Applying a Theory of Modules and Interfaces to Security Verification

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

Thomas Y.C. Woo [*]
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

We present an overview of a theory of modules and interfaces applicable to the specification and verification of systems with a layered architecture. At the heart of this theory is a module composition theorem. The theory is applied to the specification of a distributed system consisting of *subjects* and *objects* in different hosts (computers). Formal specifications of a user interface and a network interface are given. Access to objects, both local and remote, offered by the distributed system is proved to be multilevel secure.

1 Introduction

Modules and *interfaces* are important concepts. An interface that is carefully specified can be used to hide the internal implementation of a system, i.e., it allows the use of many different implementations that satisfy the same interface specification. In designing a complex system, structured as a hierarchy of layers or as a collection of interacting modules, well-defined interfaces are indispensable.

In this paper, we first present an overview of a theory of modules and interfaces developed recently by Lam and Shankar [11, 12]. In this theory, the meanings of *interface* and *module* are rigorously defined, as well as the meaning of M *offers* I, where M denotes a module and I an interface. Furthermore, for 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. The following composition theorem is proved: If M using L offers U, and N offers L, then M interacting with N offers U. This theorem is at the heart of the theory of Lam and Shankar. They have also proved the following theorem for a linear hierarchy of modules and interfaces, $M_1, I_1, M_2, I_2, \ldots, M_n, I_n$:

> If M_1 offers I_1
> and, for $i = 2, \ldots, n$, M_i using I_{i-1} offers I_i,
> then the hierarchy of modules offers I_n.

The above theorem provides a theoretical foundation for the specification and verification of systems with a layered architecture, where each layer corresponds to a module in the theorem.

An obvious problem in the security area to apply this theorem is the specification and verification of trusted computer systems with a layered architecture [6, 8]. In this paper, we illustrate another application of the theory to the security area. Consider Figure 1 where two modules (layers) are shown, referred to as *system* and *network*. The system module is made up of software in a set of hosts for managing access of subjects to objects. The hosts are interconnected by a communication network. The network module is made up of the hardware and software used to deliver messages from one host to another. The interface between the network and system modules is called the *network interface*. Note that many different network implementations can be used to offer this interface, ranging from the very simple (e.g., secure communication links) to the very complex (e.g., worldwide internetworks). The interface between the system module and subjects is called the *user interface*. Notions of secure information flow between objects and subjects are embodied in the user interface specification.

To verify the security of distributed systems, our approach has several unique characteristics:

[*]The work of Simon S. Lam and Thomas Y.C. Woo was supported in part by a grant from the Texas Advanced Research Program and in part by National Science Foundation grant no. NCR-9004464.
[†]The work of A. Udaya Shankar was supported by National Science Foundation grants no. NCR-890450.

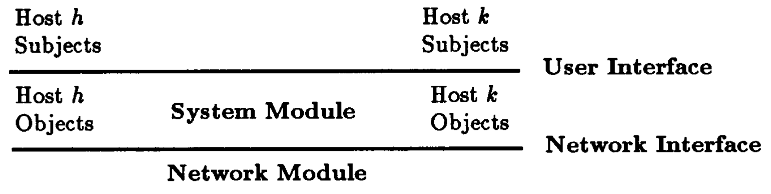

Figure 1: A Distributed System

- The property of secure information flow is a requirement that is enforced at the user interface. In our approach, interface requirements and module implementation requirements are cleanly separated. In most other approaches (e.g., the Bell-LaPadula model), however, separation of these requirements is not emphasized.
- Our approach allows both functional requirements and security requirements to be specified for each interface. Satisfaction of the functional requirements of an interface (e.g., serializability at the user interface, reliable delivery of messages at the network interface) can be achieved and verified separately.
- In our approach, interfaces at different levels of a layered architecture can have different sets of security concerns. In the distributed system of Figure 1, for example, security concerns at the user interface may include access control and secure information flow; security concerns at the network interface may include confidentiality, authenticity, integrity, as well as access control.

In the security literature, network and distributed system security has been studied in [3, 19, 20, 22, 23, 24]. The security of a composition of modules was investigated by McCullough [16, 17, 18]. McCullough's work is on a particular definition of security that is preserved by composition. In our approach, notions of security that can be specified as interface requirements are not restricted (because the notion of safety in the theory of modules and interfaces is general). Furthermore, we allow the specification of different notions of security for different interfaces in the same composition. On the other hand, we do require that modules be organized hierarchically, which is a restriction.

The balance of this paper is organized as follows. In Section 1, we present an overview of the theory of modules and interfaces, including a specification notation designed for application of the theory to nontrivial examples. This section is taken from the work of Lam and Shankar [10, 11, 12]. In Section 3, we present specifications of a user interface U, and a network interface L. Then we present the specification of a system module M, together with a proof that the module satisfies M *using* L *offers* U. The specification of a network module N and derivation of a proof that it satisfies N *offers* L can be carried out in a similar manner, and are omitted for brevity.

2 Overview of Theory

A module interacts with its environment at an interface. Semantically, an interface is specified by a set of allowed sequences of interface events, each of which defines one possible sequence of interactions betweeen the module and its environment. A module is specified by a state transition system and a set of fairness requirements (see Section 2.2).

For a module M and an interface I, the meaning of M *offers* I is defined in [11]. The 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 [2, 5, 7, 9, 10, 13, 14, 15]. Most definitions of M *satisfies* S have this informal meaning: M *satisfies* S if every possible observation of M is described by S. Various definitions differ in whether interface events or states are observable, in whether observations are finite or infinite sequences, as well as in the particular formalism used to represent these sequences. There are also differences in how interactions between a module and its environment can occur.

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 is defined in a manner not unlike the approaches of CSP [7] and I/O automata [15]. The theory of modules and interfaces, however, differs from the theories of CSP and I/O automata in two respects. First, the interaction method at an interface between a module and its environment is different (see Section 2.1). Second, the theory of modules and interfaces was designed with an objective somewhat different from those of CSP and I/O automata. Specifically, it is intended more for system decomposition than for module composition per se. An elaboration on this point follows.

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, for all i, 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, 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 two modules M and N that interact across interface I. Each module guarantees some properties of I only if its environment satisfies some 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.

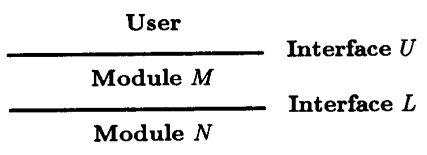

Figure 2: Module M and its Environment

Consider module M in Figure 2. It provides services to a user through interface U while it is using services offered by module N 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 module N, which is the service provider of interface L.

In the Lam-Shankar theory, a basic composition theorem is proved for the configuration in Figure 2. Composition theorems are also proved for an arbitrary number of modules organized in a linear hierarchy and as nodes of a rooted tree [12]. We note that 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—can be specified as a module in a linear hierarchy.

2.1 Interfaces

Consider models in which observations of a module are sequences of interface events [7, 14, 15]. We identify three requirements that characterize interface interactions. First, the occurrence of an interface event involves ***simultaneous participation*** by a module and its environment; moreover, such occurrence is observable by both the module and its environment. This requirement, a hallmark of Hoare's CSP [7], appears to be fundamental and is included in all models that we are familiar with.

The second requirement, which we call ***unilateral control***, specifies that each interface event is under the control of either the module or its environment, such that the occurrence of an interface event can be initiated only by the module or its environment (but not both). In particular, the set of interface events is partitioned into a set of ***input events*** controlled by the environment and a set of ***output events*** controlled by the module [14, 15]. The side (module or environment) with control of an interface event is the only one that can initiate the event's occurrence. The requirement of unilateral control restricts the class of interface specifications. This restricted class, however, appears to be adequate for specifying interface interactions for many kinds of input and output operations.

Note that while an interface event is initiated to occur by one side of an interface, its occurrence is impossible without participation by the other side. A third requirement, introduced in the model of I/O automata [15], is the following: each automaton is ***input-enabled***, i.e., every input event is enabled in every state of the automaton. With this requirement, the class of interface specifications is further restricted.

In the theory of modules and interfaces, interface interactions are characterized by both the requirements of simultaneous participation and unilateral control. However, modules are required to be input-enabled ***only when*** the occurrence of an input event would not violate any "safety" requirement of the module's interface; otherwise, occurrence of the input event ***may*** be blocked (disabled) by the module. Specifically, the module has a choice: it may block occurrence of the input event, but it is not required to do so.[1]

[1]Blocking is useful for specifying modules that have a finite

Notation A *sequence over E*, where E is a set, is a (finite or infinite) sequence $(e_0, e_1, \ldots)$, where $e_i \in E$ 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. □

2.2 Relational specifications

The semantic definitions of M *offers* I and M *using* L *offers* U given in [11, 12] are not easy to apply directly. For nontrivial applications, they were recast in the *relational notation*, a specification formalism with two basic constructs: state formulas that represent sets of states, and event formulas that represent sets of state transitions [10, 11]. In what follows, we present how state transition systems, modules and interfaces are specified in the relational notation. Conditions sufficient for the satisfaction of M *offers* I and M *using* L *offers* U, expressed in the relational notation, are then presented. For a more detailed presentation of the theory, the reader is referred to [11]; proofs can be found in [12].

To specify a state transition system, its state space is defined by a set of variables, called state variables. For system A, its set of state variables is denoted by $Variables(A)$. For each variable v, there is a set $domain(v)$ of allowed values. By definition, the set of system states is $States(A) = \prod_{v \in Variables(A)} domain(v)$. Each state $s \in States(A)$ is represented by a tuple of values, $(d_v : v \in Variables(A))$, where $d_v \in domain(v)$.

We use state formulas to represent subsets of $States(A)$. A *state formula* is a formula[2] 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.

We use event formulas to specify the transitions of events. An *event formula* is a formula in $Variables(A) \cup Variables(A)'$, where $Variables(A)' = \{v' : v \in Variables(A)\}$ 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 event formula, that is, the event 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.

Notation A *sequence over alternating E and F*, where E and F are sets, is a sequence $(e_0, f_0, e_1, f_1, \ldots)$, where $e_i \in E$ and $f_i \in F$ for all i. □

A *behavior* of A is a sequence $\sigma = (s_0, e_0, s_1, e_1, \ldots)$ over alternating $States(A)$ and $Events(A)$ such that s_0 satisfies $Initial_A$ and (s_i, s_{i+1}) satisfies $formula_A(e_i)$ 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 $Behaviors(A)$.

For $e \in Events(A)$, define

$$enabled_A(e) \equiv [\exists Variables(A)' : formula_A(e)]$$

which is a state formula representing the set of states $\{s : \text{for some state } t,\ (s, t) \text{ satisfies } formula_A(e)\}$. Event e is said to be enabled in a state s of A iff s satisfies $enabled_A(e)$. Event e is said to be disabled in state s iff it is not enabled in s.

Let us digress and state some conventions to be used in event formulas. Two examples of event definitions are shown below:

$$\begin{aligned} e_1 &\equiv v_1 > 2 \wedge v_2' \in \{1, 2, 5\} \\ e_2 &\equiv v_1 > v_2 \wedge v_1 + v_2' = 5 \end{aligned}$$

input buffer, e.g., a soda vending machine that has room to accept at most n coins prior to dispensing soda or returning coins, and is needed in the implementation of many communication protocols that enforce input control, flow control or congestion control.

[2] We use *formula* to mean a *well-formed formula* in the language of predicate logic.

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 $formula_A(e_2)$. By the following convention, it is assumed that v_1 is not changed by the occurrence of e_2.

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

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.

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 $Outputs(M) \cup Internals(M)$. 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. In a behavior $\sigma = (s_0, e_0, s_1, e_1, \ldots, s_j, e_j, \ldots)$, 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.

To specify an interface in the relational notation, we use a state transition system together with invariant and progress assertions. In what follows, we first introduce the assertions and then explain how the allowed event sequences of an interface are specified.

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.[3] A sequence $(s_0, e_0, s_1, e_1, \ldots)$ over alternating states and events satisfies P ***leads-to*** Q iff the following holds: $\forall i$: (s_i satisfies P) $\Rightarrow$ ($\exists j : j \geq i : s_j$ 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 the assertion. For a given σ, to evaluate the truth value of an assertion $Assert$, 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." The following is a sufficient condition:

[3] ***leads-to*** is the only temporal connective we use.

Definition A state transition system A has *deterministic events* iff

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

□

This condition is easy to satisfy because events in the Lam-Shankar theory can be regarded as names or labels. (Moreover, event names can be parameterized in the relational notation [10].) Note that an 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.

The restriction of a single initial state can also be circumvented by adding more events, as follows: Let s_0 denote a state not in $States(A)$, and $Init(A)$ the desired initial states of A. For all $s \in Init(A)$, specify a distinct event for each transition (s_0, s). Define s_0 to be the initial state of A.

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'. □

Convention For readability, the notation $sts(I)$ is abbreviated to I wherever such abbreviation causes no ambiguity, e.g., $States(sts(I))$ is abbreviated to $States(I)$, $formula_{sts(I)}(e)$ is abbreviated to $formula_I(e)$, etc. □

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

$$AllowedBehaviors(I) = \{\sigma : \sigma \in Behaviors(I) \text{ and } \sigma \text{ satisfies } \textit{invariant } InvReqs_I \text{ and } ProgReqs_I\}$$
$$AllowedEventSeqs(I) = \{image(\sigma, Events(I)) : \sigma \in AllowedBehaviors(I)\}.$$

where $image(\sigma, Events(I))$ denotes the sequence of events over $Events(I)$ obtained from σ by deleting all states and events that are not in $Events(I)$.

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 there are two ways to specify the safety requirements of an interface: namely, a state transition system, and a set of invariant requirements. Either one or—typically—both can be used for a particular specification. 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. The approach described above is a flexible one.

We next introduce a refinement relation between two state transition systems A and B such that $Variables(A) \supseteq Variables(B)$. Note that 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 [9, 10].

Notation Given $Variables(A) \supseteq Variables(B)$, for any $s \in States(A)$, if $s = (d_v : v \in Variables(A))$

where $d_v \in domain(v)$, then $image(s, B) = (d_v : v \in Variables(B))$. □

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 $Events(A) \supseteq Events(B)$
- $Initial_A \Rightarrow Initial_B$
- [event refinement]
  $\forall e \in Events(A) \cap Events(B)$:
  $Inv_A \wedge formula_A(e) \Rightarrow formula_B(e)$
- [null image condition]
  $\forall e \in Events(A) - Events(B)$:
  $Inv_A \wedge formula_A(e)$
  $\Rightarrow [\forall v \in Variables(B) : v = v']$ □

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 [10]. In this case, for any state formula P in $Variables(B)$, if B satisfies *invariant* P then A satisfies *invariant* P.

2.3 Specifying modules to satisfy interfaces

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
$(invariant\ InvAssum_I \Rightarrow ProgReqs_I)$

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

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
$((invariant\ (InvAssum_U \wedge InvGuar_L)$
$\wedge ProgReqs_L)$
$\Rightarrow ProgReqs_U)$

Let us point out an interesting feature in the semantic definitions of M *offers* I and M *using* L *offers* U, [11, 12]. Whenever the environment of M initiates an interface event whose occurrence would violate some interface safety requirement of M, module M can either block the event's occurrence or let the event occur. In the latter case, module M can behave arbitrarily after the event's occurrence, and still satisfy these semantic definitions. The **B** and **C** conditions are slightly stronger than the respective semantic definitions in that they distinguish between those safety requirements of an interface, say I, encoded in $sts(I)$ and those encoded in $InvAssum_I$. Conditions **B2**, **C2** and **C3** require that module M block the occurrence of any environment-controlled event that would violate an interface safety requirement encoded in $sts(I)$.

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

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

Given a compatible set of modules $\{M_j : j \in J\}$, their composition is a module M defined in the obvious way with each state of the composition being a tuple $s = (t_j : j \in J)$, where $t_j \in States(M_j)$, and output and input events that match each other becoming internal events of the composite module M.

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

- $Internals(M) \cap Internals(N) = \emptyset$
- *M using L offers U*
- *N offers L*

Then, M and N are compatible and their composition offers U. □

For a more detailed treatment of the theory of modules and interfaces, the reader is referred to [11]. A proof of Theorem 1 can be found in [12].

3 Case Study: A Secure Distributed System

Consider a distributed system where there is a collection of hosts, interconnected by a communication network. Denote the set of hosts by $\mathcal{H}$. Each host $h \in \mathcal{H}$ serves a distinct set $\mathcal{U}_h$ of subjects and maintains a distinct set $\mathcal{O}_h$ of objects.

We define a function *host* : $\mathcal{U} \cup \mathcal{O} \mapsto \mathcal{H}$ that returns the resident host for each subject or object. Its definition is:

$$host(x) = h \qquad \text{if } x \in \mathcal{U}_h \cup \mathcal{O}_h$$

We also use a boolean predicate *local* : $\mathcal{U} \times \mathcal{O} \mapsto \{\text{true}, \text{false}\}$ that returns true if subject u resides in the same host as object o; false otherwise. The formal definition is:

$$local(u, o) \equiv (host(u) = host(o))$$

The security concern of interest here is multilevel access control security. For simplicity, we do not include rules for changing subject clearance and object classification.[4] Thus both subject clearance and object classification are fixed. We denote the fixed clearance of a subject u by *clearance*(u), and the fixed classification of an object o by *classification*(o).

[4]It is known that potential security problems (covert channels) can arise if subject clearance and object classification can be dynamically varied [21].

Note that not all subjects and/or objects resident on a host possess the same clearance or classification. In other words, each host is trusted to operate at a range of security levels, denoted by $[h.lower, h.upper]$.[5] Thus, by definition:

$$\begin{array}{l} \forall\, x \in \mathcal{U} : host(x) = h \\ \quad \Rightarrow clearance(x) \in [h.lower, h.upper] \\ \forall\, x \in \mathcal{O} : host(x) = h \\ \quad \Rightarrow classification(x) \in [h.lower, h.upper] \end{array}$$

For each host h, both $h.lower$ and $h.higher$ are static and are fixed by a certification process (and possibly some administrative policies). We assume that each host maintains a table containing the security range of every other host in the distributed system. Also, each host possesses information of local subjects and objects only. In particular, a host does not know the clearances/classifications of remote subjects/objects.

An operational description of the distributed system is as follows: the activities of subjects are structured as *transactions*. A subject can issue two kinds of *commands*: *operations* and *rules*. Informally, operations can be used to modify the "value" component (e.g. object contents) of the system state, while rules can be used to modify the "protection" component (e.g. access matrix) of the system state. Each transaction is composed of a sequence of commands, delimited by a begin and an end command (both begin and end are operations). Each transaction is executed serially while different transactions (for different subjects) can be performed concurrently. Each subject can have at most one outstanding (active) transaction. If a command in a transaction names only local objects (objects maintained by the resident host of the subject), it can be carried out locally without invoking any network service; otherwise, communication services offered by an underlying communication network are needed.

The distributed system is structured into two layers as shown in Figure 1. The network module in Figure 1 denotes a layer that provides communication services for implementing remote operations. The system module offers a secure service to subjects for reading/writing objects, both local and remote. Note that the actual implementation of each module consists of a multiplicity of geographically distributed entities. In fact, each module may itself be structured as a hierarchy of modules.

[5]This means that the hosts are certified at least to the B1 level under the Trusted Security Evaluation Criteria [1].

For the layered system shown in Figure 1, there are two levels of security concerns. One level of security concern is at the user interface between subjects and the system. Here, we require access control security in the sense of Bell-LaPadula [4]. Another level of security concern is at the interface between the system module and the network module. Here, channels are established between pairs of hosts along which access requests and object contents are transferred. The security requirement of the network interface can be specified by a predicate $comm \subseteq \mathcal{H} \times \mathcal{H}$, which can be defined very generally. An example of such a requirement is the following simple variant of Bell-LaPadula:

$$\forall h, k \in \mathcal{H} : comm(h, k) \equiv$$
$$[h.lower, h.upper] \cap [k.lower, k.upper] \neq \emptyset$$

That is, a communication channel can be established between two hosts only if their security ranges overlap.

For our example, we propose just a small set of commands. In particular, we consider only these commands: the operations are **begin**, **end**, **read** and **write** and the rules are **get-obs**, **get-alt**, **rel-obs** and **rel-alt**. Informally, subjects invoke **read** to retrieve the content of an object, and **write** to update the content of an object. In order for these operations to be successful, the proper access rights must be first obtained by the subjects, and this is achieved through the use of rules: **get-obs** requests the observation right for an object, while **rel-obs** releases the observation right for an object; **get-alt** and **rel-alt** are similarly defined to request and release the alteration right for an object.

Note that the system being specified below may exhibit perculiar functional behaviors because we do not require the user interface to satisfy any consistency constraint (e.g. serializability). It is possible that different subjects simultaneously hold acess rights to write the same object. Such undesirable functional behaviors can be avoided by appropriately strenghtening the user interface specification. This illustrates the flexibility of the theory of modules and interfaces. For a formal specification of a seriablizable database interface given in the relational notation, the reader is referred to [11].

3.1 Notation

We specify an event as a tuple, with subject name, object name, host name and some event-specific parameters as typical components. For example, (u, read, o) denotes the event corresponding to a read command: u is the subject requesting the read, **read** specifies the command type, and o is the object being read. We prefix an input event with in and an output event with out. Thus $\mathsf{in}(u, \mathsf{read}, o)$ is an input event (of a particular interface or module). A subscript is used to denote the interface or module an event belongs, e.g. if U is an interface, then $\mathsf{in}_U(u, \mathsf{read}, o)$ is an input event of interface U. Since there is only one module (M) specified in this paper, events of module M do not have subscripts.

We introduce several functions for accessing the components of a tuple. The functions *sbj*, *cmd*, *obj*, *val*, and *hst* return respectively the subject component, the command type component, the object component, the value component, and the host component of a tuple if such a component exists; if the specified component does not exist, **undefined** is returned. For example, consider tuple $t = (u, \mathsf{read}, o, v)$, then $sbj(t) = u$, $cmd(t) = \mathsf{read}$, $obj(t) = o$, $val(t) = v$ while $hst(c) = \mathsf{undefined}$. We omit the precise definitions of these functions.

We define that a tuple t_1 *is contained in* another tuple t_2 if all the components in t_1 are also in t_2. Thus $(u, \mathsf{rel\text{-}obs}, o)$ is contained in both $(u, \mathsf{rel\text{-}obs}, o, \mathsf{ok})$ and $(u, \mathsf{rel\text{-}obs}, o, \mathsf{failed})$. We also say t_2 *contains* t_1 if t_1 is contained in t_2. Let T be a sequence over a set of tuples. For a tuple t, $proj_t(T)$ denotes the subsequence of tuples in T each of which contains the tuple t. For example, $proj_{(u,\mathsf{rel\text{-}obs},o)}(T)$ returns the subsequence of tuples in T each of which contains $(u, \mathsf{rel\text{-}obs}, o)$.

A *generic* command specifies a class of related commands, and is written using a wildcard symbol '$*$'. For example, $*$-obs specifies the class of commands $\{\mathsf{get\text{-}obs}, \mathsf{rel\text{-}obs}\}$; **get-obs** and **rel-obs** are called *specific* commands *corresponding to* $*$-obs. The correspondence between generic commands and specific commands should be self-explanatory in the following; hence precise definitions of such correspondence are omitted. We extend the above containment relationship to the use of generic commands: a tuple t_2 contains a tuple t_1 if t_1's command component c is generic while t_2's command component is a specific command corresponding to c, and all other components of t_1 are in t_2. For example, $(u, *\text{-}\mathsf{obs}, o)$ is contained in both $(u, \mathsf{get\text{-}obs}, o, \mathsf{failed})$ and $(u, \mathsf{rel\text{-}obs}, o, \mathsf{ok})$.

$(T)_i$ returns the ith element in the sequence T and $T@t$ represents the sequence T with t appended. $last(T)$ returns the last element in T; and **undefined** if T is empty.

In the following, u, c, o, h and v are used respectively as variables standing for an subject, a command name, an object, a host, and a value unless explicitly stated otherwise.

3.2 The user interface U

Operations	Rules
$\mathsf{in}_U(u, \mathsf{begin})$	$\mathsf{in}_U(u, \mathsf{get\text{-}obs}, o)$
$\mathsf{out}_U(u, \mathsf{begin}, \mathsf{ok})$	$\mathsf{out}_U(u, \mathsf{get\text{-}obs}, \mathsf{ok})$
$\mathsf{out}_U(u, \mathsf{begin}, \mathsf{failed})$	$\mathsf{out}_U(u, \mathsf{get\text{-}obs}, \mathsf{failed})$
$\mathsf{in}_U(u, \mathsf{end})$	$\mathsf{in}_U(u, \mathsf{get\text{-}alt}, o)$
$\mathsf{out}_U(u, \mathsf{end}, \mathsf{ok})$	$\mathsf{out}_U(u, \mathsf{get\text{-}alt}, \mathsf{ok})$
$\mathsf{in}_U(u, \mathsf{read}, o)$	$\mathsf{out}_U(u, \mathsf{get\text{-}alt}, \mathsf{failed})$
$\mathsf{out}_U(u, \mathsf{read}, o, v)$	$\mathsf{in}_U(u, \mathsf{rel\text{-}obs}, o)$
$\mathsf{out}_U(u, \mathsf{read}, o, \mathsf{failed})$	$\mathsf{out}_U(u, \mathsf{rel\text{-}obs}, o, \mathsf{ok})$
$\mathsf{in}_U(u, \mathsf{write}, o, v)$	$\mathsf{in}_U(u, \mathsf{rel\text{-}alt}, o)$
$\mathsf{out}_U(u, \mathsf{write}, o, \mathsf{ok})$	$\mathsf{out}_U(u, \mathsf{rel\text{-}alt}, o, \mathsf{ok})$
$\mathsf{out}_U(u, \mathsf{write}, o, \mathsf{failed})$	

Figure 3: Events of User Interface U

We specify the user interface U in the relational notation as defined in Section 2.2. The events of interface U are listed in Figure 3. For each subject-command pair, we have an input event; and for each input event, there may be zero or more output events representing the system's response to the input event. We use two distinct constants ok, failed $\notin \mathcal{V}$ to indicate respectively success and failure.

Variables of ***sts***(U)

- $status_U(u) : \{\mathsf{idle}, \mathsf{ready}\} \cup \{(u, \mathsf{begin}), (u, \mathsf{end})\} \cup \{(u, c, o) : c \in \{\mathsf{read}, \mathsf{write}, \mathsf{get\text{-}*}\}\}$

  For each subject u, $status_U(u)$ records its state of transaction execution. If there is an outstanding transaction for u, $status_U(u)$ would record a constant value of ready (the interface U is waiting for the next command) or the command currently being executed by u in that transaction. If there is no outstanding transaction for u, $status_U(u)$ would assume the constant value idle. Initially $status_U(u)$ is idle for all u.

- T_U : sequence over $\{(u, c, o, r) : c = \mathsf{get\text{-}*}, r \in \{\mathsf{ok}, \mathsf{failed}\}\} \cup \{(u, c, o, \mathsf{ok}) : c = \mathsf{rel\text{-}*}\}$

  T_U records the sequence of rule events (access grants and denials) that have occurred so far for a transaction. The current "protection state" can be determined by examining T_U. For instance, if a $(u, \mathsf{get\text{-}obs}, o, \mathsf{ok})$ event is in T_U and there is no subsequent $(u, \mathsf{rel\text{-}obs}, o, \mathsf{ok})$ event, then we can conclude that u holds observation right for o. T_U is initially set to nil, a constant not in $\mathcal{V}$. (T_U becomes an auxiliary variable in the system module M.)

We define two state functions to be used in event specifications.

- $observable_U : \mathcal{U} \times \mathcal{O} \mapsto \{\mathsf{true}, \mathsf{false}\}$ defined by

  $$observable_U(u, o) \equiv last(proj_{(u,*\text{-}\mathsf{obs},o)}(T_U)) = (u, \mathsf{get\text{-}obs}, o, \mathsf{ok})$$

- $alterable_U : \mathcal{U} \times \mathcal{O} \mapsto \{\mathsf{true}, \mathsf{false}\}$ defined by

  $$alterable_U(u, o) \equiv last(proj_{(u,*\text{-}\mathsf{alt},o)}(T_U)) = (u, \mathsf{get\text{-}alt}, o, \mathsf{ok})$$

Note that, $observable_U(u, o)$ (respectively $alterable_U(u, o)$) is true in a state s if u has observation (respectively alteration) right for o in s.

Events of ***sts***(U)

The event specifications of interface U are given in Figure 4. Several desirable safety properties are encoded in the events of $sts(U)$:

- An out event must be preceded by a corresponding in event, i.e. output events do not occur spontaneously.
- Each subject has at most one outstanding transaction at a time.
- A subject succeeds in reading an object only if the subject has observation right for that object.
- A subject succeeds in writing an object only if the subject has alteration right for that object.

We make several other observations: First, the precise condition for accepting a transaction from u is left unspecified (same enabling condition for $\mathsf{out}_U(u, \mathsf{begin}, \mathsf{ok})$ and $\mathsf{out}_U(u, \mathsf{begin}, \mathsf{failed})$). Thus a module that offers U can impose various implementation-dependent conditions without violating the interface specification, e.g. a transaction is accepted only if the system has sufficient resources. Such nondeterminism in an interface specification allows a wide variety of implementations, and is preferred when more specific conditions are either not known or not necessary.

$$\begin{aligned}
\mathsf{in}_U(u,\mathsf{begin}) &\equiv \mathit{status}_U(u)=\mathsf{idle}\\
&\quad \wedge\ \mathit{status}_U(u)'=(u,\mathsf{begin})\\
\mathsf{out}_U(u,\mathsf{begin},\mathsf{ok}) &\equiv \mathit{status}_U(u)=(u,\mathsf{begin})\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}\\
\mathsf{out}_U(u,\mathsf{begin},\mathsf{failed}) &\equiv \mathit{status}_U(u)=(u,\mathsf{begin})\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{idle}\\
\mathsf{in}_U(u,\mathsf{end}) &\equiv \mathit{status}_U(u)=\mathsf{ready}\\
&\quad \wedge\ \mathit{status}_U(u)'=(u,\mathsf{end})\\
\mathsf{out}_U(u,\mathsf{end},\mathsf{ok}) &\equiv \mathit{status}_U(u)=(u,\mathsf{end})\\
&\quad \wedge\ T_U'=\mathsf{nil}\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{idle}\\
\mathsf{in}_U(u,\mathsf{get\text{-}obs},o) &\equiv \mathit{status}_U(u)=\mathsf{ready}\\
&\quad \wedge\ \mathit{status}_U(u)'=(u,\mathsf{get\text{-}obs},o)\\
\mathsf{out}_U(u,\mathsf{get\text{-}obs},o,\mathsf{ok}) &\equiv \mathit{status}_U(u)=(u,\mathsf{get\text{-}obs},o)\\
&\quad \wedge\ T_U'=T_U@(u,\mathsf{get\text{-}obs},o,\mathsf{ok})\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}\\
\mathsf{out}_U(u,\mathsf{get\text{-}obs},o,\mathsf{failed}) &\equiv \mathit{status}_U(u)=(u,\mathsf{get\text{-}obs},o)\\
&\quad \wedge\ T_U'=T_U@(u,\mathsf{get\text{-}obs},o,\mathsf{failed})\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}\\
\mathsf{in}_U(u,\mathsf{rel\text{-}obs},o) &\equiv \mathit{status}_U(u)=\mathsf{ready}\ \wedge\ \mathit{observable}_U(u,o)\\
&\quad \wedge\ \mathit{status}_U(u)'=(u,\mathsf{rel\text{-}obs},o)\\
\mathsf{out}_U(u,\mathsf{rel\text{-}obs},o,\mathsf{ok}) &\equiv \mathit{status}_U(u)=(u,\mathsf{rel\text{-}obs},o)\\
&\quad \wedge\ T_U'=T_U@(u,\mathsf{rel\text{-}obs},o,\mathsf{ok})\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}\\
\mathsf{in}_U(u,\mathsf{read},o) &\equiv \mathit{status}_U(u)=\mathsf{ready}\\
&\quad \wedge\ \mathit{status}_U(u)'=(u,\mathsf{read},o)\\
\mathsf{out}_U(u,\mathsf{read},o,v) &\equiv \mathit{status}_U(u)=(u,\mathsf{read},o)\ \wedge\ \mathit{observable}_U(u,o)\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}\\
\mathsf{out}_U(u,\mathsf{read},o,\mathsf{failed}) &\equiv \mathit{status}_U(u)=(u,\mathsf{read},o)\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}\\
\mathsf{in}_U(u,\mathsf{write},o,v) &\equiv \mathit{status}_U(u)=\mathsf{ready}\\
&\quad \wedge\ \mathit{status}_U(u)'=(u,\mathsf{write},o,v)\\
\mathsf{out}_U(u,\mathsf{write},o,\mathsf{ok}) &\equiv \mathit{status}_U(u)=(u,\mathsf{write},o,v)\ \wedge\ \mathit{alterable}_U(u,o)\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}\\
\mathsf{out}_U(u,\mathsf{write},o,\mathsf{failed}) &\equiv \mathit{status}_U(u)=(u,\mathsf{write},o,v)\\
&\quad \wedge\ \mathit{status}_U(u)'=\mathsf{ready}
\end{aligned}$$

The event formulas for $*$-alt rules are similar to those for $*$-obs, and are omitted here.

Figure 4: Event Specifications of U

On the other hand, the end, rel-obs and rel-alt events always succeed, as indicated by the absence of a failed output event for each of them. This is so because the condition for success has already been incorporated in the enabling condition of the respective input event.

The enabling conditions for $\mathsf{out}_U(u,\mathsf{read},o,v)$ and $\mathsf{out}_U(u,\mathsf{read},o,\mathsf{failed})$ are not mutually exclusive: it is $\mathit{observable}_U(u,o)$ for $\mathsf{out}_U(u,\mathsf{read},o,v)$ and true for $\mathsf{out}_U(u,\mathsf{read},o,\mathsf{failed})$ (in addition to $\mathit{status}_U(u) = (u,\mathsf{read},o)$ for both). Thus, a read can always fail even when the subject u possesses the necessary observation rights. Such behavior would be removed in an implementation, i.e. the system module M.

The get-$*$ events can be invoked by a subject even when the subject already has the access right being requested. This generality is useful for handling the case in which object classifications may change dynamically. (But in the system module M to be specified, we assume that object classifications are fixed, which gives rise to an optimized implementation.)

Lastly, only the security-relevant semantics of interface U is specified. For example, the above specification does not require the value returned by a successful read to be the same as the value currently stored, nor the object content be updated by a successful **write**. This is consistent with our proposal that security specification and functional specification should be separate; the semantics of **read** and **write** can be specified by appropriately strengthening their respective event formulas with additional conjuncts, or by including extra invariant requirements below.

Invariant requirements of U

$$
\begin{aligned}
&Inv_{U,1} \equiv \forall i : (T_U)_i = (u, \mathsf{get\text{-}obs}, o, \mathsf{ok}) \\
&\quad \Rightarrow classification(o) \preceq clearance(u) \\
&\qquad \wedge [\neg local(u,o) \Rightarrow comm(host(u), host(o))] \\
&Inv_{U,2} \equiv \forall i : (T_U)_i = (u, \mathsf{get\text{-}alt}, o, \mathsf{ok}) \\
&\quad \Rightarrow clearance(u) \preceq classification(o) \\
&\qquad \wedge [\neg local(u,o) \Rightarrow comm(host(u), host(o))]
\end{aligned}
$$

$Inv_{U,1}$ and $Inv_{U,2}$ are similar to the s-secure and $\star$-secure properties of the Bell-LaPadula model, but are stated using T_U.

$$
\begin{aligned}
InvAssum_U &\equiv \mathsf{true} \\
InvGuar_U &\equiv Inv_{U,1} \wedge Inv_{U,2}
\end{aligned}
$$

Progress requirements of U

The requirement that every user command must be processed in a finite time (i.e. an input event must be followed by a corresponding output event in a finite number of steps) is obviously a nice one. However, it may not be satisfiable. Also, whether such a requirement should be considered a security concern depends on the perceived threats. If the system operates in an environment that is potentially malicious and denial of service is considered a threat, then the finite response time requirement should be included as a security requirement. Such a requirement for interface U can be formalized as follows:

$$
\begin{aligned}
&ProgReqs_U \equiv \forall u : \\
&status_U(u) \notin \{\mathsf{idle}, \mathsf{ready}\} \;\; \textit{leads-to} \;\; status_U(u) = \mathsf{ready}
\end{aligned}
$$

3.3 The network interface L

For the network interface, all events are concerned with commands involving remote objects. In fact, when a command c is executed by a subject at the user interface: if c names only local objects, the output response events are executed by the system module without making use of the service offered by L. On the other hand, if c names remote objects, the command is relayed down to L, in which case a response event at U to the user cannot occur until a response event at L has occurred.

In order for a host to communicate with another host, communication channels have to be first established. Channels are created dynamically on a per subject basis between two hosts. That is, if two subjects resident on the same host request the same remote object, two separate channels between the hosts are created, one for each subject. Channels are unidirectional, in the sense that it can be used only by one side to pass commands and retrieve results. Channels persist only for the duration of a transaction and are torn down when the transaction terminates. Note that channels are merely a logical abstraction suitable for our analysis. Further refinements may be needed to obtain realistic implementations, e.g. multiplexing, logical separation by encryption, etc.

Since channels are established on a per subject basis and a subject is resident on exactly one host, we can uniquely identify a channel by the subject and the remote host at the other end of the channel.

We follow the OSI model for peer entity communications. The peer entities are embedded in the system module. For entity A to initiate a communication with a peer entity B, entity A invokes a ***request*** event at the network interface. Subsequently, a corresponding ***indication*** event is output to B at the interface. B responds to the indication event with an input ***response*** event to the interface. The content of the response would subsequently be conveyed to A in an output ***confirm*** event.

Variables of $sts(L)$

- $status_L(h,u)$:$\{\mathsf{ready}\}$
  $\cup\{(u,c,h) : c \in \{\mathsf{conn\text{-}*}, \mathsf{disconn\text{-}req}\}\}$
  $\cup\{(u,c,o) : c \in \{\mathsf{read\text{-}*}, \mathsf{write\text{-}*}, \mathsf{get\text{-}*\text{-}*}\}\}$

  For each host h and subject u, $status_L(h,u)$ records the current activity of u on host h (with respect to the network interface). If h is the resident host of u, $status_L(h,u)$ indicates the request whose response from a remote host u is waiting for, or **ready** if there is no such outstanding request. In the case that h is not the resident host of u, $status_L(h,u)$ records whether an indication event has been received on behalf of u and a response from host h is pending; **ready** denotes the absence of such a pending response. Initially, $status_L(h,u)$ is **ready** for all h and u.

- $outchannel(u) : \mathcal{H}$

  For each subject u, $outchannel(u)$ denotes a set containing all hosts to which u (at host $host(u)$) has established a communication channel with.

- $inchannel(h) : \mathcal{H}$

  For each h, $inchannel(h)$ is a set containing all remote hosts that have established a communication channel to host h.

Both $outchannel(u)$ and $inchannel(h)$ are initially $\emptyset$.

- T_L : sequence over $\{(u, c, o, r) : c = \text{get-}*,$
  $r \in \{\text{ok}, \text{failed}\}\} \cup \{(u, c, o, \text{ok}) : c = \text{rel-}*\}$

  T_L is maintained for stating the Bell-LaPadula security requirements. It records all the access grants/denials between hosts at the network interface. T_L is initially nil. (T_U becomes an auxiliary variable in the system module M.)

- $result(h, u) : \{\text{nil}, \text{ok}, \text{failed}\} \cup \mathcal{V}$

  For each indication event, two response events are possible (e.g. one to indicate success and another to indicate failure). We use the variable $result(h, u)$ to relay the parameters in a response event to its corresponding confirm event. For example, a response event indicating success sets $result(h, u)$ to ok, which is used to generate a corresponding confirm event signifying success.

Events of $sts(L)$

The event specifications of interface L are given in Figure 5 (connection and disconnection events) and Figure 6 (access request and read/write events).

Several observations are in order here. First, a conn-req event requires that the channel requested be nonexistent, while a disconn-req event requires that the channel to be torn down be an established one. Second, the condition for a remote host to accept a connection request is unspecified, thus an implementation is allowed to impose specific conditions. Third, interface L as specified herein offers reliable message delivery; thus the disconn-req event does not require a disconn-con as an acknowledgement.

The events read-req, write-req, get-obs-req, and get-alt-req require a channel to have been established. Events read-req and write-req further require the appropriate rights to have been granted. Again, the condition for a successful read or write is left unspecified. For example, a disk failure at the remote host can cause a read or write to fail even when the subject has the necessary rights.

Note that the clearance of a subject has to be explicited passed as a value of the parameter n in the get-obs and get-alt events, since the remote host does not possess information regarding a foreign subject.

The rel-obs-req and rel-alt-req events are not strictly speaking needed at this level, since the appropriate rights for the subjects involved can be released locally. The events are included here to preserve the meaning of T_L which records all access grants and denials for remote objects. However, unlike their counterparts rel-obs and rel-alt at U, rel-obs-req and rel-alt-req do not check possession of rights; hence it is possible to release an unacquired right. Also, rel-obs-req and rel-alt-req, unlike get-obs-req and get-alt-req, do not require the remote host where the object resides to be notified. For applications that require a host to have precise knowledge of the propagation of rights for its objects (e.g. in distributed mutual exclusion), appropriate indication events will have to be added.

Most of the safety requirements of L are specified in the state transition system. For example, the strict sequencing requirement concerning peer communications (i.e. request $\rightarrow$ indication $\rightarrow$ response $\rightarrow$ confirm) is specified using the variable $status_L(h, u)$. The requirement that command executions can proceed only after the appropriate channels have been established is specified by including the conjunct $host(o) \in outchannel(u)$ in the event formulas.

In the following, $observable_L$ and $alterable_L$ are state functions defined similar to their counterparts, $observable_U$ and $alterable_U$, in U, except that all references to T_U are replaced by references to T_L.

Invariant requirements of L

An invariant assumption of the network interface is that all interface events must correspond to remote operations, i.e., the user of L can only invoke events naming remote objects at L. Formally,

$$
\begin{aligned}
&InvAssum_L \equiv \\
&\quad [\forall h, u : (status_L(h, u) = (u, c, h) \\
&\qquad\qquad \wedge\, c \in \{\text{conn-}*, \text{disconn-}*\}) \\
&\qquad\qquad\quad \Rightarrow host(u) \neq h] \\
&\quad \wedge \\
&\quad [\forall h, u, o : (status_L(h, u) = (u, c, o) \\
&\qquad\qquad \wedge\, c \notin \{\text{conn-}*, \text{disconn-}*\}) \\
&\qquad\qquad\quad \Rightarrow \neg local(u, o)]
\end{aligned}
$$

$$
\begin{array}{rcl}
\text{in}_L(u, \text{conn-req}, h) & \equiv & status_L(host(u), u) = \text{ready} \\
 & & \wedge\, h \notin outchannel(u) \\
 & & \wedge\, [\neg comm(host(u), h) \;\Rightarrow\; result(host(u), u)' = \text{failed}] \\
 & & \wedge\, [comm(host(u), h) \;\Rightarrow\; result(host(u), u)' = \text{nil}] \\
 & & \wedge\, status_L(host(u), u)' = (u, \text{conn-req}, h) \\
\text{out}_L(u, \text{conn-ind}, h) & \equiv & status_L(host(u), u) = (u, \text{conn-req}, h) \\
 & & \wedge\, status_L(h, u) = \text{ready} \;\wedge\; result(host(u), u) = \text{nil} \\
 & & \wedge\, status_L(h, u)' = (u, \text{conn-ind}, h) \\
\text{in}_L(u, \text{conn-res}, h, \text{ok}) & \equiv & status_L(h, u) = (u, \text{conn-ind}, h) \\
 & & \wedge\, inchannel(h)' = inchannel(h) \cup \{host(u)\} \\
 & & \wedge\, result(host(u), u)' = \text{ok} \\
 & & \wedge\, status_L(h, u)' = \text{ready} \\
\text{in}_L(u, \text{conn-res}, h, \text{failed}) & \equiv & status_L(h, u) = (u, \text{conn-ind}, h) \\
 & & \wedge\, result(host(u), u)' = \text{failed} \\
 & & \wedge\, status_L(h, u)' = \text{ready} \\
\text{out}_L(u, \text{conn-con}, h, \text{ok}) & \equiv & status_L(host(u), u) = (u, \text{conn-req}, h) \\
 & & \wedge\, result(host(u), u) = \text{ok} \\
 & & \wedge\, outchannel(u)' = outchannel(u) \cup \{h\} \\
 & & \wedge\, status_L(host(u), u)' = \text{ready} \\
\text{out}_L(u, \text{conn-con}, h, \text{failed}) & \equiv & status_L(host(u), u) = (u, \text{conn-req}, h) \\
 & & \wedge\, result(host(u), u) \notin \{\text{nil}, \text{ok}\} \\
 & & \wedge\, status_L(host(u), u)' = \text{ready} \\
\text{in}_L(u, \text{disconn-req}, h) & \equiv & status_L(host(u), u) = \text{ready} \;\wedge\; h \in outchannel(u) \\
 & & \wedge\, outchannel(u)' = outchannel(u) - \{h\} \\
 & & \wedge\, status_L(host(u), u)' = (u, \text{disconn-req}, h) \\
\text{out}_L(u, \text{disconn-ind}, h) & \equiv & status_L(host(u), u) = (u, \text{disconn-req}, h) \\
 & & \wedge\, inchannel(h)' = inchannel(h) - \{host(u)\} \\
 & & \wedge\, status_L(host(u), u)' = \text{ready}
\end{array}
$$

Figure 5: Event Specifications of L

$$
\begin{array}{rcl}
Inv_{L,1} & \equiv & \forall\, i : (T_L)_i = (u, \text{get-obs}, o, \text{ok}) \\
 & & \Rightarrow\; classification(o) \preceq clearance(u) \\
Inv_{L,2} & \equiv & \forall\, i : (T_L)_i = (u, \text{get-obs}, o, \text{failed}) \\
 & & \Rightarrow\; \neg(classification(o) \preceq clearance(u)) \\
Inv_{L,3} & \equiv & \forall\, i : (T_L)_i = (u, \text{get-alt}, o, \text{ok}) \\
 & & \Rightarrow\; clearance(u) \preceq classification(o) \\
Inv_{L,4} & \equiv & \forall\, i : (T_L)_i = (u, \text{get-alt}, o, \text{failed}) \\
 & & \Rightarrow\; \neg(clearance(u) \preceq classification(o))
\end{array}
$$

$$InvGuar_L \equiv Inv_{L,1} \wedge Inv_{L,2} \wedge Inv_{L,3} \wedge Inv_{L,4}$$

The conjuncts $Inv_{L,1}$ and $Inv_{L,3}$ are similar to statements of the Bell-LaPadula model.

Progress requirements of L

A progress requirement avoiding denial of service can be stated for L as shown below. Note that this also takes care of the finite response time requirement for connection and disconnection requests.

$$
\begin{array}{l}
ProgReqs_L \equiv \forall\, h, u : \\
status_L(h, u) \neq \text{ready} \;\; \textit{leads-to} \;\; status_L(h, u) = \text{ready}
\end{array}
$$

3.4 System module M

Variables of $sts(M)$

In addition to the variables in $sts(U)$ and $sts(L)$, the system module has the following variables:

- $access(u) : \{(r, o) : r \in \{\text{obs}, \text{alt}\}\}$

  For each subject u, $access(u)$ records all access rights currently held by u. This variable is updated by successful returns of the **get-obs**, **get-alt**, **rel-obs** and **rel-alt** requests, and is used to determine if a **read** or **write** request from u can be honored or not. The roles of both T_U and T_L are replaced by $access(u)$, and hence they become auxiliary variables in M. (For a formal treatment of auxiliary variables, see [10].)

$$
\begin{aligned}
\text{in}_L(u, \text{get-obs-req}, o, n) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = \text{ready} \wedge \mathit{host}(o) \in \mathit{outchannel}(u)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u)' = \text{nil}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = (u, \text{get-obs-req}, o, n)\\
\text{out}_L(u, \text{get-obs-ind}, o, n) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = (u, \text{get-obs-req}, o, n)\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u) = \text{ready} \wedge \mathit{host}(u) \in \mathit{inchannel}(h)\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u)' = (u, \text{get-obs-ind}, o, n)\\
\text{in}_L(u, \text{get-obs-res}, o, \text{ok}) \;&\equiv\; \mathit{status}_L(\mathit{host}(o), u) = (u, \text{get-obs-ind}, o, n)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u)' = \text{ok}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u)' = \text{ready}\\
\text{in}_L(u, \text{get-obs-res}, o, \text{failed}) \;&\equiv\; \mathit{status}_L(\mathit{host}(o), u) = (u, \text{get-obs-ind}, o, n)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u)' = \text{failed}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u)' = \text{ready}\\
\text{out}_L(u, \text{get-obs-con}, o, \text{ok}) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = (u, \text{get-obs-req}, o, n)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u) = \text{ok}\\
&\quad \wedge\, T_L' = T_L @ (u, \text{get-obs}, o, \text{ok})\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = \text{ready}\\
\text{out}_L(u, \text{get-obs-con}, o, \text{failed}) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = (u, \text{get-obs-req}, o, n)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u) \notin \{\text{nil}, \text{ok}\}\\
&\quad \wedge\, T_L' = T_L @ (u, \text{get-obs}, o, \text{failed})\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = \text{ready}\\
\text{in}_L(u, \text{rel-obs-req}, o) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = \text{ready}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = (u, \text{rel-obs-req}, o)\\
\text{out}_L(u, \text{rel-obs-con}, o, \text{ok}) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = (u, \text{rel-obs-req}, o)\\
&\quad \wedge\, T_L' = T_L @ (u, \text{rel-obs}, o, \text{ok})\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = \text{ready}\\
\text{in}_L(u, \text{read-req}, o) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = \text{ready} \wedge \mathit{host}(o) \in \mathit{outchannel}(u)\\
&\quad \wedge\, \mathit{observable}_L(u, o)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u)' = \text{nil}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = (u, \text{read-req}, o)\\
\text{out}_L(u, \text{read-ind}, o) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = (u, \text{read-req}, o)\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u) = \text{ready} \wedge \mathit{host}(u) \in \mathit{inchannel}(h)\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u)' = (u, \text{read-ind}, o)\\
\text{in}_L(u, \text{read-res}, o, v) \;&\equiv\; \mathit{status}_L(\mathit{host}(o), u) = (u, \text{read-ind}, o)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u)' = v\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u)' = \text{ready}\\
\text{in}_L(u, \text{read-res}, o, \text{failed}) \;&\equiv\; \mathit{status}_L(\mathit{host}(o), u) = (u, \text{read-ind}, o)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u)' = \text{failed}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(o), u)' = \text{ready}\\
\text{out}_L(u, \text{read-con}, o, v) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = (u, \text{read-req}, o)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u) = v \wedge v \in \mathcal{V}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = \text{ready}\\
\text{out}_L(u, \text{read-con}, o, \text{failed}) \;&\equiv\; \mathit{status}_L(\mathit{host}(u), u) = (u, \text{read-req}, o)\\
&\quad \wedge\, \mathit{result}(\mathit{host}(u), u) \notin \{\text{nil}\} \cup \mathcal{V}\\
&\quad \wedge\, \mathit{status}_L(\mathit{host}(u), u)' = \text{ready}
\end{aligned}
$$

The event formulas corresponding to write-$*$ and $*$-alt-$*$ are similar to those for read-$*$ and $*$-obs-$*$, and are omitted for brevity.

Figure 6: Event Specifications of L (cont.)

- $\mathit{tolower}(u) : \{(u, \text{disconn-req}, D) : D \subseteq \mathcal{H}\}$
  $\cup\{(u, c, o) : c \in \{\text{read}, \text{write}, \text{get-}*, \text{rel-}*\}\}$

When an input event of U naming a remote object o is submitted by a subject u, and it cannot be handled locally, the request is encoded in *tolower*(u). Most input events of L are enabled by a condition on $tolower(u)$. For example, consider an $\mathsf{in}(u, \mathsf{read}, o)$ event submitted by u when $\neg local(u, o)$ holds. This event requests the value $content(o)$ of a remote object (maintained by $host(o)$) and cannot be handled locally by $host(u)$. If u has the observation right for o, $tolower(u)$ is set to (u, read, o), which in turn enables the output event $\mathsf{out}(u, \mathsf{read\text{-}req}, o)$ at L. In some sense, $tolower(u)$ acts as a communication channel from U to L, along which instructions for activating the appropriate events of L are passed.

- $toupper(u) : \{(u, c, o, r) : c \in \{\mathsf{read}, \mathsf{write}, \mathsf{get\text{-}*}\}, r \in \{\mathsf{ok}, \mathsf{failed}\}\}$

  $toupper(u)$ can be viewed as a reverse communication channel from L to U, along which the results of L events are passed to the user interface so that the appropriate output response events can be generated at U.

  Both $tolower(u)$ and $toupper(u)$ are initially nil.

Events of $sts(M)$

The event specifications of M are given in Figures 7 and 8. The following notation is used in these specifications:

Notation Let G and A be two formulas such that G names only unprimed variables, while A names both primed and unprimed variables. Define

$$G \rightarrow A \equiv (G \Rightarrow A) \wedge (\neg G \Rightarrow \forall x \text{ in } A : x' = x)$$

where x in A is true if variable x' appears in the formula A. Informally, $G \rightarrow A$ says that if the guard G is true, then the state change should be as specified by action A; if G is false, then action A is not performed. □

Each event of M is obtained by refining an event of U or L. To satisfy the finer-grain atomicity requirements of a practical programming language, each interface event may have to be refined into a sequence of module events. For a discussion on how to accomplish this, the reader is referred to Section 7 of [11].

By taking advantage of the assumption that an object's classification is fixed, we provide implementations of **get-obs** and **get-alt** that are optimized as follows: If the appropriate right for a remote object has already been granted previously (and not yet released), **get-obs** and **get-alt** return immediately without invoking L for network services.

Fairness requirements of M

$$F_u = \{e \in Outputs(M) : sbj(e) = u\}$$
$$\text{Fairness requirements of } M = \{F_u : u \in \mathcal{U}\}$$

These requirements ensure that no subject be denied service because of other subjects.

3.5 Satisfaction of M *using* L *offers* U

To prove that the system module satisfies M *using* L *offers* U, we need a state formula Inv_M in $Variables(M)$ such that conditions **C1**–**C9** presented in Section 2.3 are satisfied.

We propose a formula Inv_M that is a conjunction of state formulas. Those conjuncts that are sufficient for proving satisfaction of conditions **C1**–**C7** are shown in Figure 9. To prove **C8** by applying an invariance proof rule [11], other state formulas in addition to the ones shown in Figure 9 are needed.

Lastly, while it appears that condition **C9** is satisfied by module M, a formal proof applying inference rules for proving leads-to assertions has not been carried out.

4 Discussions

In the approach of this paper, security concerns are stated as interface requirements. Interfaces at different levels of a layered architecture can have different kinds of security concerns. Furthermore, functional requirements and security requirements of an interface can be specified separately. For example, serializability of interface U in this paper can be specified and satisfied separately from the requirement that interface U is multilevel secure.

The definition of safety in the theory of modules and interfaces [11, 12] is general, and can accommodate notions of secure information flow other than Bell-LaPadula (e.g., deducibility, noninterference). The Bell-LaPadula requirements are used herein because their statements are relatively simple.

The theory of modules and interfaces provides a theoretical foundation for the design and specification of systems structured as a linear hierarchy of layers and also as a set of modules organized as the nodes of a rooted tree. Applying composition theorems in the theory, each module or layer with well-defined interfaces, say M with upper interface U and lower interface L, can be designed, implemented, and modified

$$
\begin{array}{rcl}
\mathsf{in}(u,\mathsf{begin}) & \equiv & \mathsf{in}_U(u,\mathsf{begin}) \\
\mathsf{out}(u,\mathsf{begin},\mathsf{ok}) & \equiv & \mathsf{out}_U(u,\mathsf{begin},\mathsf{ok}) \\
\mathsf{out}(u,\mathsf{begin},\mathsf{failed}) & \equiv & \mathsf{out}_U(u,\mathsf{begin},\mathsf{failed}) \\
\mathsf{in}(u,\mathsf{end}) & \equiv & \mathsf{in}_U(u,\mathsf{end}) \\
 & & \wedge\,[\mathit{outchannel}(u) \neq \emptyset \\
 & & \qquad \rightarrow\ \mathit{tolower}(u)' = (u,\mathsf{disconn\text{-}req},\mathit{outchannel}(u))] \\
\mathsf{out}(u,\mathsf{end},\mathsf{ok}) & \equiv & \mathsf{out}_U(u,\mathsf{end},\mathsf{ok})\ \wedge\ \mathit{outchannel}(u) = \emptyset \\
 & & \wedge\,\mathit{access}(u)' = \emptyset \\
\mathsf{in}(u,\mathsf{get\text{-}obs},o) & \equiv & \mathsf{in}_U(u,\mathsf{get\text{-}obs},o) \\
 & & \wedge\,[(\neg\mathit{local}(u,o)\ \wedge\ (\mathsf{obs},o) \in \mathit{access}(u)) \\
 & & \qquad \rightarrow\ \mathit{toupper}(u)' = (u,\mathsf{get\text{-}obs},o,\mathsf{ok})] \\
 & & \wedge\,[(\neg\mathit{local}(u,o)\ \wedge\ (\mathsf{obs},o) \notin \mathit{access}(u)) \\
 & & \qquad \rightarrow\ \mathit{tolower}(u)' = (u,\mathsf{get\text{-}obs},o)] \\
\mathsf{out}(u,\mathsf{get\text{-}obs},o,\mathsf{ok}) & \equiv & \mathsf{out}_U(u,\mathsf{get\text{-}obs},o,\mathsf{ok}) \\
 & & \wedge\,[\mathit{local}(u,o)\ \Rightarrow\ \mathit{classification}(o) \preceq \mathit{clearance}(u)] \\
 & & \wedge\,[\neg\mathit{local}(u,o)\ \Rightarrow\ \mathit{toupper}(u) = (u,\mathsf{get\text{-}obs},o,\mathsf{ok})] \\
 & & \wedge\,\mathit{access}(u)' = \mathit{access}(u) \cup \{(\mathsf{obs},o)\} \\
 & & \wedge\,\mathit{tolower}(u)' = \mathsf{nil}\ \wedge\ \mathit{toupper}(u)' = \mathsf{nil} \\
\mathsf{out}(u,\mathsf{get\text{-}obs},o,\mathsf{failed}) & \equiv & \mathsf{out}_U(u,\mathsf{get\text{-}obs},o,\mathsf{failed}) \\
 & & \wedge\,[\mathit{local}(u,o)\ \Rightarrow\ \neg(\mathit{classification}(o) \preceq \mathit{clearance}(u))] \\
 & & \wedge\,[\neg\mathit{local}(u,o)\ \Rightarrow\ \mathit{toupper}(u) = (u,\mathsf{get\text{-}obs},o,\mathsf{failed})] \\
 & & \wedge\,\mathit{tolower}(u)' = \mathsf{nil}\ \wedge\ \mathit{toupper}(u)' = \mathsf{nil} \\
\mathsf{in}(u,\mathsf{rel\text{-}obs},o) & \equiv & \mathit{status}_U(u) = \mathsf{ready}\ \wedge\ (\mathsf{obs},o) \in \mathit{access}(u) \\
 & & \wedge\,[\neg\mathit{local}(u,o)\ \rightarrow\ \mathit{tolower}(u)' = (u,\mathsf{rel\text{-}obs},o)] \\
 & & \wedge\,\mathit{status}_U(u)' = (u,\mathsf{rel\text{-}obs},o) \\
\mathsf{out}(u,\mathsf{rel\text{-}obs},o,\mathsf{ok}) & \equiv & \mathsf{out}_U(u,\mathsf{rel\text{-}obs},o,\mathsf{ok}) \\
 & & \wedge\,[\neg\mathit{local}(u,o)\ \Rightarrow\ \mathit{toupper}(u) = (u,\mathsf{rel\text{-}obs},o,\mathsf{ok})] \\
 & & \wedge\,\mathit{access}(u)' = \mathit{access}(u) - \{(\mathsf{obs},o)\} \\
\mathsf{in}(u,\mathsf{read},o) & \equiv & \mathsf{in}_U(u,\mathsf{read},o) \\
 & & \wedge\,[(\neg\mathit{local}(u,o)\ \wedge\ (\mathsf{obs},o) \in \mathit{access}(u)) \\
 & & \qquad \rightarrow\ \mathit{tolower}(u)' = (u,\mathsf{read},o)] \\
\mathsf{out}(u,\mathsf{read},o,v) & \equiv & \mathit{status}_U(u) = (u,\mathsf{read},o)\ \wedge\ (\mathsf{obs},o) \in \mathit{access}(u) \\
 & & \wedge\,[\mathit{local}(u,o)\ \Rightarrow\ v = \mathit{content}(o)] \\
 & & \wedge\,[\neg\mathit{local}(u,o)\ \Rightarrow\ (\mathit{toupper}(u) = (u,\mathsf{read},o,v)\ \wedge\ v \neq \mathsf{failed})] \\
 & & \wedge\,\mathit{tolower}(u)' = \mathsf{nil}\ \wedge\ \mathit{toupper}(u)' = \mathsf{nil} \\
 & & \wedge\,\mathit{status}_U(u)' = \mathsf{ready} \\
\mathsf{out}(u,\mathsf{read},o,\mathsf{failed}) & \equiv & \mathsf{out}_U(u,\mathsf{read},o,\mathsf{failed}) \\
 & & \wedge\,[(\mathsf{obs},o) \notin \mathit{access}(u)\ \vee\ \mathit{toupper}(u) = (u,\mathsf{read},o,\mathsf{failed})] \\
 & & \wedge\,\mathit{tolower}(u)' = \mathsf{nil}\ \wedge\ \mathit{toupper}(u)' = \mathsf{nil}
\end{array}
$$

The event formulas for write and *-alt are similar to those for read and *-obs, and are omitted for brevity.

Figure 7: Event Specifications of M

independently. As long as the interfaces remain the same and M *using* L *offers* U is satisfied, the internals of M can change.

References

[1] "Trused Computer System Evaluation Criteria," *DoD Computer Security Center*, CSC-STD-001-83, August 15, 1983.

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

[3] J.P. Anderson, "A Unification of Computer and Network Security Concepts," *Proceedings of the Symposium on Research in Security and Privacy*, pp. 77–87, 1985.

[4] D.E. Bell and L.J. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation,"

$\text{out}(u, \text{conn-req}, h)$	$\equiv$	$\text{in}_L(u, \text{conn-req}, h) \wedge \mathit{tolower}(u) = (u, *, o) \wedge h = \mathit{host}(o)$
$\text{in}(u, \text{conn-ind}, h)$	$\equiv$	$\text{out}_L(u, \text{conn-ind}, h)$
$\text{out}(u, \text{conn-res}, h, \text{ok})$	$\equiv$	$\text{in}_L(u, \text{conn-res}, h, \text{ok})$
$\text{out}(u, \text{conn-res}, h, \text{failed})$	$\equiv$	$\text{in}_L(u, \text{conn-res}, h, \text{failed})$
$\text{in}(u, \text{conn-con}, h, \text{ok})$	$\equiv$	$\text{out}_L(u, \text{conn-con}, h, \text{ok})$
$\text{in}(u, \text{conn-con}, h, \text{failed})$	$\equiv$	$\text{out}_L(u, \text{conn-con}, h, \text{failed})$
		$\wedge\ \mathit{toupper}(u)' = (u, \mathit{cmd}(\mathit{tolower}(u)), o, \text{failed})$
$\text{out}(u, \text{disconn-req}, h)$	$\equiv$	$\text{in}_L(u, \text{disconn-req}, h)$
		$\wedge\ \mathit{tolower}(u) = (u, \text{disconn-req}, D) \wedge h \in D$
		$\wedge\ [D - \{h\} \neq \emptyset \Rightarrow \mathit{tolower}(u)' = (u, \text{disconn-req}, D - \{h\})]$
		$\wedge\ [D - \{h\} = \emptyset \Rightarrow (\mathit{tolower}(u)' = \text{nil} \wedge T_L' = \text{nil})]$
		where $D \in \mathcal{H}$ is a set of hosts
$\text{in}(u, \text{disconn-ind}, h)$	$\equiv$	$\text{out}_L(u, \text{disconn-ind}, h)$
$\text{out}(u, \text{get-obs-req}, o, n)$	$\equiv$	$\text{in}_L(u, \text{get-obs-req}, o, n) \wedge \mathit{tolower}(u) = (u, \text{get-obs}, o)$
		$\wedge\ n = \mathit{clearance}(u)$
$\text{in}(u, \text{get-obs-ind}, o, n)$	$\equiv$	$\text{out}_L(u, \text{get-obs-ind}, o, n)$
$\text{out}(u, \text{get-obs-res}, o, \text{ok})$	$\equiv$	$\text{in}_L(u, \text{get-obs-res}, o, \text{ok})$
		$\wedge\ \mathit{classification}(o) \preceq n$
$\text{out}(u, \text{get-obs-res}, o, \text{failed})$	$\equiv$	$\text{in}_L(u, \text{get-obs-res}, o, \text{failed})$
		$\wedge\ \neg(\mathit{classification}(o) \preceq n)$
$\text{in}(u, \text{get-obs-con}, o, \text{ok})$	$\equiv$	$\text{out}_L(u, \text{get-obs-con}, o, \text{ok})$
		$\wedge\ \mathit{toupper}(u)' = (u, \text{get-obs}, o, \text{ok})$
$\text{in}(u, \text{get-obs-con}, o, \text{failed})$	$\equiv$	$\text{out}_L(u, \text{get-obs-con}, o, \text{failed})$
		$\wedge\ \mathit{toupper}(u)' = (u, \text{get-obs}, o, \text{failed})$
$\text{out}(u, \text{rel-obs-req}, o)$	$\equiv$	$\text{in}_L(u, \text{rel-obs-req}, o) \wedge \mathit{tolower}(u) = (u, \text{rel-obs}, o)$
$\text{in}(u, \text{rel-obs-con}, o, \text{ok})$	$\equiv$	$\text{out}_L(u, \text{rel-obs-con}, o, \text{ok})$
		$\wedge\ \mathit{toupper}(u)' = (u, \text{rel-obs}, o, \text{ok})$
$\text{out}(u, \text{read-req}, o)$	$\equiv$	$\text{in}_L(u, \text{read-req}, o) \wedge \mathit{tolower}(u) = (u, \text{read}, o)$
$\text{in}(u, \text{read-ind}, o)$	$\equiv$	$\text{out}_L(u, \text{read-req}, o)$
$\text{out}(u, \text{read-res}, o, v)$	$\equiv$	$\text{in}_L(u, \text{read-res}, o, v) \wedge v = \mathit{content}(o)$
$\text{out}(u, \text{read-res}, o, \text{failed})$	$\equiv$	$\text{in}_L(u, \text{read-res}, o, \text{failed})$
$\text{in}(u, \text{read-con}, o, v)$	$\equiv$	$\text{out}_L(u, \text{read-con}, o, v)$
		$\wedge\ \mathit{toupper}(u)' = (u, \text{read}, o, v)$
$\text{out}(u, \text{read-con}, o, \text{failed})$	$\equiv$	$\text{in}_L(u, \text{read-res}, o, \text{failed})$
		$\wedge\ \mathit{toupper}(u)' = (u, \text{read}, o, \text{failed})$

The event formulas for write-* and *-alt-* are similar to those for read-* and *-obs-*, and are omitted for brevity.

Figure 8: Event Specifications of M (cont.)

Technical Report Mitre Corporation, ESD-TR-75-306, March 1976.

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

[6] Morrie Gasser, *Building a Secure Computer System*, Van Nostrand Reinhold Company, New York, 1988.

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

[8] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason and C.E. Kahn, "A VMM Security Kernel for the VAX Architecture," *Proceedings of the Symposium on Research in Security and Privacy*, pp. 2–19, 1990.

[9] S.S. Lam and A.U. Shankar, "Protocol Verification via Projections," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 325–342, July 1984.

[10] S.S. Lam and A.U. Shankar, "A Relational Notation for State Transition Systems," *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 7, pp. 755–775, July 1990.

$$
\begin{aligned}
Inv_{M,1} &\equiv \forall\, u,o : (\mathsf{obs},o) \in access(u) \Leftrightarrow observable_U(u,o)\\
Inv_{M,2} &\equiv \forall\, u,o : (\mathsf{alt},o) \in access(u) \Leftrightarrow alterable_U(u,o)\\
Inv_{M,3} &\equiv \forall\, u,o : (\neg local(u,o) \wedge status_U(u) = \mathsf{ready}) \Rightarrow [(\mathsf{obs},o) \in access(u)) \Leftrightarrow observable_L(u,o)]\\
Inv_{M,4} &\equiv \forall\, u,o : (\neg local(u,o) \wedge status_U(u) = \mathsf{ready}) \Rightarrow [(\mathsf{alt},o) \in access(u)) \Leftrightarrow alterable_L(u,o)]\\
Inv_{M,5} &\equiv \forall\, h,u : (status_L(h,u) = (u,c,h) \wedge c \in \{\mathsf{conn\text{-}*}, \mathsf{disconn\text{-}*}\}) \Rightarrow host(u) \neq h\\
Inv_{M,6} &\equiv \forall\, h,u,o : (status_L(h,u) = (u,c,o) \wedge c \notin \{\mathsf{conn\text{-}*}, \mathsf{disconn\text{-}*}\}) \Rightarrow \neg local(u,o)\\
Inv_{M,7} &\equiv \forall\, u,o : cmd(tolower(u)) \neq \mathsf{disconn\text{-}*} \Rightarrow \neg local(sbj(tolower(u)), obj(tolower(u)))\\
Inv_{M,8} &\equiv \forall\, u,o : toupper(u) \neq \mathsf{nil} \Rightarrow \neg local(sbj(toupper(u)), obj(toupper(u)))\\
Inv_{M,9} &\equiv \forall\, u,o : (toupper(u) = (u,c,o,\mathsf{ok}) \wedge c = \mathsf{get\text{-}*})\\
&\qquad \Rightarrow (comm(host(u), host(o))\\
&\qquad\quad \wedge [c = \mathsf{get\text{-}obs} \Rightarrow classification(o) \preceq clearance(u)]\\
&\qquad\quad \wedge [c = \mathsf{get\text{-}alt} \Rightarrow clearance(u) \preceq classification(o)])\\
Inv_{M,10} &\equiv \forall\, u,o : (toupper(u) = (u, \mathsf{get\text{-}*}, o, \mathsf{ok}))\\
&\qquad \Rightarrow last(T_L) = toupper(u)\\
Inv_{M,11} &\equiv \forall\, u,o : last(T_U) = (u,c,o,\mathsf{ok})\\
&\qquad \Rightarrow ([(local(u,o) \wedge c = \mathsf{get\text{-}obs}) \Rightarrow classification(o) \preceq clearance(u)]\\
&\qquad\quad \wedge [(local(u,o) \wedge c = \mathsf{get\text{-}alt}) \Rightarrow clearance(u) \preceq classification(o)]\\
&\qquad\quad \wedge [\neg local(u,o) \Rightarrow (comm(host(u), host(o)) \wedge last(T_U) = last(T_L))])
\end{aligned}
$$

Figure 9: Conjuncts of Inv_M

[11] S.S. Lam and A.U. Shankar, "Specifying Modules to Satisfy Interfaces: A State Transition System Approach," Technical Report TR 88-30, Department of Computer Sciences, University of Texas at Austin, revised January 1991; to appear in *Distributed Computing.*

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

[13] L. Lamport, "What it means for a concurrent program to satisfy a specification: Why no one has specified priority," *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 78–83 1985.

[14] L. Lamport, "A simple approach to specifying concurrent systems," *Communications of the ACM*, Vol. 32, No. 1, January 1989.

[15] N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pp. 137–151, 1987.

[16] D. McCullough, "Specifications for Multi-Level Security and a Hook-Up Property," *Proceedings of the Symposium on Research in Security and Privacy*, pp. 161–166, 1987.

[17] D. McCullough, "Noninterference and the Composability of Security Properties," *Proceedings of the Symposium on Research in Security and Privacy*, pp. 177–186, 1988.

[18] D. McCullough, "A Hookup Theorem for Multilevel Security," *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 6, pp. 563–568, June 1990.

[19] J.K. Millen, "A Network Security Perspective," *Proceedings of National Computer Security Conference*, pp. 7–15, 1986

[20] D. Nessett, "Factors Affecting Distributed System Security," *IEEE Transactions on Software Engineering*, Vol. SE-13 No. 2, pp. 233–248, February 1987.

[21] J.M. Rushby, "The Bell and La Padula Security Model," Working Draft, June 20, 1986.

[22] V. Varadharajan, "A Multilevel Security Policy Model for Networks," *Proceedings of INFOCOM '90*, pp. 710–718, 1990.

[23] V.L. Voydock and S.T. Kent, "Security Mechanisms in High-Level Network Protocols," *Computing Surveys*, Vol. 15, No. 2, pp. 135–171, June 1983.

[24] S.T. Walker, "Network Security Overview," *Proceedings of the Symposium on Research in Security and Privacy*, pp. 62–76, 1985.