated in Figure 1. The environment of M consists of the user of U and the module that offers L. The meaning of *M using L offers U* is next defined. In what follows, we use "σ is safe wrt U and L" to mean "σ is safe wrt U and σ is safe wrt L."

Definition. Given module M and interfaces U and L, *M using L offers U* iff the following conditions hold:

- Naming constraints:
  $Events(U) \cap Events(L) = \varnothing$,
  $Inputs(M) = Inputs(U) \cup Outputs(L)$, and
  $Outputs(M) = Outputs(U) \cup Inputs(L)$.
- Safety constraints:
  For all $\sigma \in FiniteBehaviors(M)$, if σ is safe wrt U and L, then

  $$\forall e \in Outputs(M)\colon\ last(\sigma) \in enabled_M(e) \ \Rightarrow\ \sigma@e \text{ is safe wrt } U \text{ and } L, \text{ and}$$

  $$\forall e \in Inputs(M)\colon\ \sigma@e \text{ is safe wrt } U \text{ and } L \ \Rightarrow\ last(\sigma) \in enabled_M(e).$$
- Progress constraints:
  For all $\sigma \in AllowedBehaviors(M)$, if σ is safe wrt U and L, then

  $$\sigma \text{ is allowed wrt } L \ \Rightarrow\ \sigma \text{ is allowed wrt } U.$$

The definition of *M using L offers U* is similar to the definition of *M offers I* in most respects. We first review the key elements that are common to both definitions:

- Module M is required to satisfy its interface(s) only if the environment of M satisfies the safety requirements of its interface(s). Specifically, for any finite behavior that is not safe wrt M's interface(s), the two Safety constraints are satisfied trivially; for any allowed behavior that is not safe wrt M's interface(s), the Progress constraint is satisfied trivially. That is, as soon as the environment of M violates some safety requirement of M's interface(s), module M can behave arbitrarily and still satisfy *M offers I* or *M using L offers U*.
- Module M satisfies the safety requirements of its interface(s). Specifically, whenever an output event of M is enabled to occur, the event's occurrence would be safe.
- Whenever an input event of M (controlled by its environment) can occur safely, M does not block the event's occurrence.

For an input event whose occurrence would be unsafe, module M has a choice: it may block the event's occurrence or let it occur. (In this respect, our model is more general than the I/O automata model [14], which requires an I/O automaton to be always input-enabled.)

The main difference between the definitions of *M offers I* and *M using L offers U* is in the Progress constraints. For module M using interface L, it is required to satisfy the progress requirements of interface U *only if* the module that offers L satisfies the progress requirements of L.

Definition. A finite set of modules $\{M_j : j \in J\}$ are compatible iff $\forall j, k \in J, j \neq k$:

$Internals(M_j) \cap Events(M_k) = \varnothing$, and $Outputs(M_j) \cap Outputs(M_k) = \varnothing$.

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 $t_j \in States(M_j)$. We use $image(s, M_j)$ 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:

- $Events(M)$ defined by:

  $Internals(M) = [\bigcup_{j \in J} Internals(M_j)] \cup [(\bigcup_{j \in J} Outputs(M_j)) \cap (\bigcup_{j \in J} Inputs(M_j))]$

  $Outputs(M) = [\bigcup_{j \in J} Outputs(M_j)] - [\bigcup_{j \in J} Inputs(M_j)]$

  $Inputs(M) = [\bigcup_{j \in J} Inputs(M_j)] - [\bigcup_{j \in J} Outputs(M_j)]$

- $sts(M)$ defined by:

  $States(M) = \prod_{j \in J} States(M_j)$

  $Initial(M) = \prod_{j \in J} Initial(M_j)$

  $Transitions_M(e)$, for all $e \in Events(M)$, defined by: $(s, t) \in Transitions_M(e)$ iff, $\forall j \in J$,

  if $e \in Events(M_j)$ then $(image(s, M_j), image(t, M_j)) \in Transitions_{M_j}(e)$, and

  if $e \notin Events(M_j)$ then $image(s, M_j) = image(t, M_j)$.

- $Fairness\ requirements\ of\ M = [\bigcup_{j \in J} Fairness\ requirements\ of\ M_j]$.

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

- $Internals(M) \cap Internals(N) = \varnothing$
- 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 limitation. Theorem 1 is at the heart of our approach to compose modules hierarchically.

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

- For all j, k, if $j \neq k$ then $Events(I_j) \cap Events(I_k) = \varnothing$ and $Internals(M_j) \cap Internals(M_k) = \varnothing$.
- M_1 offers I_1.
- For $j = 2, \cdots, n$, M_j using I_{j-1} offers I_j.

Then, modules $M_1, \cdots, M_n$ 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 is specified by a set of variables, called state variables. For a state transition system A, the set of variables is denoted by $Variables(A)$. For each variable v, there is a set $domain(v)$ of allowed values. By definition, $States(A) = \prod_{v \in Variables(A)} domain(v)$. Each state $s \in States(A)$ is represented by a tuple of values, $(d_v\colon v \in Variables(A))$, where $d_v \in domain(v)$.

We use state formulas to represent subsets of $States(A)$. A *state formula* is a formula in $Variables(A)$ that evaluates to true or false when $Variables(A)$ is assigned s, for every state $s \in States(A)$. 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.[1]

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

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

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

Note that for each event e, we have

$$enabled_A(e) = [\exists Variables(A)'\colon formula_A(e)]$$

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

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: *invariant* P, where P is a state formula. A finite sequence over alternating states and events satisfies *invariant* P iff every state in the sequence satisfies P. An infinite sequence over alternating states and events satisfies *invariant* P iff every finite prefix of the sequence satisfies *invariant* P.

We use leads-to assertions of the form: P *leads−to* Q, where P and Q are state formulas.[2] A sequence $(s_0, e_0, s_1, e_1, \cdots)$ 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 σ denote a sequence over alternating states and events. An assertion is true for σ iff σ 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 *Assert*, we first evaluate for σ the truth value of every atomic assertion within *Assert*. For example, σ satisfies the assertion $X \wedge Y \Rightarrow Z$, where X, Y and Z are atomic assertions, iff (σ satisfies X) $\wedge$ (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 *progress assertion* is an assertion constructed from atomic assertions that include at least one leads-to assertion. A module satisfies a progess 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

- $Internals(A) = \varnothing$,
- $Initial(A)$ is a single state, and
- for all $e \in Events(A)$, $Transitions_A(e)$ is a partial function, i.e., for all $s \in States(A)$, there is at most one state s' such that $(s, s') \in Transitions_A(e)$.

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 $States(A)$, and $Init(A)$ the desired initial states of A. Define $Initial(A)$ to be $\{s_0\}$ and, for all $s \in Init(A)$, specify a distinct event for each transition (s_0, s).

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'.

Definition. An interface I is specified in the relational notation by:

- Disjoint sets of events, $Inputs(I)$ and $Outputs(I)$, with $Events(I)$ being their union.
- $sts(I)$, a state transition system with deterministic events specified in the relational notation such that $Events(sts(I)) = Events(I)$.
- $InvAssum_I$, a conjunction of state formulas referred to as *invariant assumptions* of I, such that

  $Initial(I) \Rightarrow InvAssum_I$, and

  $\forall e \in Outputs(I)$: $InvAssum_I \wedge formula_I(e) \Rightarrow InvAssum_I'$
- $InvGuar_I$, a conjunction of state formulas referred to as *invariant guarantees* of I, such that

  $Initial(I) \Rightarrow InvGuar_I$, and

  $\forall e \in Inputs(I)$: $InvGuar_I \wedge formula_I(e) \Rightarrow InvGuar_I'$
- $ProgReqs_I$, a conjunction of progress assertions, referred to as *progress requirements* of I.

The invariant assumptions and guarantees of interface I are collectively referred to as *invariant requirements* of interface I. Define

$$InvReqs_I \equiv InvAssum_I \wedge InvGuar_I.$$

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

$AllowedBehaviors(I) = \{\sigma: \sigma \in Behaviors(I)$ and σ satisfies *invariant* $InvReqs_I$ and $ProgReqs_I\}$,

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

Lastly, for event $e \in Events(I)$, define

$$possible_I(e) \equiv InvReqs_I \wedge [\exists Variables(I)': formula_I(e) \wedge InvReqs_I']$$

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 $States(A) \times States(A)$ 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 iff

- $Variables(A) \supseteq Variables(B)$ and $Events(A) \supseteq Events(B)$
- $Initial_A \Rightarrow Initial_B$
- $\forall e \in Events(B)$: $Inv_A \wedge formula_A(e) \Rightarrow formula_B(e)$

  (event refinement condition)
- $\forall e \in Events(A) - Events(B)$: $Inv_A \wedge formula_A(e) \Rightarrow [\forall v \in Variables(B): v = v']$

  (null image condition)

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 $sts(M)$ is a refinement of $sts(I)$ assuming Inv_M

B3 $\forall e \in Inputs(I)$: $Inv_M \wedge possible_I(e) \Rightarrow enabled_M(e)$

B4 $\forall e \in Outputs(I)$: $Inv_M \wedge formula_M(e) \Rightarrow InvGuar_I'$

B5 $sts(M)$ satisfies (*invariant* $InvAssum_I$ $\Rightarrow$ *invariant* Inv_M)

B6 M satisfies (*invariant* $InvAssum_I$ $\Rightarrow$ $ProgReqs_I$)

Theorem 3. For a module M, an interface I, and some state formula Inv_M in *Variables* (M), if conditions **B1-B6** hold, then

(a) M offers I, and
(b) $\forall \sigma \in$ *Behaviors* (M): σ satisfies *invariant* $InvAssum_I \Rightarrow \sigma$ 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 $sts(M)$ are named such that **B1** is satisfied. Second, events of $sts(M)$ are specified such that $sts(M)$ is a refinement of $sts(I)$ (**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, Inv_M is equal to $InvAssum_I$. But to prove **B2-B4**, we may have to assume that $sts(M)$ has additional invariant properties, which are used to strengthen Inv_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 Inv_M in *Variables* (M), the following conditions, expressed in the relational notation, are sufficient for *M using L offers U*:

C1 $Events(U) \cap Events(L) = \varnothing$
$Inputs(M) = Inputs(U) \cup Outputs(L)$
$Outputs(M) = Outputs(U) \cup Inputs(L)$
$Variables(U) \cap Variables(L) = \varnothing$

C2 $sts(M)$ is a refinement of $sts(U)$ assuming Inv_M

C3 $sts(M)$ is a refinement of $sts(L)$ assuming Inv_M

C4 $\forall e \in Inputs(U)$: $Inv_M \wedge possible_U(e) \Rightarrow enabled_M(e)$

C5 $\forall e \in Outputs(L)$: $Inv_M \wedge possible_L(e) \Rightarrow enabled_M(e)$

C6 $\forall e \in Inputs(L)$: $Inv_M \wedge formula_M(e) \Rightarrow InvAssum_L'$

C7 $\forall e \in Outputs(U)$: $Inv_M \wedge formula_M(e) \Rightarrow InvGuar_U'$

C8 $sts(M)$ satisfies (*invariant* $(InvAssum_U \wedge InvGuar_L) \Rightarrow$ *invariant* Inv_M)

C9 M satisfies (*invariant* $(InvAssum_U \wedge InvGuar_L) \wedge ProgReqs_L \Rightarrow ProgReqs_U$)

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

(a) M using L offers U, and
(b) $\forall \sigma \in$ *Behaviors* (M): σ satisfies *invariant* $(InvAssum_U \wedge InvGuar_L) \Rightarrow \sigma$ is safe wrt U and L.

C8 indicates that we can set Inv_M equal to $InvAssum_U \wedge InvGuar_L$ initially. However, to prove **C2-C7** for a module M, we may have to assume that $sts(M)$ has additional invariant properties, which are used to strengthen Inv_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:

$$e_1 \equiv v_1 > 2 \wedge v_2' \in \{1,2,5\}$$

$$e_2 \equiv v_1 > v_2 \wedge v_1 + v_2' = 5$$

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

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

$$e_3(m) \equiv v_1 > v_2 \wedge v_1 + v_2' = m$$

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.

Lastly, 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:

$State_i$: $\{Closed, PassiveOpening, ActiveOpening, PassiveOpen, ActiveOpen\}$. Initially $Closed$.

Input events:

$$ConnReq_i \equiv State_i = Closed \wedge State_i' = ActiveOpening$$

$$ConnResp_i \equiv State_i = PassiveOpening \wedge State_i' = PassiveOpen$$

$$DiscReq_i \equiv State_i = ActiveOpen \wedge State_i' = Closed$$

Output events:

$$ConnInd_i \equiv State_i = Closed \wedge State_i' = PassiveOpening$$

$$Collision_2 \equiv State_2 = ActiveOpening \wedge State_2' = PassiveOpening$$

$$ConnConf_i \equiv State_i{=}ActiveOpening \wedge State_i'{=}ActiveOpen$$

$$DiscInd_i \equiv State_i{=}PassiveOpen \wedge State_i'{=}Closed$$

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

Invariant and progress requirements:

$$InvAssum_U \equiv true$$

$InvGuar_U \equiv InvGuar_{U,1} \wedge InvGuar_{U,2}$, where

$$InvGuar_{U,i} \equiv (State_i{=}ActiveOpen \Rightarrow State_j{=}PassiveOpen) \wedge (State_i{=}PassiveOpening \Rightarrow State_j{=}ActiveOpening)$$

The first conjunct of $InvGuar_{U,i}$ can be falsified only by the event $ConnConf_i$ (which makes the antecedent true) and the event $DiscInd_j$ (which makes the consequent false). The second conjunct of $InvGuar_{U,i}$ can be falsified only by the event $ConnInd_i$ (which makes the antecedent true), the event $ConnConf_j$ (which makes the consequent false), and the event $Collision_2$ (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 $InvGuar_U$ as required by our definition of a relationally-specified interface.

$$\begin{aligned} ProgReqs_U \equiv\ & ((State_2{=}PassiveOpening \text{ leads-to } State_2{=}PassiveOpen) \\ & \quad \Rightarrow (State_1{=}ActiveOpening \text{ leads-to } State_1{=}ActiveOpen)) \\ & \wedge ((State_1{=}PassiveOpening \text{ leads-to } State_1{=}PassiveOpen) \\ & \quad \wedge (State_2{=}PassiveOpening \text{ leads-to } State_2{=}PassiveOpen) \\ & \quad \Rightarrow (State_2{=}ActiveOpening \text{ leads-to } State_2 \in \{ActiveOpen, PassiveOpen\})) \end{aligned}$$

4.2. Interface L specifying reliable message delivery

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

State variables:

$Sent_i$: sequence of messages. Initially the null sequence.
$Received_i$: sequence of messages. Initially the null sequence.

$Sent_i$ is the sequence of messages that have been sent at access point i since the beginning of execution. $Received_i$ 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:

$$Send_i(m) \equiv Sent_i'{=}Sent_i @ (m)$$

Output events:

$$Rec_i(m) \equiv Received_i'{=}Received_i @ (m)$$

Invariant and progress requirements:

$$InvAssum_L \equiv true$$

$$InvGuar_L \equiv (Received_2 \text{ prefix-of } Sent_1) \wedge (Received_1 \text{ prefix-of } Sent_2)$$

$$ProgReqs_L \equiv (|Sent_1| \geq k \text{ leads-to } |Received_2| \geq k) \wedge (|Sent_2| \geq l \text{ leads-to } |Received_1| \geq l)$$

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, *conn* denoting a connect request, *disc* denoting a disconnect request, and *ack* denoting an acknowledgement to a connect request.

State variables of protocol entity i:

$State_i$: <as defined in upper interface U>.

$Sent_i$, $Received_i$: <as defined in lower interface L>.

S_i: $\{null, connS, connR, ackS, ackR, discS, discR, disc\&connS, disc\&connR\}$. Initially $null$.

$S_i{=}null$ indicates that protocol entity i does not have any obligation. $S_i{=}connS$ indicates that protocol entity i must send a *conn* message. $S_i{=}connR$ indicates that protocol entity i has received a *conn* message for which it must execute an appropriate output event. The values *discS* and *discR* (and *ackS* and *ackR*) indicate similar conditions for *disc* (and *ack*) messages. The value *disc&connS* indicates that protocol entity i must send a *disc* message followed by a *conn* message; this can happen if protocol entity i was in the *ActiveOpen* state, and the local user entity issued a disconnect request followed by a connect request before protocol entity i could handle the disconnect request. The value *disc&connR* indicates that protocol entity i has received a *disc* message followed by a *conn* message, for which it must execute appropriate output events.

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_i has the form $f \wedge g$ 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 $i \neq j$.

$$ConnReq_1 \equiv formula_U(ConnReq_1) \wedge ((S_1{=}discS \wedge S_1'{=}disc\&connS) \vee (S_1{\neq}discS \wedge S_1'{=}connS))$$

$$ConnReq_2 \equiv formula_U(ConnReq_2) \wedge ((S_2{=}discS \wedge S_2'{=}disc\&connS) \vee (S_2{=}connR \wedge S_2'{=}connR) \vee (S_2 \notin \{discS, connR\} \wedge S_2'{=}connS))$$

$$ConnResp_i \equiv formula_U(ConnResp_i) \wedge S_i'{=}ackS$$

$$DiscReq_i \equiv formula_U(DiscReq_i) \wedge S_i'{=}discS$$

$$ConnInd_i \equiv formula_U(ConnInd_i) \wedge S_i{=}connR \wedge S_i'{=}null$$

$$Collision_2 \equiv formula_U(Collision_2) \wedge S_2{=}connR \wedge S_2'{=}null$$

$$ConnConf_i \equiv formula_U(ConnConf_i) \wedge S_i{=}ackR \wedge S_i'{=}null$$

$$DiscInd_i \equiv formula_U(DiscInd_i) \wedge ((S_i{=}discR \wedge S_i'{=}null) \vee (S_i{=}disc\&connR \wedge S_i'{=}connR))$$

$$Send_i(conn) \equiv formula_L(Send_i(conn)) \wedge S_i{=}connS \wedge S_i'{=}null$$

$$Send_i(ack) \equiv formula_L(Send_i(ack)) \wedge S_i{=}ackS \wedge S_i'{=}null$$

$$Send_i(disc) \equiv formula_L(Send_i(disc)) \wedge ((S_i{=}discS \wedge S_i'{=}null) \vee (S_i{=}disc\&connS \wedge S_i'{=}connS))$$

$$Rec_1(conn) \equiv formula_L(Rec_1(conn)) \wedge ((State_1{=}Closed \wedge S_1'{=}connR) \vee (S_1{=}discR \wedge S_1'{=}disc\&connR) \vee (State_1{\neq}Closed \wedge S_1{\neq}discR \wedge S_1'{=}S_1))$$

$Rec_2(conn) \equiv formula_L(Rec_2(conn)) \wedge ((S_2{=}discR \wedge S_2'{=}disc\&connR) \vee (S_2{\neq}discR \wedge S_2'{=}connR))$

$Rec_i(ack) \equiv formula_L(Rec_i(ack)) \wedge S_i'{=}ackR$

$Rec_i(disc) \equiv formula_L(Rec_i(disc)) \wedge S_i'{=}discR$

Fairness requirements: For each protocol entity i of module M, there is a fairness requirement consisting of the output events of the protocol entity.

4.4. Satisfaction of conditions C1-C9

It is obvious that condition **C1** is satisfied. **C2** and **C3** are satisfied (for $Inv_M{=}true$) because each module event has the special form $f \wedge g$ described above. **C4** and **C5** are satisfied (for $Inv_M{=}true$) because every input event of module M has the special form $f \wedge g$ and $enabled(g){=}true$. **C6** holds since $InvAssum_L \equiv true$. **C7** holds if we define $Inv_M \equiv Inv_{M,1} \wedge Inv_{M,2}$, where

$$
\begin{aligned}
&Inv_{M,i} \equiv \\
&\quad (State_i{=}ActiveOpening \wedge S_i{=}ackR \Rightarrow State_j{=}PassiveOpen) \\
&\quad \wedge (State_i{=}ActiveOpen \Rightarrow State_j{=}PassiveOpen \wedge S_j \notin \{discR, disc\&connR\}) \\
&\quad \wedge (State_i{=}Closed \wedge S_i{=}connR \Rightarrow State_j{=}ActiveOpening) \\
&\quad \wedge (State_i{=}PassiveOpening \Rightarrow State_j{=}ActiveOpening \wedge S_j \notin \{connR, ackR\}) \\
&\quad \wedge (State_2{=}ActiveOpening \wedge S_2{=}connR \Rightarrow State_1{=}ActiveOpening)
\end{aligned}
$$

The first two conjuncts of $Inv_{M,i}$ are sufficient (and necessary) for the output events of M to preserve the first conjunct of $InvGuar_{U,i}$. The last three conjuncts of $Inv_{M,i}$ are sufficient (and necessary) for the output events of M to preserve the second conjunct of $InvGuar_{U,i}$.

It remains to be proved that **C8** and **C9** hold. (The proof is omitted due to space limitation.)

5. Concluding Remarks

The concept of layering was described by Dijkstra more than two decades ago [4]. Layering has been applied to the design and implementation of computer network protocols, and also operating systems (in particular security kernels). However, to reap the benefits of a layered architecture—i.e., to be able to design, implement, and modify each layer individually—we need formal definitions of the meanings of *interface*, *M offers I*, and *M using L offers U*, as well as a composition theorem such as the one presented in this paper.

In designing our model and theory, we were faced with two conflicting goals. On the one hand, we would like to have a model that is as general as possible so that our theory has wide applicability. On the other hand, to prove the composition theorem, the model needs to be restricted in various ways. Below, we compare our model and theory to those in [2,5,14].

Our model and the CSP model [5] are different in many ways. We mention two here. First, the semantics of a process in the CSP model is given by a set of finite traces and associated refusal sets, whereas we specify a module using a set of behaviors and a set of fairness requirements (each behavior is represented by a sequence of alternating states and events). Specifically, the concepts of internal state and fairness are essential in our theory but are absent in the theory of CSP. Second, the notion of *M satisfies S*, where S is a specification, in the theory of CSP is not the same as our notion of *M offers I*, where I is a two-sided interface between a service provider and a service consumer; in particular, there is no requirement in the CSP model that interface events are unilaterally controlled.

In the theory of I/O automata [14,15], there is no distinction between module and interface, service provider and service consumer. There is the notion of one automaton simulating another automaton, but not our notion of a two-sided interface. Furthermore, each I/O automaton is required to be input-enabled, i.e., every 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 I/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 implementors 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

[1] M. Abadi and L. Lamport, *The existence of refinement mappings*, Research Report 29, Digital Systems Research Center, Palo Alto, CA 94301, August 1988.

[2] M. Abadi and L. Lamport, "Composing specifications," in *Stepwise Refinement of Distributed Systems*, J. W. de Bakker, W.-P. de Roever and G. Rozenberg (Eds.), LNCS 430, Springer-Verlag, 1990.

[3] K. M. Chandy and J. Misra, *A Foundation of Parallel Program Design*, Addison-Wesley, Reading, Massachusetts, 1988.

[4] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, Vol. 1, 1971.

[5] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[6] B. Jonsson, "On Decomposing and Refining Specifications of Distributed Systems," in *Stepwise Refinement of Distributed Systems*, J. W. de Bakker, W.-P. de Roever and G. Rozenberg (Eds.), LNCS 430, Springer-Verlag, 1990.

[7] S. S. Lam and A. U. Shankar, "Protocol verification via projections," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 10, July 1984, pp. 325-342.

[8] S. S. Lam and A. U. Shankar, "Specifying Modules to Satisfy Interfaces: A State Transition System Approach," presented at 26th Lake Arrowhead Workshop on *How will we specify concurrent systems in the year 2000?*, September 1987; technical report TR-88-30, Department of Computer Sciences, University of Texas at Austin, January 1991 (revised).

[9] S. S. Lam and A. U. Shankar, "A relational notation for state transition systems," *IEEE Transactions on Software Engineering*, Vol. 16, No. 7, July 1990, pp. 755-775; an abbreviated version entitled "Refinement and projection of relational specifications" in *Stepwise Refinement of Distributed Systems*, J. W. de Bakker, W.-P. de Roever and G. Rozenberg (Eds.), LNCS 430, Springer-Verlag, 1990.

[10] S. S. Lam and A. U. Shankar, "A Theory of Modules and Interfaces," Technical Report, Department of Computer Science, University of Maryland, 1991, in preparation.

[11] S. S. Lam, A. U. Shankar, and T. Y. C. Woo, "Applying a Theory of Modules and Interfaces to Security Verification," *Proceedings IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 1991.

[12] L. Lamport, "What it means for a concurrent program to satisfy a specification: Why no one has specified priority," *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985.

[13] L. Lamport, "A simple approach to specifying concurrent systems," *Comm. ACM*, Vol. 32, No. 1, January 1989.

[14] N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., August 1987.

[15] N. Lynch, M. Merritt, W. Weihl and A. Fekete, "A Theory of Atomic Transactions," Technical Report MIT/LCS/TM-362, Laboratory for Computer Science, M.I.T, June 1988.

[16] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 4, July 1981, pp. 417-426.

[17] S. L. Murphy and A. U. Shankar, "Service specification and protocol construction for the transport layer," CS-TR-2033, UMIACS-TR-88-38, Computer Science Dept., Univ. of Maryland, May 1988; an abbreviated version appears in *Proc. ACM SIGCOMM '88 Symposium*, August 1988.

[18] A. Pnueli, "In transition from global to modular temporal reasoning about programs," NATO ASI Series, Vol. F13, *Logics and Models of Concurrent Systems*, K. R. Apt (ed.), Springer-Verlag, Berlin, Heidelberg, 1984, pp. 123-144.

[19] A. U. Shankar and S. S. Lam, "An HDLC protocol specification and its verification using image protocols," *ACM Transactions on Computer Systems*, Vol. 1, No. 4, November 1983, pp. 331-368.

[20] A. U. Shankar and S. S. Lam, "A stepwise refinement heuristic for protocol construction," Technical report UMIACS-TR-87-12, University of Maryland, College Park, March 1987 (revised March 1989); an abbreviated version entitled "Construction of Network Protocols by Stepwise Refinement" in *Stepwise Refinement of Distributed Systems*, J. W. de Bakker, W.-P. de Roever and G. Rozenberg (Eds.), LNCS