FORMAL DESCRIPTION TECHNIQUES, IV
K.R. Parker and G.A. Rose
Elsevier Science Publishers B.V. (North-Holland)

Understanding Interfaces*

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

The concept of layering has been applied to the design and implementation of computer network protocols, operating systems, and other large complex systems. However, to reap the benefits of a layered architecture—i.e., to be able to design, implement, and modify each module in a layered system individually—a composition theorem such as one we formulated and proved recently is necessary. To arrive at the theorem, we explore the semantics of interfaces. In particular, we investigate how modules should be designed to satisfy interfaces as a service provider and as a service consumer. The requirements are then presented formally, as well as our composition theorem for a general model of layered systems.

1. Introduction

Consider the design of a system to provide services through a user interface U. Instead of designing a monolithic system to provide these services, the system design may be decomposed into components that are implemented separately. For example, Figure 1 shows a system design with two modules, M and N, interacting across interface L, and with users of the system interacting with M across interface U. The intention of the design is that N provides the services of interface L (formally, *N offers L*), and M provides the services of U while utilizing the services of L (formally, *M using L offers U*).

The design in Figure 1 can be used only if the following claim can be established: M while interacting with N does indeed provide the services of U to users of the system. The above claim can be established in general by proving the following composition theorem: If M using L offers U, and N offers L, then the composite system consisting of M interacting with N offers U. To prove the theorem, we need to understand how to specify interfaces,

* The work of Simon S. Lam was supported by National Science Foundation grant no. NCR-9004464. The work of A. Udaya Shankar was supported by National Science Foundation grant no. NCR-8904590.

and how modules should be designed to satisfy interfaces as a service provider and as a service consumer. Specifically, we need formal definitions for *interface*, *M offers I*, and *M using L offers U*, where M denotes a module and I, U, L denote interfaces.

We emphasize that these formal definitions are needed not only for the composition theorem but also for practical applications, i.e., for the designer of a module to check that the module does satisfy each one of its interfaces. With the composition theorem, we are assured that each module in Figure 1 can be designed, implemented, and modified *individually*. The internals of M can change so long as M satisfies L as a service consumer and satisfies U as a service provider. Similarly, the internals of N can change so long as N satisfies L as a service provider. This we consider to be the *key benefit of decomposition*.

users

interface U

module M

interface L

module N

Figure 1. A system of two interacting modules.

Figure 1 is a simple illustration of the concept of layering (described by Dijkstra more than two decades ago [4]). Layering has been applied to the design and implementation of computer network protocols, operating systems, and other large complex systems. It is surprising that a composition theorem applicable to layered systems has not been formulated and proved. (In fact, to our knowledge, it has not even been formally stated by designers of layered systems.) Without formal semantics for the notions of *interface*, *using an interface*, and *offering an interface*, and a composition theorem based upon the semantics, we cannot get the key benefit of decomposing a system into modules or layers—because there are no applicable guidelines for designing each module to satisfy its interfaces.

The main result of this paper is a composition theorem for a general model of layered systems. Specifically, a layered system is organized as a stack of layers, with a finite number of modules in each layer. Each module offers a set of interfaces. Each module may use a set of interfaces offered by other modules, each of which resides in a lower layer of the stack. More precisely, a system can be represented by a directed acyclic graph where each node is a module, and each arc, say an arc from node M to node N, represents an interface whose service provider is N and whose service consumer is M. (Conversely, any directed acyclic graph represents a layered system in our model.)

For computer networks, we note that each module in our model represents a protocol (e.g., data link, transport, routing) rather than a protocol entity (i.e., a process). When there are several modules in a layer (e.g., the transport layer), they represent different protocols (e.g., TCP, TP4 and UDP).

The balance of this paper is organized as follows. In Section 2, we explore informally the semantics of interfaces, subsequently arriving at the concept of a "two-sided" interface. The requirements for a module to satisfy such an interface as a service consumer and for a module to satisfy it as a service provider are discussed. In Section 3 we present formal definitions. Our composition theorem is presented in Section 4. The concept of module implementation and theorems relevant to this concept are presented in Section 5.

2. Exploring Interface Semantics

A physical interface is where a module and its environment interact. For different kinds of physical interfaces, such interactions take on a variety of physical forms. For a vending machine, an interaction may be the insertion of a coin. For a workstation, an interaction may be the striking of a key on a keyboard. For a communication protocol, an interaction may be the passing of a set of parameter values. For a hardware circuit, an interaction may be the changing of voltages on certain pins.

Semantically, we model interface interactions between a module and its environment as discrete event occurrences. An interface event occurs only when both the module and environment are simultaneously executing the event (*simultaneous participation*). Such an occurrence is observable from either side of the interface. Thus an interface may be specified by a set of sequences of interface events; each such sequence defines an allowed sequence of interactions between the module and its environment. This semantic view of an interface is akin to the *specification* of a process in CCS [15], CSP [5] and Lotos [2], or the specification of an I/O automaton [14].

Let S denote the specification of a module M. Most definitions of *M satisfies S* in the literature have this informal meaning [5,6,13,14]: If every possible observation of M is described by S, then M satisfies S. (Specific definitions 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 formalism for specifying these sequences, and (4) in the conditions under which interface events can occur.)

A straightforward way to define interface semantics is to use the following paradigm: every module is viewed by an observer situated in its environment. From the viewpoint of the observer, the module is completely enclosed by a physical interface that is semantically specified by S, a set of sequences of interface events. Informally, the module satisfies its interface if and only if every possible observation of the module is described by S.

In what follows, we first illustrate this paradigm with an example. We then discuss why it is inadequate for achieving our *goal* stated in Section 1—namely, to find conditions sufficient for designing, implementing, and modifying each module in a layered system individually; in particular, each module can be designed and implemented by a different person or team. Clearly, these conditions should be as weak as possible for them to be useful in practice.

Observer as paradigm

Consider the design of a vending machine that is made up of two modules, a control module and a storage module. (See Figure 2.) The control module has the following specification (in CSP notation [5]):

$$CONT = (coin \rightarrow request \rightarrow response \rightarrow choc \rightarrow CONT)$$

The intent of the designer can be stated as follows. A customer comes up to the vending machine and inserts a coin. Having accepted the coin, the control module sends a request to the storage module. Having got the request, the storage module responds by releasing a chocolate to the control module, which then dispenses the chocolate to the outside of the vending machine. The storage module has the following specification:

$$STOR = (request \rightarrow response \rightarrow STOR)$$

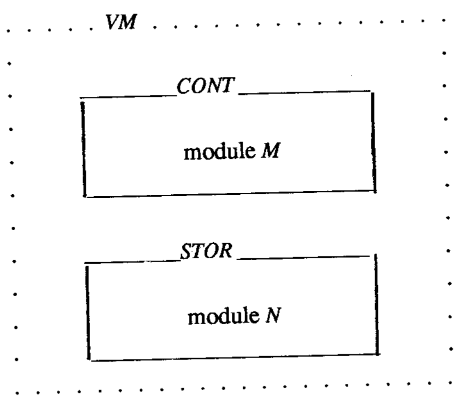

Figure 2. External views as specifications.

Let VM denote the parallel composition of $CONT$ and $STOR$ with interactions between the two modules hidden.

$$VM = (CONT \mid\mid STOR) \backslash \{request, response\}$$
$$= (coin \rightarrow choc \rightarrow VM)$$

VM represents the allowed interaction sequences between the vending machine and its environment. Note that these allowed interaction sequences (as well as those between the control and storage modules) are not explicitly specified. Instead, they are derived from *CONT* and *STOR*. (This approach of system design is characterized as *compositional* or *bottom-up*.)

Suppose we have shown that *VM* satisfies the intended property for a vending machine. Let *M* denote a module that implements *CONT*, and *N* a module that implements *STOR*. (See Figure 2.) We can then use *observational equivalence*, defined by Milner [15], to be the *satisfies* relation between a module and its specification to arrive at a composition theorem—namely, if *M* is observationally equivalent to *CONT*, and *N* is observationally equivalent to *STOR*, then we claim that the composite system consisting of *M* interacting with *N* is observationally equivalent to *VM*.

Actually, various weaker notions of equivalence can be used instead of observational equivalence. In fact, to have a useful composition theorem, the *satisfies* relation between a module *P* and its specification *S* should be much weaker. Specifically, consider an implementation relation from [2,3], stated informally as follows:

P is an implementation of *S* iff

(I1) *P* can only execute events that *S* can execute, and

(I2) *P* can only refuse events that *S* can refuse.

For the vending machine example, we claim that if *M* is an implementation of *CONT* and *N* is an implementation of *STOR*, then the composite system consisting of *M* interacting with *N* is an implementation of *VM*. However, for reasons given below, the requirements **I1** and **I2** are still too strong for achieving our goal.

Events controlled by environment

Consider module *M* in Figure 2, which implements *CONT*. Module *M* participates in the execution of four events, *coin*, *choc*, *request*, and *response*. In applying the implementation relation (or observational equivalence) to *M* and *CONT*, all four events are treated in the same way. However, there is clearly an intuitive distinction between the events {*choc*, *request*}, for which module *M* has control of, and the events {*coin*, *response*}, for which module *M* does not have control of.

Consider an occurrence of the event *coin*, requiring insertion of a coin by a customer in the environment of the vending machine, and participation by module *M* to accept the coin. Note that the *initiative* to insert a coin can only be taken by a customer in the environment; hence, the environment has control of the *coin* event.

In addition to initiative, control of an event also includes a notion of *responsibility*, e.g., the coin inserted by the customer is not a "bad" coin. The specification *CONT* above is unsatisfactory because it requires module M to have perfect discrimination of good and bad coins—a highly unreasonable premise—in the sense that in order for a module and *CONT* to satisfy the implementation relation, the module must accept only good coins and refuse all bad ones.

A more reasonable specification for the module is that it accepts only objects of a certain size, shape and weight. For such a module, a bad coin can be one of these cases:

- It is an object larger than the specified size of the coin slot. When a customer tries to insert the object, it is blocked (refused).
- It is an object that meets the size specification but not the shape or weight specification. When a customer inserts the object, it is accepted. Having accepted the object (e.g., a piece of scrap metal), module M breaks down or malfunctions in an arbitrary manner. (Can we blame the module or the designer of the module?)
- It is an object that meets the specification of size, shape and weight (e.g., a counterfeit coin). When a customer inserts it, it is accepted and module M dispenses a chocolate.

In each of the three cases, we believe that the behavior of module M is satisfactory and the module should not be considered as failing its specification. However, such would be the conclusion if the implementation relation were the criterion for M to satisfy *CONT*; hence it is too strong.

The reader, one who is familiar with CSP (or Lotos), might disagree with this conclusion. Clearly, we can replace the event *coin* by three events, *big.object*, *bad.object*, and *good.object*, with *good.object* representing both genuine and counterfeit coins. We can then rewrite the specification *CONT* as described above for the three cases. In fact, we can and should rewrite *CONT* to describe what module M must do for every possible sequence of objects that a customer may try to insert.

Indeed, CSP (or Lotos) is sufficiently expressive for specifying how a module responds to all kinds of inputs from its environment. The moral of the story here, however, is not about expressiveness, but something else, namely: In designing a module, we have *information* that certain events are controlled by the environment of the module. Such information is not utilized in the definition of the implementation relation. Consequently, unless the module's specification (i.e., *CONT* in the example) is designed to explicitly make use of this information and, moreover, all possible input sequences from the environment are accounted for in the specification, the implementation relation is too strong.

Similarly, consider the event *response* that is under the control of module N but not under the control of module M. If module M fails to dispense a chocolate because module N does not respond to a request from M, or the response of module N is bad, then module M should not be considered as failing its specification.

In Section 3 below, when we define an interface, each event is identified to be under the control of the service provider or consumer of the interface. Intuitively, we employ the following approach: for every event under the control of the consumer, it is the responsibility of the consumer, rather than the provider, to ensure that the event is not a bad input. In our definitions of *M offers I* and *M using L offers U*, we make use of information that certain events are controlled by the environment of M to arrive at "safety constraints" that are similar to, but weaker than, **I1** and **I2**. With our definitions, there is no need to *explicitly* account for all possible input sequences when specifying the interfaces of a module.

Designing module vs. testing black box

In *testing* an existing module—specifically, one whose internal states are either unobservable or too complicated to comprehend—the tester is the outside observer and the module is regarded as a black box. In designing a module to satisfy its interfaces, however, there is no need to consider the module as a black box. In fact, we do not, for the following reason.

Consider a vending machine and a tester. The tester can initiate interaction with the vending machine by inserting a coin or some object. However, it cannot initiate interaction with the vending machine in the *choc* event. Having inserted a coin, the tester can only wait to interact in the *choc* event. Suppose an indefinite duration of time has elapsed and there is no sign of chocolate. The tester cannot conclude that the vending machine has refused the *choc* event (because real time is not part of the interface semantics).

In designing a module, there is no need to view it as a black box. In our approach, the designer of a module is the one who demonstrates that the module satisfies its interface as a service provider, and the designer knows the module's internal behaviors. (See "progress constraints" in definitions of *M offers I* and *M using L offers U* in Section 3.)

Decompositional vs. compositional approach

In general, let S denote a system specification, and $\{S_i\}$ specifications of individual modules in the system. In a compositional approach, $\{S_i\}$ are specified first and S is derived from them. If S does not have the intended system property, the module specifications $\{S_i\}$ are redesigned. On the other hand, in a decompositional or top-down approach, the system specification S is first given and module specifications $\{S_i\}$ are derived from S.

When there are constraints on how a system should be decomposed into processes, as in the design of many distributed algorithms—e.g., one process in each node of a network performing a parallel computation—a compositional approach is appropriate. On the other hand, to design a system beginning with no constraint other than S, a decompositional approach provides the maximum freedom of choice on how to decompose the system.

Decomposing a system specification S into a set $\{S_i\}$ of module specifications is a difficult task in general. For a layered system, however, the task is facilitated by the hierarchical provider-consumer relationships between pairs of modules in the system. In this case, S corresponds to the "topmost" interface offered to the users of the system. Other interfaces in the system can be derived from S by a topdown approach as follows. Consider any interface U in the system. To design a module that offers the services of U, we may assume that certain services are offered by other modules through a set of interfaces $\{L_j\}$. In this manner, interfaces offered by other modules in lower layers of the system are specified.

Contract as paradigm

Our interface semantics differs in several ways from those based upon the paradigm of an external observer [2,5,14,15]. First, each module in a system is specified by a set of interfaces rather than a single external view (e.g., module M specified by interfaces U and L in Figure 1). We think of an interface to be like a legal contract between two modules in the system (e.g., interface L in Figure 1), or between a module and the environment of the system (e.g., interface U in Figure 1).

Each interface has a service provider on one side and a service consumer on the other. The allowed interaction sequences between the service provider and consumer are specified *explicitly*. Specifically, let I denote an interface between M and N. (See Figure 3.) In our design approach, I is first specified to be a set of allowed interaction sequences between M and N. Specifications of M and N are to be derived from I.

```
            interface I
                 |
                 |
   module M      |     module N
                 |
                 |
```

Figure 3. Interface I constraining behaviors of both M and N.

Note that the *same* set of interaction sequences constrains the behaviors of both M and N. This is like a legal contract between two parties: the same document contains the entire bilateral agreement, and is interpreted by each party to determine its privileges and obligations. For example, consider a loan agreement between a debtor and a creditor. The identity of either the debtor or the creditor may change (e.g., a house is sold and its mortgage assumed by the buyer). The loan agreement remains in force so long as it has been honored by its debtor and creditor, whose actual identities over time might have changed.

We refer to interface I illustrated in Figure 3 as a *two-sided interface* because, like a bilateral agreement, I encodes all information that the designers of M and N need to know and the same I is to be satisfied by both M and N—albeit the obligations of service provider and service consumer are not exactly the same.

Each event in interface I is explicitly defined to be under the control of M or N. This additional semantic information gives rise to definitions—of what it means for a service provider and a service consumer to satisfy an interface—that are adequate for our goal, i.e., design, implement and modify modules individually. (See Section 3 for details.)

The notion of control is not new (e.g., see [13]). In the theory of I/O automata [14], the events of an I/O automaton are partitioned into events under its control and events controlled by the automaton's environment. Each I/O automaton, however, is required to be input-enabled, i.e., every input event, controlled by its environment, must be enabled to occur in every state of the automaton. With this requirement, the class of interfaces that can be specified using I/O automata is restricted. For example, a module with a finite input buffer such that inputs causing overflow are refused cannot be specified. A consequence of the restriction is that it is not always possible to use an I/O automaton to encode all the semantic information that a designer wants to include in a specification, e.g., the input buffer size. Such information has to be supplied separately by other means. (The theory of I/O automata differs from ours in other ways also. For example, the specification of an I/O automaton is defined to be its external view as seen by an outside observer; specifically, the *satisfies* relation is the usual one, i.e., an automaton M satisfies its specification S if every possible observation of M is described by S.)

Obligations of service provider and consumer

Consider Figure 3. Since interface events are partitioned into events under the control of M and events under the control of N, in general interface I can be satisfied only if M and N cooperate with each other in some manner. In order to design each module individually, terms of the required cooperation must be completely encoded in I.

For illustration, we consider some special cases, i.e., the terms of cooperation are in the form of a set of guarantees a module must ensure given that the other module satisfies a set of assumptions, where assumptions and guarantees are assertions of safety or progress. (For this section, assumptions and guarantees are stated informally and only very simple ones are illustrated. See Part II of our report [11] for a general and more rigorous presentation of safety and progress assertions in our method.)

A safety assertion is a statement that something bad never occurs. An example of some safety assumptions and guarantees for M and N is shown below.

(S1) M never executes $e_1 \Rightarrow N$ never executes e_2

(**S2**) N never executes $e_2 \Rightarrow M$ never executes e_1

(The consequent of **S1** is a guarantee of N given an assumption about M, which is the antecedent of **S1**. Similarly, the consequent of **S2** is a guarantee of M given an assumption about N, which is the antecedent of **S2**.)

A progress assertion is a statement that something good eventually occurs. An example of some progress assumptions and guarantees for M and N is shown below.

(**P1**) M eventually executes $e_3 \Rightarrow N$ eventually executes e_4

(**P2**) N eventually executes $e_4 \Rightarrow M$ eventually executes e_3

Suppose M and N are designed individually and it has been proved that N satisfies **S1** and **P1** and M satisfies **S2** and **P2**. To infer that the composite system of M and N satisfies the guarantees—more generally, to prove a composition theorem—we must take care that circular reasoning is not used. The possibility of circular reasoning in composing processes has been addressed by other researchers. For processes that communicate by CSP primitives, Misra and Chandy gave a proof rule for assumptions and guarantees that are restricted to safety properties [16]. Using different models, Pnueli [17] presented a proof rule and Abadi and Lamport [1] presented a composition principle that are more general in that the class of assertions includes progress properties (albeit the class is still restricted).

In summary, we know the following: Safety assumptions and guarantees can be composed without circular reasoning. (For **S1** and **S2**, this is intuitively evident.) But with progress assumptions and guarantees, such as **P1** and **P2**, circular reasoning is involved.

In formulating our composition theorem below, circular reasoning is avoided in a straightforward manner. Specifically, each interface in our model is between a service provider and consumer. Therefore, we need only assert that the provider eventually performs a service given that the consumer eventually does something good. (E.g., for a vending machine, if eventually a customer inserts a coin, then the vending machine eventually dispenses a chocolate.) Thus, if N is the service provider and M the service consumer of interface I in Figure 3, only **P1** is meaningful (but not **P2**). Since our composition theorem applies to layered systems that are modeled by a set of modules organized as the nodes of a directed acyclic graph, circular reasoning is avoided.

Our *implements* relation

In the next section, we formally define *M offers I* and *M using L offer U*, where M denotes a module and I, U and L interfaces. These definitions embody our semantics for a module satisfying an interface as a service provider and as a service consumer. Each module in a system can be designed separately given all of the interfaces offered and used by the module, However, having derived a module, say M_1, that satisfies all of the given interfaces, it is useful to have an *implements* relation to facilitate additional refinements of M_1 in the manner described below.

Suppose M_1 has been designed such that M_1 offers I and M_1 using L offers U for arbitrary interfaces I, U and L. Suppose M_2 is derived from M_1 by a series of refinements. The *implements* relation should be defined such that it is as weak as possible and allows the following to be inferred: If M_2 implements M_1, then M_2 offers I and M_2 using L offers U.

Consider Figure 3. Having derived modules M_1 and N_1 that cooperate to satisfy I, our *implements* relation is then used in the same way as the implementation relation [2,3] described above. It is however a weaker relation because interface events are under the control of either the service provider or consumer. Its definition, given in Section 5 below, is similar to that of *M offers I*.

3. Definitions

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.

By definition, 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 service consumer (user) of I. For 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 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. Let σ be a sequence over a set F. For any set E, *image* (σ, E) is the sequence over E obtained from σ by deleting all elements 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.

By definition, a module has control of its internal and output events, but its input events are under the control of its environment.

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 said to be disabled in 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 an interface I. Let σ be a sequence over a set of states and events.

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 a finite behavior σ, and @ to denote concatenation of two sequences. (For sequences consisting of a single element, say e, the sequence notation $<e>$ is abbreviated to e for simplicity.)

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

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

  $\forall e \in Inputs(M)$: $\sigma@e$ 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.

Note that module M is required to satisfy interface I only if its environment satisfies the safety requirements of I. Specifically, for any finite behavior that is not safe wrt I, the two Safety constraints are satisfied trivially; for any allowed behavior of M that is not safe wrt I, the Progress constraint is satisfied trivially. That is, as soon as the environment of M violates some safety requirement of I, module M can behave arbitrarily and still satisfy the definition of M *offers* I.

The two Safety constraints can be stated informally as follows: First, whenever an output event of M is enabled to occur, the event's occurrence would be safe, i.e., if the event occurs next, the resulting sequence of interface event occurrences is a prefix of an allowed event sequence of I. Second, 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 of M whose occurrence would be unsafe, module M has a choice: it may block the event's occurrence or let it occur.

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. 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
  $\forall e \in Outputs(M)$: $last(\sigma) \in enabled_M(e) \Rightarrow \sigma@e$ is safe wrt U and L, and
  $\forall e \in Inputs(M)$: $\sigma@e$ is safe wrt U 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
  σ is allowed wrt $L \Rightarrow \sigma$ is allowed wrt U.

The definition of *M using L offers U* is similar to the definition of *M offers I* in most respects. The main difference between the two definitions 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.

Note that *M using L offers U* reduces to *M offers U* when L is a *null* interface—i.e., $Events(L)$ is empty, and $AllowedEventSeqs(L)$ has the null sequence <> as its only element.

4. Composition Theorem

We first define how modules are composed.

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

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

Convention. For any set of modules with distinct names, $\{M_j : j \in J\}$, it is assumed that $Internals(M_j) \cap Events(M_k) = \varnothing$, for all $j, k \in J, j \neq k$.

The above convention can be ensured by, for instance, including the name of each module as part of the name of each of its internal events. Thus to check that a set of modules $\{M_j : j \in J\}$ is compatible, it suffices to check that their output event sets are pairwise disjoint.

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.

(Note that the ordering of module states in the tuple is arbitrary. In fact, the state of the composite system can be represented by an unordered tuple provided that, for all $i, j \in J$, $States(M_i) \cap States(M_j) = \varnothing$. This requirement can be ensured by including the name of each module as part of its state.)

Definition. Given a compatible set of modules $\{M_j: j \in J\}$, their composition 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* $M_j]$.

Definition. A set of interfaces $\{I_j: j \in J\}$ is *disjoint* iff $\forall j, k \in J, j \neq k$,

$Events(I_j) \cap Events(I_k) = \varnothing$.

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

- M using L offers U
- N offers L

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

Since the composition of any two compatible modules is also a module, Theorem 1 is easily extended to the following theorem for an arbitrary number of modules organized in a linear hierarchy.

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:

- $I_1, I_2, \cdots$, and I_n are disjoint interfaces.
- 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.

Theorem 2 can be used for the design and specification of layered systems by considering each system layer as a module in our theory. For some complex systems, however, it is desirable to consider each system layer as a set of modules. For example, the transport layer of a computer network may consist of a set of different transport protocols (TCP, TP4, UDP, etc.).

We next formulate and prove a composition theorem for a general model of layered systems.

Definition. The composition of a set of disjoint interfaces, $\{I_j : j \in J\}$, is an interface I defined by:

- $Events(I)$ that is the union of

  $Inputs(I) = \bigcup_{j \in J} Inputs(I_j)$, and

  $Outputs(I) = \bigcup_{j \in J} Outputs(I_j)$

- $AllowedEventSeqs(I) = \{w : w$ is a sequence over $Events(I)$ such that $\forall j \in J : image(w, Events(I_j)) \in AllowedEventSeqs(I_j)\}$

Definition. Given a set $\{U_1, U_2, \cdots, U_n, L_1, L_2, \cdots, L_m\}$ of disjoint interfaces, M *using* $L_1, L_2, \cdots, L_m$ *offers* $U_1, U_2, \cdots, U_n$ iff M using the composition of $\{L_1, L_2, \cdots, L_m\}$ offers the composition of $\{U_1, U_2, \cdots, U_n\}$. Also M *offers* $U_1, U_2, \cdots, U_n$ iff M offers the composition of $\{U_1, U_2, \cdots, U_n\}$.

Before considering a layered architecture in general, we first prove the following *basic composition theorem*:

Theorem 3. Let modules, M and N, and disjoint interfaces $\{U, L, V\}$, satisfy the following:

- M using L offers U
- N offers L, V

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

Note that Theorem 3 subsumes Theorem 1. Specifically, it reduces to Theorem 1 when V is a null interface. A proof of Theorem 3 is presented in [10]; it is quite long, requiring seven lemmas.

Definition. A layered system with layers 1 through J is defined by

- *Modules*, a set of modules with distinct names partitioned into sets $Modules(j)$, $j=1, \cdots, J$, one for each layer.
- *Interfaces*, a set of disjoint interfaces partitioned into sets $Interfaces(j)$, $j=1, \cdots, J$, one for each layer.
- for each module $M \in Modules$, $U(M)$, a set of interfaces to be offered by M, and $L(M)$, a set of interfaces to be used by M.

such that the following Naming constraints are satisfied:

(1) for all $j=1, \cdots, J$:

$$Interfaces(j) = \bigcup_{M \in Modules(j)} U(M)$$

(2) for every $M \in Modules$:

(a) $M \in Modules(j) \wedge j>1 \Rightarrow L(M) \subseteq \bigcup_{k<j} Interfaces(k)$

(b) $Inputs(M) = [\bigcup_{I \in U(M)} Inputs(I)] \cup [\bigcup_{I \in L(M)} Outputs(I)]$

(c) $Outputs(M) = [\bigcup_{I \in U(M)} Outputs(I)] \cup [\bigcup_{I \in L(M)} Inputs(I)]$

(3) for every pair of distinct modules M and N:

$$U(M) \cap U(N) = \varnothing$$
$$L(M) \cap L(N) = \varnothing$$

The above Naming constraints ensure that *Modules* is a compatible set of modules.

In our model of layered systems, a module in layer j can use an interface offered by any module in a lower layer, provided that no other module is using the same interface. (This provision is simply a naming constraint. In fact, a module can offer services to multiple users concurrently. But by tagging interface event names with user names, the interface offered to each user is distinct.) A layered system corresponds to a directed graph whose nodes are modules and whose arcs are defined as follows: for modules M and N in *Modules*, there is an arc from M to N iff for some interface I in *Interfaces*, N offers I and M uses I. It is not hard to see that every layered system in our model can be represented by a directed acyclic graph. Furthermore, every directed acyclic graph represents a layered system allowed by our model.

Let $Services(j)$ denote the services available to the user(s) of layer j. Formally,

$$Services(1) = Interfaces(1)$$

and for $j > 1$

$$Services(j) = [Interfaces(j)] \cup [Services(j-1) - \bigcup_{M \in Modules(j)} L(M)]$$

Theorem 4. For a layered system, if the following hold:

- $\forall M \in Modules(1)$: M offers $U(M)$
- for $j=2, \cdots, J$, $\forall M \in Modules(j)$: M using $L(M)$ offers $U(M)$

Then, $\bigcup_{k \in \{1, \cdots, J\}} Modules(k)$ is a set of compatible modules and their composition offers $Services(J)$.

5. Implementation Theorems

To define our *implements* relation between two modules, we extend the definitions of "safe wrt" and "allowed wrt" as follows. Let M and N denote modules, and let σ be a sequence over a set of states and events.

Definition. σ is safe wrt N iff for some $w \in Behaviors(N)$,

$$image(w, Inputs(N) \cup Outputs(N)) = image(\sigma, Inputs(N) \cup Outputs(N)).$$

Definition. σ is allowed wrt N iff for some $w \in AllowedBehaviors(N)$,

$$image(w, Inputs(N) \cup Outputs(N)) = image(\sigma, Inputs(N) \cup Outputs(N)).$$

Definition. Given modules M and N, M *implements* N iff the following conditions hold:

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

  $\forall e \in Outputs(M)$: $last(\sigma) \in enabled_M(e) \Rightarrow \sigma@e$ is safe wrt N, and

  $\forall e \in Inputs(M)$: $\sigma@e$ is safe wrt $N \Rightarrow last(\sigma) \in enabled_M(e)$.
- Progress constraints:
  For all $\sigma \in AllowedBehaviors(M)$, if σ is safe wrt N, then σ is allowed wrt N.

Suppose a module has been designed and shown to satisfy a set of interfaces. Subsequently, we may want to refine it to derive new modules. The following theorems are useful for justifying such refinement steps.

Theorem 5. Let M and N be modules and I an interface. If M implements N and N offers I, then M offers I.

Theorem 6. Let M and N be modules, and U and L be interfaces. If M implements N and N using L offers U, then M using L offers U.

Theorem 7. Let M_1, M_2 and M_3 be modules. If M_3 implements M_2 and M_2 implements M_1, then M_3 implements M_1.

6. Concluding Remarks

Proofs of the theorems and lemmas in this paper are presented in [10]. For interfaces and modules specified in the relational notation [8], we have developed a proof method based upon the theory in this paper [11]. A small example illustrating application of our method to the specification of a connection management protocol can be found in [9]. Nontrivial applications of our method to the specification and verification of protocols for concurrency control and secure access control can be found in [7] and [12] respectively.

Acknowledgement

We thank Michael Merritt of Bell Laboratories for his constructive criticisms of our proof method in [7], which motivated us to develop the theory presented in this paper.

References

[1] 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.

[2] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems*, Vol. 14, 1987.

[3] S. D. Brookes, C. A. R. Hoare, and A. D. Roscoe, "A Theory of Communicating Sequential Processes," *JACM*, Vol. 31, No. 3, 1984.

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

[7] 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, August 1988; revised, January 1991, to appear in *Distributed Computing*.

[8] 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; 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.

[9] S. S. Lam and A. U. Shankar, "A Composition Theorem for Layered Systems," *Proceedings 11th Int. Symp. on Protocol Specification, Testing and Verification*, Stockholm, June 1991.

[10] S. S. Lam and A. U. Shankar, "A Theory of Interfaces and Modules I—Composition Theorem," Technical Report, Department of Computer Sciences, University of Texas at Austin, in preparation.

[11] S. S. Lam and A. U. Shankar, "A Theory of Interfaces and Modules II—Proof Method," Technical Report, Department of Computer Sciences, University of Texas at Austin, in preparation.

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

[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] R. Milner, *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag, Berlin, 1980.

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

[17] 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, 1984.