A Theory of Interfaces and Modules I—Composition Theorem

Simon S. Lam, *Fellow, IEEE*, and A. Udaya Shankar, *Member, IEEE*

***Abstract*—We model a system as a directed acyclic graph where nodes represent modules and arcs represent interfaces. At the heart of our theory is a definition of what it means for a module to satisfy a set of interfaces as a service provider for some and as a service consumer for others. Our definition of interface satisfaction is designed to be *separable*; i.e., interfaces encode adequate information such that each module in a system can be designed and verified separately, and *composable*; i.e., we have proved a composition theorem for the system model in general.**

***Index Terms*—Specification, verification, interfaces, modules, composition, decomposition, composition theorem, separable design.**

I. Introduction

CONSIDER the design of a system to provide services to users. Suppose the system is to be constructed as a collection of interacting modules. Our system model is a directed acyclic graph where nodes are modules. Each module is a service provider and may also be a service consumer. Each arc in the graph, say, from module M to module N, represents an interface through which M uses services provided by N. Additional interfaces, called *system interfaces*, through which modules provide services to users of the system are specified.

The concepts of interfaces and modules are well known. To construct any large system, it is desirable that different modules in the system can be designed and implemented separately by different persons (teams). To maintain the system, it is desirable that module implementations can be modified whenever technology changes or better solutions are found.

Our concepts of service providers and consumers are motivated by communication network protocols. For example, a routing protocol provides the service of unreliable end-to-end packet delivery to a transport protocol. Using this service, the transport protocol offers a reliable end-to-end byte delivery service to various applications.

Of interest in this paper is the development of a formal method to support system design such as described above. At the heart of every formal method is a definition of what it means for a module (or system) to satisfy its specification. In our model, a module is specified by its interfaces. The main contribution of this paper is a definition of what it means for a module to satisfy its interfaces as a service provider for some interfaces and as a service consumer for others. Our definition of interface satisfaction has two properties.

- *Separable*—The modules in a system can be designed and verified separately. That is, the designer of a module can demonstrate that the module satisfies its interfaces without knowledge of the module's environment (in particular, without knowing how other modules in the system are implemented).
- *Composable*—The proof obligations are to show that each module in a system satisfies the module's interfaces. When modules are composed to form the system, a composition theorem (to be presented) guarantees that the collection of interacting modules satisfies the system interfaces.

Although composability is a well-known property, our notion of separability has not been discussed in the literature. (We coined the term *separable* when we wrote this paper.)

We found that to formulate a satisfaction definition that is both separable and composable, there are conflicting demands. In particular, proving a composition theorem is facilitated by a "strong" notion of satisfaction. But the notion may be so strong that separable design of individual modules is impossible or very difficult to do. In Sections II and IV, we present an example to illustrate why a generally accepted notion of satisfaction [3], [15], [16] is too strong, making it difficult to achieve separability.

We found that the notion of satisfaction can be—and should be—weakened by making use of information that for each module, some interface events are not under its control. Such weakening gives rise to a definition of satisfaction that is both separable and composable. Weakening the notion of satisfaction, however, makes it much more difficult to prove the desired composition theorem, but it has been proved. (See Section V and the Appendixes at the end of this paper.)

Most concurrency theories, such as CSP [5], CCS [16], and I/O automata [15], have been designed for composing processes that interact as *peers*; i.e., there is no distinction between service provider and consumer, and any process in a system can interact with any other process. In this regard, our system model is less general. However, such generality is not important in a large number of application domains of interest to us. In particular, the development

Manuscript received February 1992; revised September 1993. A portion of this paper was presented as an invited address at the Fourth International Conference on Formal Description Techniques (FORTE), Sydney, Australia, November 1991. The work of S. S. Lam was supported by National Science Foundation under Grant NCR-9004464. The work of A. Udaya Shankar was supported by National Science Foundation under Grant NCR-8904590. Recommended by M. Zelkowitz.

S. S. Lam is with the Department of Computer Sciences, University of Texas, Austin, TX 78712.

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

IEEE Log Number 9214485.

of our theory was originally motivated by layered systems, e.g., communication network protocols. Layering, such as that described by Dijkstra [4] more than two decades ago, has been applied to the design and implementation of not only communication network protocols but also operating systems and other large complex systems. It is surprising that a formal method tailored to the needs of layered systems has not been formulated and that an appropriate composition theorem has not been proved.

In applying our theory, a module in our system model should be interpreted broadly as a subsystem (rather than narrowly as a process). For example, in modeling the protocols of a communication network, a module would represent a protocol (e.g., data link, transport, routing) that is composed of a set of peer processes. It is the entire set of peer processes that is the consumer and provider of services. Note that a protocol layer of a communication network is generally made up of a set of protocols. For example, in the transport layer, there can be different protocols (e.g., TCP, TP4, UDP), each of which would be represented by a distinct module in our system model.

The balance of this paper is organized as follows. In Section II, we explore informally the semantics of interfaces using a vending machine example, and motivate the key concepts of our theory. In Section III, we present our definition of what it means for a module to satisfy interfaces as a service consumer and as a service provider. In Section IV, we revisit the vending machine example, and use it to illustrate concepts and ideas introduced in the first three sections as well as how our notion of satisfaction differs from the generally accepted notion. Our composition theorem is presented in Section V. The concept of module implementation and theorems relevant to this concept are presented in Section VI. Proofs of our theorems and lemmas are presented in four appendixes.

In this paper, we are concerned only with semantics—in particular, semantics of the service offered by a provider to a consumer across an interface. Upon this semantic foundation, different languages for specifying interfaces and modules, as well as proof methods, may be developed. In a companion paper [12], we present one such proof method for interfaces and modules specified in the relational notation [9]. Applications of our theory and method can be found in [8] for concurrency control protocols, in [13] for access control protocols, and in [19] for communication protocols.

II. Exploring Interface Semantics

A physical interface occurs 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. An interface is 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 CSP [5], CCS [16], and LOTOS [2], or the specification of an I/O automaton [15].

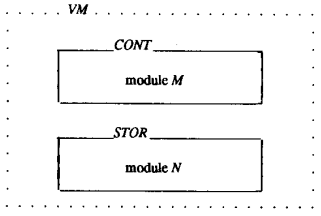


Fig. 1. External views as specifications.

Let S denote the specification of a module M. A widely accepted notion of satisfaction is the following [5, p. 59]: *Every possible observation of the behavior of M is described by S.* Many definitions of *M satisfies S* in the literature are based upon this notion (see [3], [7], [14], [15] for examples). But due to the use of different models, 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 condition under which interface events can occur.

Using this widely accepted notion of satisfaction, a straightforward way to define satisfaction for our model is to adopt the paradigm of an external observer: Every module is viewed by an observer situated in its environment. From the viewpoint of the observer, the module is enclosed by a physical interface, which is semantically specified by S, a set of sequences of interface events.

In what follows, we first illustrate this paradigm with an example. We then discuss why the property of separability is difficult to achieve with this paradigm.

A. 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 Fig. 1.) 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:

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

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

$$\begin{aligned} VM &= (CONT \,\|\, STOR)\backslash\{request, response\} \\ &= (coin \rightarrow choc \rightarrow VM). \end{aligned}$$

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 the module specifications, $CONT$ and $STOR$, which have been specified first. (This approach of system design is said to be *compositional*.)

Suppose 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 Fig. 1.) At this point, not only do we need a *satisfies* relation between a module and its specification, but we also desire a composition theorem that allows the use of local reasoning only; that is, if we show that M satisfies $CONT$ and that N satisfies $STOR$, the theorem guarantees that the composite system, consisting of M interacting with N, satisfies VM.

A candidate for the *satisfies* relation is observational equivalence [16], which has been proposed for use with LOTOS specifications of computer network protocols. However, it is obvious that observational equivalence, being a symmetric relation, is much stronger than what we need. (We need only show that a module satisfies its specification, but not vice versa.) A better candidate would be an implementation relation from the theory of CSP [3], stated informally as follows [2]:

P is an implementation of S iff

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

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

where P denotes a module and S its specification. For reasons given below, however, the conditions **I1** and **I2** are still stronger than what they should be for our purposes.

B. Events Controlled by Environment

Consider module M in Fig. 1, which implements *CONT*. Module M participates in the execution of four events, *coin, choc, request*, and *response*. In applying the implementation relation to M and *CONT*, all four events are treated in the same way. However, there is clearly an intuitive distinction between the events $\{choc, request\}$, of which module M has control, and the events $\{coin, response\}$, of which module M does not have control.

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 be taken only 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 must be a genuine coin. For the specification *CONT* given, the implementation relation is unsatisfactory, because it requires module M to have perfect discrimination of coins. Specifically, for a module and *CONT* to satisfy the implementation relation, the module must accept only coins and refuse anything that is not a (genuine) coin.

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 if the response of module N is something other than a chocolate, then module M should not be considered as failing its specification.

For this example, the implementation relation is so strong that it is not practical to design a module M such that M and *CONT* satisfy the implementation relation, unless the designer of module M has knowledge that vending machine customers will $always$ insert genuine coins and that module N will $always$ release a chocolate when requested. But such knowledge means that the design of M is *not separable*.

C. How to Achieve Separability

We have $information$ that the events $\{coin, response\}$ are not under the control of module M. Such information, however, is not used in the definition of the implementation relation or, as far as we know, in the definition of any other *satisfies* relation for models in which interfaces are specified by event traces.[1]

To achieve separability for the implementation relation, the specification *CONT* must be rewritten to include such information. This may be carried out as follows. Add two events to the interface between module M and vending machine customers: *large*, representing any object larger than the size of a coin, and *small*, representing any object smaller than or equal to the size of a coin. The set $\{coin, large, small\}$ represents the universe of all possible inputs from vending machine customers. Similarly, add an event *error* to the interface between modules M and N such that the set $\{response, error\}$ represents the universe of all possible inputs coming from module N. Rewrite *CONT* such that the responses of module M to *all* possible input sequences from its environment are fully specified—a nontrivial task.

Clearly, CSP is sufficiently expressive for specifying how a module responds to all possible sequences 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 interface events are controlled by the module's environment. Rather than putting this information into interface specifications, we make use of this information in our definition of interface satisfaction.[2] The benefit is that we need specify only $intended$ sequences of interface interactions, and, as will be shown later, our definition of interface satisfaction is composable as well as separable.

D. Decomposing a System Specification

Conceptually, to design a system that is a collection of interacting modules, there are two basic approaches. Let S denote a system specification, and let $\{S_i\}$ denote specifications of individual modules in the system. In a compositional (bottom-up) approach, module specifications $\{S_i\}$ are specified first, and S is derived from them. If S does not have the intended system

[1] The work of Kay and Reed [6] is an exception that recently came to our attention. The goal of their Rely and Guarantee method is similar to ours, but the method is being developed for a different model, namely, Timed CSP.

[2] See *safety constraints* in our definitions of *M offers I* and *M using L offers U* in Section III. These constraints are similar to, but weaker than, **I1** and **I2**.

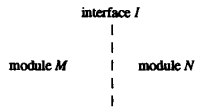


Fig. 2. Interface I constraining behaviors of both M and N.

properties, the module specifications $\{S_i\}$ are redesigned. This procedure is repeated until S has the intended properties. In a decompositional (top-down) approach, the system specification S with the intended properties is given first, and module specifications $\{S_i\}$ are to be derived from S.

Global reasoning is needed to derive S from $\{S_i\}$ in the compositional approach, and to derive $\{S_i\}$ from S in the decompositional approach. Decomposing a system specification S into a set $\{S_i\}$ of module specifications is facilitated for a directed acyclic graph model where modules interact as service providers and consumers, rather than as peers. For this model, S corresponds to a composition of the system interfaces offered to users of the system. Other interfaces in the system can be derived from S by a top-down approach as follows. Consider any interface U in the system. To design the subsystem that provides the services of U, we may assume that certain services are provided through a set of interfaces $\{L_j\}$. That is, the subsystem is decomposed into a module that uses the services of $\{L_j\}$ to provide the services of U, and a set of modules that provide the services of $\{L_j\}$.

E. Contract as Paradigm

Our interfaces differ in several ways from module specifications based upon the paradigm of an external observer [2], [5], [15], [16]. In our model, each module in a system is specified by a set of interfaces that it uses and by a set of interfaces it offers, rather than by a single external view. Each interface is like a legal contract between two parties, i.e., between two modules or between a module and the system environment.

In our design approach, the service of an interface is specified first. More specifically, consider Fig. 2. The set of allowed interaction sequences representing interface I is specified first. Specifications of modules M and N are to be derived from I. (This conforms to the decompositional approach discussed above.)

Note that the *same* set of interaction sequences representing I 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 is assumed by the buyer.) The loan agreement remains in force for as long as it is honored by its debtor and creditor, whose actual identities over time might have changed.

We refer to interface I, illustrated in Fig. 2, 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 that the obligations of service provider and service consumer are not the same. Each event in interface I is specified to be under the control of M or N. We make use of this information to define what it means for a service provider and a service consumer to satisfy an interface. (See Section III.)

The notion of control is not new. (See [14], [15].) For example, in the theory of I/O automata [15], the events of each automaton are partitioned into events under its control and events controlled by the automaton's environment. However, this information is not used in defining its *satisfies* relation.[3] Instead, each automaton is required to be input-enabled; i.e., every input event, controlled by its environment, is enabled to occur in every state of the automaton. With this requirement, the responses of an automaton to all possible inputs must be specified for every state of the automaton (either explicitly or implicitly). Moreover, the class of interfaces that can be specified by I/O automata is restricted. For example, a module with a finite input buffer such that inputs causing overflow are refused cannot be specified.

F. Obligations of Service Provider and Consumer

Consider Fig. 2. 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 separately, 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 that 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 [12] 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. Examples of some safety assumptions and guarantees for M and N are 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. Examples of some progress assumptions and guarantees for M and N are 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 that 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

[3] The specification of an I/O automaton is defined to be its external view, and the *satisfies* relation is the usual one for external views.

Chandy gave a proof rule for assumptions and guarantees that are restricted to safety properties [17]. Using different models, Pnueli [18] presented a proof rule and Abadi and Lamport [1] presented a composition principle, both of which are more general in that the class of assertions includes progress properties (albeit that 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.

We define our notion of interface satisfaction such that circular reasoning is avoided in a straightforward manner. Specifically, each interface in our model is between a service provider and consumer. Therefore, we need assert only 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 is the service consumer of interface I in Fig. 2, only **P1** is meaningful (but **P2** is not). Since our composition theorem applies to systems that are modeled by a set of modules organized as the nodes of a directed acyclic graph, circular reasoning is avoided.

G. Our implements Relation

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

Suppose that M_1 has been designed such that M_1 offers I and M_1 using L offers U for some interfaces I, U and L. Subsequently, 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 yet allows the following to be inferred: If M_2 implements M_1, then (i) M_2 offers I and (ii) M_2 using L offers U.

Consider Fig. 2. 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 a weaker relation, however, because its definition, given in Section V below, is similar to that of *M offers I*.

III. Definition of Interface Satisfaction

We first define some notation for sequences. A *sequence over E*, where E is a set, means a finite or an 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 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) to be the following set:

$$\{w : w \text{ is a finite prefix of an allowed event sequence of } I\},$$

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 in 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), define $enabled_A(e)$ to be the following set of states:

$$\{s : \text{for some state } t, (s, t) \in \textit{Transitions}_A(e)\}.$$

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*$_A(e)$.

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 $enabled_M(e)$.

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.

Notation: Let σ be a sequence over a set F. For any set E, $proj(\sigma, E)$ is the sequence over E obtained from σ by deleting all elements that are not in E.

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:

$$proj(\sigma, Events(I)) \in AllowedEventSeqs(I).$$

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

- σ is finite and $proj(\sigma, Events(I)) \in SafeEventSeqs(I)$;
- 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.

Consider next a module M that offers interface U and uses interface L. 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:
  $Events(U) \cap Events(L) = \emptyset$,
  $Inputs(M) = Inputs(U) \cup Outputs(L)$, 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.

IV. Vending Machine Example

To illustrate our definition of interface satisfaction, we return to the vending machine example introduced in Section II.

Notation: For a string α and positive integer n, α^n denotes the string $\alpha@\alpha@\cdots@\alpha$ where α appears n times. α^0 denotes the empty string <>. α^∞ denotes the string $\alpha@\alpha@\cdots$ where α appears infinitely often.

The user interface of the vending machine, denoted by U, has the following events,

+*coin* insertion of coin by customer
+*large* insertion of object larger than coin by customer
+*small* insertion of object smaller than or same size as coin by customer
−*choc* dispensing of chocolate by machine
−*coin* return of coin by machine
−*small* return of small object by machine

where the character + is used in names of events controlled by the environment, and the character − is used in names of events controlled by the machine. Specifically, we have

$$Inputs(U) = \{+coin, +small, +large\}$$
$$Outputs(U) = \{-choc, -coin, -small\}.$$

Aside from this distinction of events, the specification of U is the same as VM in Section II. See the equation below:

$$AllowedEventSeqs(U) = \{a^n : a = (+coin, -choc), \text{ and } n \text{ is 0, a positive integer or } \infty\}.$$

Note that only the intended interface interaction is specified, namely, if someone inserts a coin, then the machine will eventually dispense a chocolate.

We now design a control module, called M, that offers the service of interface U while using the service of a storage module through interface L. The events of interface L are as follows:

$-request$ signal from module M to storage module
$+response$ release of a chocolate from storage module to module M
$+error$ signal from storage module to module M

where the character $+$ is used in names of events controlled by the storage module, and the character $-$ is used in names of events controlled by module M. Specifically, we have

$$Inputs(L) = \{-request\}$$
$$Outputs(L) = \{+response, +error\}.$$

Aside from this distinction of events, the specification of L is the same as *STOR* in Section II:

$$AllowedEventSeqs(L) = \{b^n : b = (-request, +response) \text{ and } n \text{ is 0, a positive integer or } \infty\}.$$

Again, only the intended interface interaction is specified, namely, if module M makes a request, then the storage module will eventually respond with a chocolate.

A. Designing a Control Module

Numerous designs of module M can be specified such that *M using L offers U* is satisfied. Fig. 3 shows one specified as a communicating finite state machine. Its input events are $+coin, +small, +large, +response$, and $+error$. Its output events are $-choc, -coin, -small$, and $-request$. It has one internal event (not named) whose occurrence is associated with a state transition from ILLEGAL to HALT. Its fairness requirements are $\{-choc\}$ and $\{-request\}$. Note that the other output events, $-coin$ and $-small$, and the internal event are not required to be fairly scheduled. Note also that the events $+large, +small, +error, -coin$ and $-small$ do not appear in the allowed event sequences of U and L. Thus, any occurrence of these events would be unsafe.

Let us examine the behavior of module M as specifed in Fig. 3. Note that there is no state transition associated with the event $+large$, which means that the module must always block the insertion of large objects by customers.

When the module is in state READY, both $+coin$ and $+small$ are enabled to occur. Suppose that a customer inserts a small object that is not a coin. Having accepted the object, the module may behave in different ways. In particular, the small object may be a counterfeit coin, say, a well-made one.

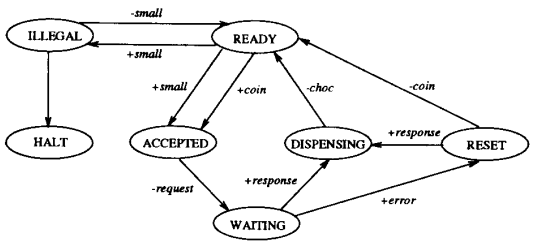


Fig. 3. A control module.

In this case, the module may be fooled into dispensing a chocolate. Alternatively, the module may detect that the object accepted is not a genuine coin (state transition from READY to ILLEGAL). Subsequently, it may either return the object or halt.

Note that module M is designed to behave correctly ($-choc$ following $+coin$) if the storage module behaves correctly ($+response$ following $-request$). Moreover, it is possible that module M behaves correctly ($-choc$ following $+coin$) after the storage module has behaved incorrectly ($+response$ following $+error$ following $-request$).

The behavior of module M as specified in Fig. 3 appears to be reasonable. However, it would not satisfy interfaces U and L if we were to adopt the usual notion of satisfaction in [3], [5], [15], [16].[4] In particular, module M has observable behaviors at interfaces U and L that are not in $AllowedEventSeqs(U)$ and $AllowedEventSeqs(L)$, respectively, unless the following is known to be true: Vending machine customers will never insert small objects, and the storage module will always respond correctly. But such knowledge implies that the design of module M is not separable.

On the other hand, module M as specified above satisfies *M using L offers U*. (A proof is given below.) In particular, the interfaces U and L encode all information that a designer needs to specify a correct module M. That is, the design of M is separable.

There are many ways to modify the specification of M in Fig. 3 to make it more realistic. For example, we might allow module M to accept $+coin$ and $+small$ in all states and specify how module M responds to the insertion of coins and small objects when it is in states other than READY. Such modification of the specification in Fig. 3 would not affect the set of behaviors of M that are safe wrt to U and L, and the modified specification would still satisfy *M using L offers U*.

In general, interfaces U and L, as specified above, admit many different designs of module M that are all deemed correct. If some of these designs are deemed undesirable, they can be ruled out by modifying the allowed event sequences of U and L.

Keep in mind that for each interface, only the intended interaction sequences between the service provider and consumer of the interface are specified. We believe that such specifica-

[4] There is another difference in [15], namely, the restriction that every input must be enabled in every state of an automaton. The blocking of events $+large$, $+small$, $+coin$, $+error$, and $+response$ by module M in various states cannot be specified.

tions are natural. Moreover, with our definition of interface satisfaction, such specifications encode adequate information for separable module design.

B. Proof of Satisfaction

To illustrate our definition of *M using L offers U*, we provide a proof that the specification of M shown in Fig. 3 satisfies the interfaces U and L given. It is easy to see that the Naming constraints are satisfied. Also, every behavior of M corresponds to a path in Fig. 3 starting from state READY.

To show that the Safety constraints are satisfied, let σ denote a finite behavior of M that is safe wrt U and L. Note that neither $+small$ nor $+error$ can appear in σ. From Fig. 3, it is clear that σ must be a finite prefix of (READY) $@\alpha^{\infty}$, where α = ($+coin$, ACCEPTED, $-request$, WAITING, $+response$, DISPENSING, $-choc$, READY). Also, being a behavior, σ ends in a state. It is sufficient to show that for σ ending in each of the four states, the two Safety constraints are satisfied.

- σ ending in state READY—No output event is enabled in READY. Hence the first Safety constraint is satisfied trivially. The only input event e such that $\sigma@e$ is safe wrt U and L is $+coin$, and the event is enabled in READY. Hence, the second Safety contraint is satisfied.
- σ ending in state DISPENSING—The only output event enabled in state DISPENSING is $-choc$, and $\sigma@(-choc)$ is safe wrt U and L. Hence, the first Safety constraint is satisfied. There is no input event e such that $\sigma@e$ is safe wrt U and L. Hence, the second Safety constraint is satisfied trivially.

For σ ending in state ACCEPTED or WAITING, the proof is similar to that given above and is omitted.

To show that the Progress constraints are satisfied, we consider the set of behaviors of M that are safe wrt U and L. This set is made up of the infinite behavior (READY) $@\alpha^{\infty}$, where α = ($+coin$, ACCEPTED, $-request$, WAITING, $+response$, DISPENSING, $-choc$, READY), and all of its prefixes that end in a state.

Module M has two fairness requirements $\{-request\}$ and $\{-choc\}$. The infinite safe behavior is an allowed behavior of M because $-request$ and $-choc$ each appears infinitely often in it. For this allowed behavior, the observable event sequence at interface L is a^{∞} where a = ($-request, +response$), and the observable event sequence at interface U is b^{∞} where b = ($+coin, -choc$). Hence, the Progress constraint is satisfied.

Of the finite safe behaviors, those ending in state READY or WAITING are allowed behaviors of M, because in these states, neither $-choc$ nor $-request$ is enabled.

Of the finite allowed behaviors ending in state READY, the observable event sequence at interface L is a^n, and the observable event sequence at interface U is b^n, where n is a non-negative integer. Hence, the Progress constraint is satisfied.

Of the finite allowed behaviors ending in state WAITING, the observable event sequence at interface L is $b^n@(-request)$, where n is a non-negative integer. Each such behavior is not allowed wrt L. Hence, the Progress constraint is satisfied trivially. (That is, module M is required to offer the service of U only if it interacts with some module that offers the service of L.)

We have proved satisfaction of *M using L offers U*.

V. Composition Theorem

We first define how modules are composed.

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

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

Convention: For any set of modules with distinct names, $\{M_j : j \in J\}$, it is assumed that for all $j, k \in J, j \neq k$,

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

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 $i, j \in J, States(M_i) \cap States(M_j) = \emptyset$. 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:

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

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

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

- $sts\,(M)$ 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*$(I_k) = \emptyset$.

Theorem 1 (Basic Composition): Let modules, M and N, and disjoint interfaces, U and L, satisfy the following:

- M using L offers U, and
- N offers L.

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

A proof of Theorem 1 is not given, because the theorem is subsumed by Theorem 3, which is proved. 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 (Stack Composition): 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.

Proof: The compatibility of $\{M_1, \cdots, M_n\}$ is obvious. To show that their composition offers I_n, it suffices to establish the following inductive step, for $j = 2, \cdots, n$:

> If the composition of $\{M_1, \cdots, M_{j-1}\}$ offers I_{j-1}, and M_j using I_{j-1} offers I_j, then the composition of $\{M_1, \cdots, M_{j-1}, M_j\}$ offers I_j.

But this is implied by Theorem 1, with the composition of $\{M_1, \cdots, M_{j-1}\}$ being N, M_j being M, I_{j-1} being L, and I_j being U. □

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

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 the following:

- *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
  $\forall j \in J$:
  $proj(w, Events(I_j)) \in$ *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 theorem:

Theorem 3 (Bypass composition): 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.

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

Definition: A layered system with layers 1 through J is defined by the following:

- $Modules$, 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) $M \in Modules(j) \wedge j > 1$
      $\Rightarrow L(M) \subseteq \cup_{k<j} Interfaces(k)$
   b) $Inputs(M)$
      $= [\cup_{I \in U(M)} Inputs(I)] \cup [\cup_{I \in L(M)} Outputs(I)]$
   c) $Outputs(M)$
      $= [\cup_{I \in U(M)} Outputs(I)] \cup [\cup_{I \in L(M)} Inputs(I)]$
3) For every pair of distinct modules M and N:
   $U(M) \cap U(N) = \emptyset$
   $L(M) \cap L(N) = \emptyset$

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 requirement 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 $j > 1$

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

Theorem 4 (Dag Composition): 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, $\cup_{k \in \{1,\cdots,J\}} Modules(k)$ is a set of compatible modules and their composition offers $Services(J)$.

A proof of Theorem 4 is given in Appendix B.

VI. 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 $w \in Behaviors(N)$,

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

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

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

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 that a module has been designed and shown to satisfy a set of interfaces. Subsequently, we may want to refine it to derive new modules. (For an example, see [19].) The following theorems are useful for justifying such refinement steps. Their proofs are given in Appendix D.

Theorem 5 (Implementation Replacement): Let M and N be modules, and I, U, and L be interfaces.

a) If M implements N and N offers I, then M offers I.
b) If M implements N and N using L offers U, then M using L offers U.

Theorem 6 (Implementation Transitivity): 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.

VII. Concluding Remarks

We view interfaces as being two-sided. Each interface has a service provider on one side and a service consumer on the other. Interfaces encode all information that the designers of modules need to know, so that each module can be designed and implemented *separately*. To achieve separability, each event in an interface is explicitly defined to be under the control of the service provider or consumer of the interface, and this additional information is used in our definition of interface satisfaction.

We characterize our approach of system design as *decompositional*, which was initially motivated by the design of layered systems. Although a composition theorem, such as the theorems proved in this paper, is useful for both compositional and decompositional approaches to system design, our notion of separability in module design appears to be more important in the decompositional approach.

The concepts of separability and composability are similar to goals of object-oriented programming. There is, however, a nontrivial difference between "objects" and our modules. In the literature of object-oriented designs, interface specifications are limited to the use of preconditions and postconditions for individual operations (routines), as well as invariant assertions—these are *safety* properties. In our theory, on the other hand, modules are service providers and consumers (e.g., network protocols). The notion of providing a service cannot be captured by a safety property. Instead, what we need are *progress* properties (e.g., a vending machine will dispense candy) or, more generally, conditional progress properties (e.g., if someone inserts a coin, the vending machine will dispense candy).

In practice, modules and interfaces can be specified by using many different languages. The theory presented in this paper provides a semantic foundation upon which proof methods can be developed for various specification languages. For interfaces and modules specified in the relational notation [9], we have developed one such proof method [12].

Acknowledgment

We thank Michael Merritt of Bell Laboratories for his constructive criticisms of our proof method in [8], which motivated us to develop the theory presented in this paper. We also thank Tony Hoare of Oxford University and Jay Misra of the University of Texas at Austin for their comments on an earlier version of this paper [11] and for sharing their insights with us.

Appendix A

Proof of Theorem 3 (Bypass Composition Theorem)

Lemma 1. Let I denote the composition of disjoint interfaces $\{I_j: j \in J\}$, and σ a sequence over alternating states and events. Then,

(a) σ is safe wrt I iff $\forall j \in J$, σ is safe wrt I_j, and

(b) σ is allowed wrt I iff $\forall j \in J$, σ is allowed wrt I_j.

Lemma 1 is an immediate consequence of the definition of the composition of a set of disjoint interfaces in Section 4.

Notation. Let $M_1 | M_2$ denote the composition of any two compatible modules M_1 and M_2.

Notation. For a sequence σ over alternating $States(M_1|M_2)$ and $Events(M_1|M_2)$, we use $image(\sigma, M_i)$, for $i \in \{1,2\}$, to denote a sequence over alternating $States(M_i)$ and $Events(M_i)$ obtained from σ as follows: First, replace every state s in σ by $image(s, M_i)$ and delete every event not in $Events(M_i)$. Second, wherever a state t appears as consecutive elements in the resulting sequence, replace the string of consecutive t elements by a single t.

Lemma 2. Let σ be a behavior of $M_1|M_2$, and $F \subseteq Outputs(M_i) \cup Internals(M_i)$ a fairness requirement, where $i \in \{1,2\}$. Suppose the following hold:

(a) $\forall e \in Outputs(M_i) \cup Internals(M_i)$,

$\forall$ finite prefix w of σ ending in a state:

$image(last(w), M_i) \in enabled_{M_i}(e)$

$\Rightarrow last(w) \in enabled_{M_1|M_2}(e)$

(b) σ satisfies fairness requirement F of $M_1|M_2$.

Then $image(\sigma, M_i)$ satisfies fairness requirement F of M_i.

A proof of Lemma 2 is given in Appendix C.

Lemma 3. For $i \in \{1,2\}$:

$\sigma \in Behaviors(M_1|M_2) \Rightarrow image(\sigma, M_i) \in Behaviors(M_i)$.

Lemma 3 is an immediate consequence of the definition of $Transitions_{M_1|M_2}$ (see definition of module composition in Section 4).

We proceed to prove Theorem 3.

Notation. We use L^+ to denote the composition of L and V, and U^+ the composition of U and V.

From Naming constraints of M *using* L *offers* U and N *offers* L^+, we infer that M and N are compatible. Also, Naming constraints of $M|N$ *offers* U^+ are satisfied.

It remains to show that Safety and Progress constraints of $M|N$ *offers* U^+ are satisfied. To do so, we need Lemmas 1, 2 and 3. We also need Lemmas 4-7 which are specific to modules M and N in Theorem 3.

Lemma 4. For any sequence σ over alternating $States(M|N)$ and $Events(M|N)$:

(a) $image(\sigma, M)$ is safe wrt $U(L)$ iff σ is safe wrt $U(L)$

(b) $image(\sigma, N)$ is safe wrt $V(L)$ iff σ is safe wrt $V(L)$

(c) $image(\sigma, M)$ is allowed wrt $U(L)$ iff σ is allowed wrt $U(L)$

(d) $image(\sigma, N)$ is allowed wrt $V(L)$ iff σ is allowed wrt $V(L)$.

Lemma 4 is an immediate consequence of

$Events(M) \supseteq Events(U) \cup Events(L)$ and

$Events(N) \supseteq Events(V) \cup Events(L)$

which follow from Naming constraints of M *using* L *offers* U and N *offers* L^+.

The following lemma states that if U^+ is not misused, then M and N do not misuse L^+ (proof in Appendix C).

Lemma 5.

$\sigma \in Behaviors(M|N) \wedge \sigma$ safe wrt $U^+ \Rightarrow \sigma$ safe wrt L^+.

The following lemma states that if U^+ is not misused, then, in the composition $M|N$, module N does not block any event e controlled by module M. That is, if e of $M|N$ is disabled in a state s of $M|N$, then e of M is disabled in $image(s, M)$ (proof in Appendix C).

Lemma 6.

$\sigma \in FiniteBehaviors(M|N) \wedge \sigma$ safe wrt U^+

$\wedge e \in (Outputs(M) \cup Internals(M)) \wedge image(last(\sigma), M) \in enabled_M(e)$

$\Rightarrow last(\sigma) \in enabled_{M|N}(e)$.

The following lemma states that if U^+ is not misused, then, in the composition $M|N$, module M does not block any event e controlled by module N. That is, if e of $M|N$ is disabled in a state s of $M|N$, then e of N is disabled in $image(s, N)$ (proof in Appendix C).

Lemma 7.

$\sigma \in FiniteBehaviors(M|N) \wedge \sigma$ safe wrt U^+

$\wedge e \in (Outputs(N) \cup Internals(N)) \wedge image(last(\sigma), N) \in enabled_N(e)$

$\Rightarrow last(\sigma) \in enabled_{M|N}(e)$.

Resuming our proof of Theorem 3, we proceed to prove that Safety and Progress constraints of $M|N$ *offers* U^+ are satisfied.

Proof for Safety constraints:

(1) $\sigma \in FiniteBehaviors(M|N)$ (Assumption)

(2) σ safe wrt U^+ (Assumption)

[We have to prove the following:

$\forall e \in Inputs(U^+)$: $\sigma@e$ safe wrt $U^+ \Rightarrow last(\sigma) \in enabled_{M|N}(e)$

$\forall e \in Outputs(U^+)$: $last(\sigma) \in enabled_{M|N}(e) \Rightarrow \sigma@e$ safe wrt U^+]

(3) σ is safe wrt L^+ (1, 2, Lemma 5)

(4) $image(\sigma, M) \in FiniteBehaviors(M)$ (1, Lemma 3)

(5) $image(\sigma, M)$ is safe wrt U and L (2, 3, Lemma 1(a), Lemma 4(a))

(6) $image(\sigma, N) \in FiniteBehaviors(N)$ (1, Lemma 3)

(7) $image(\sigma, N)$ is safe wrt L^+ (3, Lemma 4(b))

(8) $e \in Inputs(V) \wedge \sigma@e$ safe wrt U^+

$\Rightarrow last(\sigma) \in enabled_{M|N}(e)$ (proof follows)

(8.1) $e \in Inputs(V)$ (Assumption)

(8.2) $\sigma@e$ safe wrt U^+ (Assumption)

(8.3) $\sigma@e$ safe wrt V (8.2, Lemma 1(a))

(8.4) $image(\sigma, N)@e$ safe wrt V (8.3, Naming constraints of N *offers* L^+)

(8.5) $image(\sigma, N)@e$ safe wrt L^+ (7, $e \notin Events(L)$, 8.4, definition of L^+)

(8.6) $last(image(\sigma, N)) \in enabled_N(e)$ (6, 7, 8.1, 8.5, second Safety constraint of N *offers* L^+)

(8.7) $last(\sigma) \in enabled_{M|N}(e)$ (8.6, $e \notin Events(M)$, definition of $Transitions_{M|N}(e)$)

[(8) follows from (8.1), (8.2), (8.7).]

(9) $e \in Inputs(U) \wedge \sigma@e$ safe wrt U^+

$\Rightarrow last(\sigma) \in enabled_{M|N}(e)$ (proof follows)

(9.1) $e \in Inputs(U)$ (Assumption)

(9.2) $\sigma@e$ safe wrt U^+ (Assumption)

(9.3) $\sigma@e$ safe wrt U (9.2, Lemma 1(a))

(9.4) $image(\sigma, M)@e$ safe wrt U (9.3, Lemma 4(a))

(9.5) $image(\sigma, M)@e$ safe wrt U and L (5, 9.4, $e \notin Events(L)$)

(9.6) $last(image(\sigma, M)) \in enabled_M(e)$ (4, 9.4, second Safety constraint of M *using* L *offers* U)

(9.7) $last(\sigma) \in enabled_{M|N}(e)$ (9.6, $e \notin Events(N)$)

[(9) follows from (9.1), (9.2), (9.7).]

(10) $e \in Outputs(V) \wedge last(\sigma) \in enabled_{M|N}(e)$

$\Rightarrow \sigma@e$ safe wrt U^+ (proof follows)

(10.1) $e \in Outputs(V)$ (Assumption)

(10.2) $last(\sigma) \in enabled_{M|N}(e)$ (Assumption)

(10.3) $image(last(\sigma), N) \in enabled_N(e)$ (10.2, definition of $Transitions_{M|N}(e)$)

(10.4) $last(image(\sigma, N)) \in enabled_N(e)$ (10.3)

(10.5) $image(\sigma, N)@e$ safe wrt L^+ (6, 7, 10.1, 10.4, first Safety constraint of N *offers* L^+)

(10.6) $\sigma@e$ safe wrt L^+ (10.5, Lemma 4(b))

(10.7) $\sigma@e$ safe wrt V (10.6, Lemma 1(a))

(10.8) $\sigma@e$ safe wrt U (2, $e \notin Events(U)$)

(10.9) $\sigma@e$ safe wrt U^+ (10.7, 10.8)

[(10) follows from (10.1), (10.2), (10.9).]

(11) $e \in Outputs(U) \wedge last(\sigma) \in enabled_{M|N}(e)$

$\Rightarrow \sigma@e$ safe wrt U^+ (proof follows)

(11.1) $e \in Outputs(U)$ (Assumption)

(11.2) $last(\sigma) \in enabled_{M|N}(e)$ (Assumption)

(11.3) $image(last(\sigma), M) \in enabled_M(e)$
(11.2, definition of $Transitions_{M|N}(e)$)

(11.4) $last(image(\sigma, M)) \in enabled_M(e)$ (11.3)

(11.5) $image(\sigma, M)@e$ safe wrt U and L
(4, 5, 11.4, first Safety constraint of M *using* L *offers* U)

(11.6) $\sigma@e$ safe wrt U (11.5, Lemma 4(a))

(11.7) $\sigma@e$ safe wrt V (2, Lemma 1(a), $e \notin Events(V)$)

(11.8) $\sigma@e$ safe wrt U^+ (11.6, 11.7, Lemma 1(a))

[(11) follows from (11.1), (11.2), (11.8).]

[Safety constraints follow from (1), (2), (8), (9), (10), (11).]

End of proof for Safety constraints.

Proof for Progress constraints:

(1) $\sigma \in AllowedBehaviors(M|N)$ (Assumption)

(2) σ safe wrt U^+ (Assumption)

[We need to prove that σ is allowed wrt U^+.]

(3) σ is safe wrt L^+ (1, 2, Lemma 5)

(4) $image(\sigma, M) \in Behaviors(M)$ (1, Lemma 3)

(5) $image(\sigma, M)$ is safe wrt U and L (2, 3, Lemma 4(a), Lemma 1(a))

(6) $image(\sigma, N) \in Behaviors(N)$ (1, Lemma 3)

(7) $image(\sigma, N)$ is safe wrt L^+ (2, 3, Lemma 4(b))

[We first prove that
$image(\sigma, M) \in AllowedBehaviors(M)$ and
$image(\sigma, N) \in AllowedBehaviors(N)$.]

(8) For every finite prefix w of σ ending in a state:
w is a finite behavior of $M|N \wedge w$ safe wrt U^+ (1, 2)

(9) $image(\sigma, M) \in AllowedBehaviors(M)$ (proof follows)

(9.1) $F \subseteq Outputs(M) \cup Internals(M)$
is a fairness requirement of M (Assumption)

(9.2) F is a fairness requirement of $M|N$
(9.1, definition of fairness requirements of $M|N$)

(9.3) σ satisfies F of $M|N$ (9.2, 1)

(9.4) For every finite prefix w of σ ending in a state,
for all $e \in Outputs(M) \cup Internals(M)$:
$image(last(w), M) \in enabled_M(e) \Rightarrow last(w) \in enabled_{M|N}(e)$
(8, Lemma 6 with σ renamed w)

(9.5) $image(\sigma, M)$ satisfies F of M
(1, 9.1, 9.3, 9.4, Lemma 2)

[(9) follows from, (4), (9.1) and (9.5).]

(10) $image(\sigma, N) \in AllowedBehaviors(N)$ (proof follows)

(10.1) $F \subseteq Outputs(N) \cup Internals(N)$
is a fairness requirement of N (Assumption)

(10.2) F is a fairness requirement of $M|N$
(10.1, definition of fairness requirements of $M|N$)

(10.3) σ satisfies F of $M|N$ (10.2, 1)

(10.4) For every finite prefix w of σ ending in a state,
for all $e \in Outputs(N) \cup Internals(N)$:
$image(last(w), N) \in enabled_N(e) \Rightarrow last(w) \in enabled_{M|N}(e)$
(8, Lemma 7 with σ renamed w)

(10.5) $image(\sigma, N)$ satisfies F of N
(1, 10.1, 10.3, 10.4, Lemma 2)

[(10) follows from (6), (10.1) and (10.5).]

(11) $image(\sigma, N)$ is allowed wrt L^+
(10, 6, 7, Progress constraint of N *offers* L^+)

(12) $image(\sigma, N)$ is allowed wrt L (11, Lemma 1(b))

(13) $image(\sigma, M)$ is allowed wrt L (12, Lemma 4(c-d))

(14) $image(\sigma, M)$ is allowed wrt U
(4, 5, 13, Progress constraint of M *using* L *offers* U)

(15) σ is allowed wrt U (14, Lemma 4(c))

(16) $image(\sigma, N)$ is allowed wrt V (11, Lemma 1(b))

(17) σ is allowed wrt V (16, Lemma 4(d))

(18) σ is allowed wrt U^+ (15, 17)

[(1), (2), (18) imply Progress constraints.]

End of proof for Progress constraints.

End of proof of Theorem 3.

Appendix B

Proof of Theorem 4 (Dag Composition Theorem)

A set of modules is *disjoint* iff for every pair of modules M and N in the set, $Events(M) \cap Events(N) = \emptyset$. Clearly, disjoint modules are compatible. To prove Theorem 4, we need two lemmas pertaining to disjoint modules.

Lemma 8. Let *Modules* be a set of disjoint modules, and $\{I(M): M \in Modules\}$ be a set of disjoint interfaces. If M offers $I(M)$ for every $M \in Modules$, then the composition of $\{M: M \in Modules\}$ offers the composition of $\{I(M): M \in Modules\}$.

A proof of Lemma 8 is given in Appendix C. This is an obvious result for a set of disjoint modules, each of which offers its services without being interfered by any other module in the set. This result can be extended to the following lemma (proof omitted because it is similar to the proof of Lemma 8).

Lemma 9. Let *Modules* be a set of disjoint modules, and $\{L(M): M \in Modules\} \cup \{U(M): M \in Modules\}$ be a set of disjoint interfaces. If M using $L(M)$ offers $U(M)$ for every $M \in Modules$, then the composition of $\{M: M \in Modules\}$ using the composition of $\{L(M): M \in Modules\}$ offers the composition of $\{U(M): M \in Modules\}$.

We proceed to prove Theorem 4. Naming constraints (1)-(3) in the definition of our model of layered systems ensure that *Modules* is a compatible set of modules. This can be proved by contradiction as follows.

Assume that the following holds for some event e and distinct modules M and N in *Modules*:

$$X \equiv e \in Outputs(M) \cap Outputs(N)$$

From Naming constraint (2c), $e \in Outputs(M)$ implies $Y_1 \vee Y_2$, where

$$Y_1 \equiv e \in Outputs(I_1), \text{ for some } I_1 \in U(M)$$
$$Y_2 \equiv e \in Inputs(I_2), \text{ for some } I_2 \in L(M)$$

and $e \in Outputs(N)$ implies $Z_1 \vee Z_2$, where

$$Z_1 \equiv e \in Outputs(K_1), \text{ for some } K_1 \in U(N)$$
$$Z_2 \equiv e \in Inputs(K_2), \text{ for some } K_2 \in L(N)$$

Therefore, we have $(Y_1 \vee Y_2) \wedge (Z_1 \vee Z_2)$, which by boolean algebra is equivalent to $(Y_1 \wedge Z_1) \vee (Y_1 \wedge Z_2) \vee (Y_2 \wedge Z_1) \vee (Y_2 \wedge Z_2)$. Because *Interfaces* is a set of disjoint interfaces, $Y_1 \wedge Z_1$ implies $I_1 = K_1$, i.e. they are the same interface. But this contradicts Naming constraint (3). Therefore $(Y_1 \wedge Z_1)$ is false. Similarly, $(Y_2 \wedge Z_2)$ is false. Therefore, X implies $(Y_1 \wedge Z_2) \vee (Y_2 \wedge Z_1)$.

Assume that $Y_1 \wedge Z_2$ holds. Because *Interfaces* is a set of disjoint interfaces, $Y_1 \wedge Z_2$ implies that $I_1 = K_2$. But this is not possible for the following reason: $I_1 = K_2$ together with $e \in Inputs(K_2)$, from Z_2, imply $e \in Inputs(I_1)$, which contradicts Y_1 because $Inputs(I_1)$ and $Outputs(I_1)$ are required to be disjoint. A similar argument shows that $Y_2 \wedge Z_1$ is false.

Let $S(j)$ denote the composition of $\{I: I \in Services(j)\}$. Let $Layer(j)$ denote the composition of $\{M: M \in Modules(j)\}$. Let $System(1..j)$ denote the composition of $\{Layer(k): k \in \{1, \cdots, j\}\}$. We have to prove that $System(1..J)$ offers $S(J)$. The proof is by induction.

Modules(1) is a set of disjoint modules because *Modules* is a set of disjoint modules. For any two modules M and N in *Modules*(1), $U(M)$ and $U(N)$ are disjoint, because of Naming constraint (3). Also, from condition (a) in the hypothesis of Theorem 4, each $M \in$ *Modules*(1) satisfies M *offers* $U(M)$. Therefore, from Lemma 8, we have the base case, namely:

Layer(1) offers $S(1)$

Induction step:

If *System*(1..$j-1$) offers $S(j-1)$, then *System*(1..j) offers $S(j)$

Modules(j) is a set of disjoint modules because *Modules* is a set of disjoint modules. For any two modules M and N in *Modules*(j), interfaces $U(M)$, $U(N)$, $L(M)$ and $L(N)$ are disjoint, because of Naming constraint (3) and because *Interfaces*($j-1$) and *Interfaces*(j) are disjoint. Also, from condition (b) in the hypothesis of Theorem 4, each $M \in$ *Modules*(j) satisfies M *offers* $U(M)$ *using* $L(M)$. Therefore, from Lemma 9, we have the following:

(i) *Layer*(j) using L offers U,

where

L is the composition of $\bigcup_{M \in Modules(j)} L(M)$, and

U is the composition of $\bigcup_{M \in Modules(j)} U(M)$.

The hypothesis, *System*(1..$j-1$) offers $S(j-1)$, of the induction step, can be written as follows:

(ii) *System*(1..$j-1$) offers L, V,

where V is the composition of [*Services*($j-1$) $-$ $\bigcup_{M \in Modules(j)} L(M)$].

From Naming constraints of our model of layered systems and the above definition of V, interfaces L, U and V are disjoint. Applying Theorem 3 to (i) and (ii), we infer that *System*(1..j) offers U, V. But the composition of U and V is the same as the composition of *Services*(j). This establishes the induction step.

End of proof of Theorem 4.

Appendix C

Proofs of Lemmas 2, 5, 6, 7 and 8

Proof of Lemma 2

Here we use $enabled_{M_1|M_2}(F)$ to represent [$\exists e \in F$: $enabled_{M_1|M_2}(e)$], and $enabled_{M_i}(F)$ to represent [$\exists e \in F$: $enabled_{M_i}(e)$].

Case 1: σ is finite

(1) σ is finite (Assumption)

(2) $image(\sigma, M_i)$ is finite (1, definition of $image(\sigma, M_i)$)

(3) $last(\sigma) \notin enabled_{M_1|M_2}(F)$ (1, b in Lemma 2)

(4) $image(last(\sigma), M_i) \notin enabled_{M_i}(F)$ (3, a in Lemma 2)

(5) $last(image(\sigma, M_i)) \notin enabled_M(F)$ (4, 1, $image(last(\sigma), M_i) = last(image(\sigma, M_i))$)

(6) $image(\sigma, M_i)$ satisfies F of M_i (2, 5)

End of proof for case 1.

In the remaining cases below, which are for infinite σ, we assume that $\sigma = (s_0, e_0, s_1, e_1, \cdots)$. Then condition (a) in Lemma 2 can be restated as the following:

(c) $\forall e \in Outputs(M_i) \cup Internals(M_i)$, $\forall s_k \in \sigma$:
$image(s_k, M_i) \in enabled_{M_i}(e) \Rightarrow s_k \in enabled_{M_1|M_2}(e)$

Case 2: σ is infinite and $image(\sigma, M_i)$ is finite

(1) σ is infinite (Assumption)

(2) $image(\sigma, M_i)$ is finite (Assumption)

(3) $\exists n$, $\forall k > n$:
$e_k \notin Events(M_i)$ and $image(s_k, M_i) = last(image(\sigma, M_i))$ (1, 2, definition of $image(\sigma, M_i)$)

(4) $\exists j > n$: $s_j \notin enabled_{M_1|M_2}(F)$ (b in Lemma 2, 1, 3)

(5) $\exists j > n$: $image(s_j, M_i) \notin enabled_{M_i}(F)$ (4, c)

(6) $last(image(\sigma, M_i)) \notin enabled_{M_i}(F)$ (5, 3)

(7) $image(\sigma, M_i)$ satisfies F of M_i (6, 2)

End of proof for case 2.

Case 3: σ is infinite and $image(\sigma, M_i)$ is infinite

(1) σ is infinite (Assumption)

(2) $image(\sigma, M_i)$ is infinite (Assumption)

(3) Let $proj(\sigma, Events(M_i)) = (e_{i_0}, e_{i_1}, \cdots)$ (Assumption)
Let $R_0 = \{0, \cdots, i_0\}$, and $R_j = \{i_{j-1}+1, \cdots, i_j\}$ for $j = 1, 2, \cdots$

(4) $image(\sigma, M_i) = (t_0, e_{i_0}, t_1, e_{i_1}, \cdots)$
where $\forall j \geq 0$, $\forall k \in R_j$, $image(s_k, M_i) = t_j$ (2, 3, definition of $image(\sigma, M_i)$)

(5) F occurs infinitely often in σ
$\Rightarrow$ F occurs infinitely often in $image(\sigma, M_i)$ (3, 4)

(6) F disabled infinitely often in σ (Assumption)
Let F be disabled at states $(s_{k_1}, s_{k_2}, \cdots)$
Let $J = \{j: \exists k_l \in R_j\}$

(7) $\forall l > 0$: $s_{k_l} \notin enabled_{M_1|M_2}(F)$ (6, definition of the s_{k_l}'s)

(8) $\forall l > 0$: $image(s_{k_l}, M_i) \notin enabled_{M_i}(F)$ (7, c)

(9) $\forall j \in J$: $t_j \notin enabled_{M_i}(F)$ (4, definition of J (in 6))

(10) J is infinite ($(k_1, k_2, \cdots)$ is infinite (from 6), every R_j is finite (from 2, 3))

(11) F of M_i is disabled infinitely often in $image(\sigma, M_i)$ (4, 9, 10)

(12) F of $M_1|M_2$ disabled infinitely often in σ
$\Rightarrow$ F of M_i disabled infinitely often in $image(\sigma, M_i)$ (6, 11)
[(5) and (12) prove that $image(\sigma, M_i)$ satisfies F for case 3.]

End of proof for case 3.

End of proof of Lemma 2.

Proof of Lemma 5

From the definition of 'safe wrt', it is sufficient to prove Lemma 5 for a finite behavior σ. The proof is by induction on the length of σ.

Base case:

Lemma 5 holds trivially for σ of length 0.

Induction step:

[We assume Lemma 5 for σ and show that it holds for $\sigma@(e, s)$.]

(1) $\sigma@(e, s) \in FiniteBehaviors(M|N)$ (Assumption)

(2) $\sigma@(e, s)$ is safe wrt U^+ (Assumption)

(3) $\sigma@(e, s)$ is safe wrt V (2, Lemma 1(a))

[Because of (3), it suffices to show that $\sigma@(e, s)$ is safe wrt L.]

(4) $\sigma \in FiniteBehaviors(M|N)$ (1)

(5) σ is safe wrt U^+ (2)

(6) σ is safe wrt L^+ (4, 5, Induction hypothesis)

(7) σ is safe wrt U (5, Lemma 1(a))

(8) σ is safe wrt L (6, Lemma 1(a))

(9) $e \notin Events(L) \Rightarrow \sigma@(e, s)$ safe wrt L (8)

(10) $last(\sigma) \in enabled_{M|N}(e)$ (1)

(11) $e \in Inputs(L) \Rightarrow \sigma@(e, s)$ safe wrt L (proof follows)

(11.1) $e \in Inputs(L)$ (Assumption)

(11.2) $image(\sigma, M) \in Behaviors(M)$ (4, Lemma 3)

(11.3) $image(\sigma, M)$ safe wrt U and L (7, 8, Lemma 4(a))

(11.4) $image(last(\sigma), M) \in enabled_M(e)$ (10, definition of $Transitions_{M|N}(e)$)

(11.5) $image(\sigma, M)@e$ is safe wrt L
(11.1, 11.2, 11.3, 11.4, first Safety constraint of M *using* L *offers* U)

(11.6) $\sigma@(e, s)$ is safe wrt L (11.5, Lemma 4(a))

[(11) follows from (11.1) and (11.6).]

(12) $e \in Outputs(L) \Rightarrow \sigma@(e, s)$ safe wrt L (proof follows)

(12.1) $e \in Outputs(L)$ (Assumption)

(12.2) $image(\sigma, N) \in Behaviors(N)$ (4, Lemma 3)

(12.3) $image(\sigma, N)$ safe wrt L^+ (6, Lemma 4(b))

(12.4) $image(last(\sigma), N) \in enabled_N(e)$ (10, definition of $Transitions_{M|N}(e)$)

(12.5) $image(\sigma, N)@e$ is safe wrt L^+ (12.1, 12.2, 12.3, 12.4, first Safety constraint of N *offers* L^+)

(12.6) $\sigma@(e, s)$ is safe wrt L (12.5, Lemma 4(b))

[(12) follows from (12.1) and (12.6).]

(13) $\sigma@(e, s)$ is safe wrt L (9, 11, 12)

(14) $\sigma@(e, s)$ is safe wrt L^+ (3, 13)

[(1), (2), (14) imply induction step.]

End of proof of Lemma 5.

Proof of Lemma 6

(1) $\sigma \in FiniteBehaviors(M|N)$ (Assumption)

(2) σ safe wrt U^+ (Assumption)

(3) $e \in Outputs(M) \cup Internals(M)$ (Assumption)

(4) $image(last(\sigma), M) \in enabled_M(e)$ (Assumption)

[We have to show that $last(\sigma) \in enabled_{M|N}(e)$.]

(5) $e \notin Inputs(N) \Rightarrow last(\sigma) \in enabled_{M|N}(e)$ (proof follows)

(5.1) $e \notin Inputs(N)$ (Assumption)

(5.2) $e \notin Events(N)$ (5.1, 3)

(5.3) $last(\sigma) \in enabled_{M|N}(e)$ (5.2, 4, definition of $Transitions_{M|N}(e)$)

[(5) follows from (5.1) and (5.3).]

(6) $e \in Inputs(N) \Rightarrow last(\sigma) \in enabled_{M|N}(e)$ (proof follows)

(6.1) $e \in Inputs(N)$ (Assumption)

(6.2) σ safe wrt L^+ (1, 2, Lemma 5)

(6.3) σ safe wrt U^+ and L^+ (2, 6.2)

(6.4) $image(\sigma, M)$ is safe wrt U and L (6.3, Lemma 4(a), Lemma 1(a))

(6.5) $image(\sigma, M) \in FiniteBehaviors(M)$ (1, Lemma 3)

(6.6) $image(\sigma, M)@e$ is safe wrt U and L (6.4, 6.5, 4, first Safety constraint of M *using* L *offers* U)

(6.7) $\sigma@e$ is safe wrt U and L (6.6, Lemma 4(a))

(6.8) $image(\sigma, N) \in FiniteBehaviors(N)$ (1, Lemma 3)

(6.9) $image(\sigma, N)$ is safe wrt L^+ (6.2, Lemma 4(b))

(6.10) $image(\sigma, N)@e$ is safe wrt L (6.7, Lemma 4(b))

(6.11) $image(\sigma, N)@e$ is safe wrt L^+ (6.10, $e \notin Events(V)$)

(6.12) $last(image(\sigma, N)) \in enabled_N(e)$ (6.8, 6.9, 6.11, second Safety constraint of N *offers* L^+)

(6.13) $last(\sigma) \in enabled_{M|N}(e)$ (4, 6.12, definition of $Transitions_{M|N}(e)$)

[(6) follows from (6.1) and (6.13).]

(7) $last(\sigma) \in enabled_{M|N}(e)$ (5, 6)

[(1), (2), (3), (4), (7) imply Lemma 6.]

End of proof of Lemma 6.

Proof of Lemma 7

The proof is similar to proof of Lemma 6.

(1) $\sigma \in FiniteBehaviors(M|N)$ (Assumption)

(2) σ safe wrt U^+ (Assumption)

(3) $e \in Outputs(N) \cup Internals(N)$ (Assumption)

(4) $image(last(\sigma), N) \in enabled_N(e)$ (Assumption)

[We have to show that $last(\sigma) \in enabled_{M|N}(e)$.]

(5) $e \notin Inputs(M) \Rightarrow last(\sigma) \in enabled_{M|N}(e)$ (proof follows)

(5.1) $e \notin Inputs(M)$ (Assumption)

(5.2) $e \notin Events(M)$ (5.1, 3)

(5.3) $last(\sigma) \in enabled_{M|N}(e)$ (5.2, 4, definition of $Transitions_{M|N}(e)$)

[(5) follows from (5.1) and (5.3).]

(6) $e \in Inputs(M) \Rightarrow last(\sigma) \in enabled_{M|N}(e)$ (proof follows)

(6.1) $e \in Inputs(M)$ (Assumption)

(6.2) σ safe wrt L^+ (1, 2, Lemma 5)

(6.3) σ safe wrt U^+ and L^+ (2, 6.2)

(6.4) $image(\sigma, N)$ is safe wrt L^+ (6.3, Lemma 4(b))

(6.5) $image(\sigma, N) \in FiniteBehaviors(N)$ (1, Lemma 3)

(6.6) $image(\sigma, N)@e$ is safe wrt L^+ (6.4, 6.5, 4, first Safety constraint of N *offers* L^+)

(6.7) $\sigma@e$ is safe wrt L^+ (6.6, Lemma 4(b))

(6.8) $\sigma@e$ is safe wrt U^+ (6.3, 3, $e \notin Events(U^+)$)

(6.9) $\sigma@e$ is safe wrt U and L (6.7, 6.8, Lemma 1(a))

(6.10) $image(\sigma, M)@e$ is safe wrt U and L (6.9, Lemma 4(a))

(6.11) $image(\sigma, M) \in FiniteBehaviors(M)$ (1, Lemma 3)

(6.12) $last(image(\sigma, M)) \in enabled_M(e)$ (6.10, 6.11, second Safety constraint of M *using* L *offers* L)

(6.13) $last(\sigma) \in enabled_{M|N}(e)$ (4, 6.12, definition of $Transitions_{M|N}(e)$)

[(6) follows from (6.1) and (6.13).]

(7) $last(\sigma) \in enabled_{M|N}(e)$ (5, 6)

[(1), (2), (3), (4), (7) imply Lemma 7.]

End of proof of Lemma 7.

Proof of Lemma 8

Let X denote the composition of $\{M: M \in Modules\}$. Let Y denote the composition of $\{I(M): M \in Modules\}$. From the disjoincy of the modules and Naming constraints of M *offers* $I(M)$, we infer that Naming constraints of X *offers* Y are satisfied. It remains to prove the Safety and Progress constraints of X *offers* Y.

Proof for Safety constraints:

(1) $\sigma \in FiniteBehaviors(X)$ (Assumption)

(2) σ safe wrt Y (Assumption)

(3) $\forall M \in Modules: image(\sigma, M) \in FiniteBehaviors(M)$ (1, Lemma 3)

(4) $\forall M \in Modules: image(\sigma, M)$ safe wrt $I(M)$ (2)

(5) $e \in Outputs(X) \wedge last(\sigma) \in enabled_X(e) \Rightarrow \sigma@e$ safe wrt Y (proof follows)

(5.1) $last(\sigma) \in enabled_X(e)$ (Assumption)

(5.2) $e \in Outputs(X)$ (Assumption)

(5.3) $e \in Outputs(M)$, for some $M \in Modules$ (5.2, definition of X)

(5.4) $image(last(\sigma), M) \in enabled_M(e)$ (5.1, definition of $Transitions_X(e)$)

(5.5) $last(image(\sigma, M) \in enabled_M(e)$ (5.4)

(5.6) $image(\sigma, M)@e$ safe wrt $I(M)$
(3, 4, 5.3, 5.5, first Safety constraint of *M offers* $I(M)$)

(5.7) $\sigma@e$ safe wrt $I(M)$
(5.6, $Events(I(M)) \subseteq Events(M)$)

(5.8) $\sigma@e$ safe wrt Y
(5.7, $e \notin (Events(Y) - Events(I(M)))$)

[(5) follows from (5.1), (5.2), (5.8).]

(6) $e \in Inputs(X) \wedge \sigma@e$ safe wrt Y (proof follows)
$\Rightarrow last(\sigma) \in enabled_X(e)$

(6.1) $\sigma@e$ safe wrt Y (Assumption)

(6.2) $e \in Inputs(X)$ (Assumption)

(6.3) $e \in Inputs(M)$, for some $M \in Modules$ (6.2, definition of X)

(6.4) $\sigma@e$ safe wrt $I(M)$ (6.1, definition of Y, Lemma 1(a))

(6.5) $image(\sigma, M)@e$ safe wrt $I(M)$
(6.4, $Events(I(M)) \subseteq Events(M)$)

(6.6) $last(image(\sigma, M)) \in enabled_M(e)$
(3, 4, 6.5, second Safety constraint of *M offers* $I(M)$)

(6.7) $image(last(\sigma), M) \in enabled_M(e)$ (6.6)

(6.8) $last(\sigma) \in enabled_X(e)$
(6.7, definition of $Transitions_X(e)$)

[(6) follows from (6.1), (6.2), (6.8).]

[(1), (2), (5), (6) imply Safety constraints.]

End of proof for Safety constraints.

Proof for Progress constraints:

(1) $\sigma \in AllowedBehaviors(X)$ (Assumption)

(2) σ safe wrt Y (Assumption)

(3) $\forall M \in Modules$: $image(\sigma, M) \in Behaviors(M)$ (1, Lemma 3)

(4) $\forall M \in Modules$: $image(\sigma, M)$ safe wrt $I(M)$ (2)

(5) $\forall M \in Modules$, $\forall e \in Events(M)$, $\forall s \in States(X)$:
$s \in enabled_X(e) \Leftrightarrow image(s, M) \in enabled_M(e)$
(disjoincy of modules, definition of $Transitions_X(e)$)

(6) $\forall M \in Modules$: $image(\sigma, M) \in AllowedBehaviors(M)$
(5, Lemma 2 with M_1 being M and M_2 being the composition of $Modules - \{M\}$)

(7) $\forall M \in Modules$: $image(\sigma, M)$ allowed wrt $I(M)$
(6, 4, Progress constraint of *M offers* $I(M)$)

(8) $\forall M \in Modules$: σ allowed wrt $I(M)$ (7, definition of Y)

(9) σ allowed wrt Y (8)

[(1), (2), (9) imply Progress constraints.]

End of proof for Progress constraints.

End of proof of Lemma 8.

Appendix D

Proof of Theorems 5 and 6 (Implementation Theorems)

Notation. For any sequence σ over the set of states and events of a module, we abbreviate the notation $proj(\sigma, Inputs(N) \cup Outputs(N))$ to $proj[\sigma]$.

To prove Theorem 5, we need the following lemmas:

Lemma 10. Let M and N be modules and I an interface.
If M implements N and N offers I, then the following holds:

$\sigma \in Behaviors(M) \wedge \sigma$ safe wrt $I \Rightarrow \sigma$ safe wrt N.

Proof. We apply induction on the length of σ.

Base case:

(1) $\sigma = (s_0) \in FiniteBehaviors(M)$ (Assumption)

(2) $proj[\sigma] = <>$ (1)

(3) σ safe wrt N (2)

[(1) and (3) imply the base case.]

Induction step:

[We assume Lemma 10 for σ and show that it holds for $\sigma@(e, s)$.]

(1) $\sigma@(e, s) \in FiniteBehaviors(M)$ (Assumption)

(2) $\sigma@(e, s)$ is safe wrt I (Assumption)

(3) $\sigma \in FiniteBehaviors(M)$ (1)

(4) σ is safe wrt I (2)

(5) σ is safe wrt N (3, 4, Induction hypothesis)

(6) $last(\sigma) \in enabled_M(e)$ (1)

(7) $e \in Internals(M) \Rightarrow \sigma@(e, s)$ safe wrt N
(5, $e \in Internals(M) \Rightarrow e \notin Events(N)$)

(8) $e \in Outputs(M) \Rightarrow \sigma@(e, s)$ safe wrt N (proof follows)

(8.1) $e \in Outputs(M)$ (Assumption)

(8.2) $\sigma@e$ safe wrt N
(3, 5, 6, 8.1, first Safety constraint of *M implements N*)

(8.3) $\sigma@(e, s)$ safe wrt N (8.2)

[(8) follows from (8.1), (8.3).]

(9) $e \in Inputs(M) \Rightarrow \sigma@(e, s)$ safe wrt N (proof follows)

(9.1) $e \in Inputs(M)$ (Assumption)

(9.2) $e \in Inputs(N)$
(9.1, Naming constraints of *M implements N*)

(9.3) $\exists w \in FiniteBehaviors(N)$: $proj[w] = proj[\sigma]$ (5)

(9.4) w safe wrt I (9.3, 4)

(9.5) $w@e$ safe wrt I (2, 9.3)

(9.6) $last(w) \in enabled_N(e)$
(9.2, 9.3, 9.4, 9.5, second Safety constraint of *N offers I*)

(9.7) $\exists t \in States(N)$: $w@(e, t) \in FiniteBehaviors(N)$ (9.3, 9.6)

(9.8) $proj[w@(e, t)] = proj[\sigma@(e, s)]$ (9.3)

(9.9) $\sigma@(e, s)$ safe wrt N (9.7, 9.8)

[(9) follows from (9.1), (9.9).]

[(1), (2), (7), (8), (9) imply the induction step.]

End of proof of Lemma 10.

Lemma 11. Let M and N be modules, and U and L be interfaces.
If M implements N and N using L offers U, then the following holds:

$\sigma \in Behaviors(M) \wedge \sigma$ safe wrt U and $L \Rightarrow \sigma$ safe wrt N.

The proof of Lemma 11 is the same as the proof of Lemma 10, with "safe wrt I" replaced by "safe wrt U and L", and "N offers I" replaced by "N using L offers U".

Proof of Theorem 5 (Implementation Replacement)

We first prove Theorem 5(a). Naming constraints of *M offers I* follow from Naming constraints of *M implements N* and *N offers I*.

Proof for Safety constraints:

(1) $\sigma \in FiniteBehaviors(M)$ (Assumption)

(2) σ is safe wrt I (Assumption)

(3) σ is safe wrt N (1, 2, Lemma 10)

(4) $e \in Outputs(M) \wedge last(\sigma) \in enabled_M(e)$ (proof follows)
$\Rightarrow \sigma@e$ safe wrt I

(4.1) $e \in Outputs(M)$ (Assumption)

(4.2) $last(\sigma) \in enabled_M(e)$ (Assumption)

(4.3) $\sigma@e$ safe wrt N
(1, 3, 4.1, 4.2, first Safety constraint of *M implements N*)

(4.4) $\exists w \in FiniteBehaviors(N)$, $\exists t \in States(N)$:
$w@(e, t) \in FiniteBehaviors(N)$
$\wedge proj[w@(e, t)] = proj[\sigma@e]$ (4.3)

(4.5) $last(w) \in enabled_N(e) \wedge w$ safe wrt I (4.4, 2)

(4.6) $e \in Outputs(N)$
(4.1, Naming constraint of *M implements N*)

(4.7) $w@e$ safe wrt I

(4.4, 4.5, 4.6, first Safety constraint of *N offers I*)

(4.8) $\sigma@e$ safe wrt I

(4.7, 1, 4.4, Naming constraints of *M implements N* and *N offers I*)

[(4) follows from (4.1), (4.2), (4.8).]

(5) $e \in Inputs(M) \wedge \sigma@e$ safe wrt I

$\Rightarrow last(\sigma) \in enabled_M(e)$ (proof follows)

(5.1) $e \in Inputs(M)$ (Assumption)

(5.2) $\sigma@e$ safe wrt I (Assumption)

(5.3) $e \in Inputs(N)$

(5.1, Naming constraints of *M implements N*)

(5.4) $\exists w \in FiniteBehaviors(N)$: $proj[w]=proj[\sigma]$ (3)

(5.5) $w@e$ safe wrt I (5.4, 5.2)

(5.6) $last(w) \in enabled_N(e)$

(5.3, 5.4, 5.5, second Safety constraint of *N offers I*)

(5.7) $\exists t \in States(N)$: $w@(e,t) \in FiniteBehaviors(N)$ (5.4, 5.6)

(5.8) $proj[w@(e,t)]=proj[\sigma@e]$ (5.4)

(5.9) $\sigma@e$ safe wrt N (5.8, 5.7)

(5.10) $last(\sigma) \in enabled_M(e)$

(5.9, 3, 1, 5.1, second Safety constraint of *M implements N*)

[(5) follows from (5.1), (5.2), (5.10).]

[(1), (2), (4), (5) imply Safety constraints of *M* offers *I*.]

End of proof for Safety constraints.

Proof for Progress constraints:

(1) $\sigma \in AllowedBehaviors(M)$ (Assumption)

(2) σ is safe wrt I (Assumption)

(3) σ is safe wrt N (1, 2, Lemma 10)

(4) σ is allowed wrt N (1, 3, Progress constraint of *M implements N*)

(5) $\exists w \in AllowedBehaviors(N)$: $proj[w]=proj[\sigma]$ (4)

(6) w is safe wrt I (5, 2)

(7) w is allowed wrt I (5, 6, Progress constraint of *N offers I*)

(8) σ is allowed wrt I (5, 7)

[(1), (2), (8) imply Progress constraints.]

End of proof for Progress constraints.

End of proof of Theorem 5(a).

A proof of Theorem 5(b) follows; it is similar to the proof of Theorem 5(a). First, Naming constraints of *M using L offers U* follow from Naming constraints of *M implements N* and *N using L offers U*.

Proof for Safety constraints:

Same as the proof for Safety constraints of Theorem 5(a) with the following replacements: "*M* offers *I*" by "*M* using *L* offers *U*", "*N* offers *I*" by "*N* using *L* offers *U*", "safe wrt *I*" by "safe wrt *U* and *L*", and 'Lemma 10" by "Lemma 11".

End of proof for Safety constraints.

Proof for Progress constraints:

(1) $\sigma \in AllowedBehaviors(M)$ (Assumption)

(2) σ is safe wrt U and L (Assumption)

(3) σ is safe wrt N (1, 2, Lemma 11)

(4) σ is allowed wrt N (1, 3, Progress constraint of *M implements N*)

(5) $\exists w \in AllowedBehaviors(N)$: $proj[w]=proj[\sigma]$ (4)

(6) w is safe wrt U and L (5, 2)

(7) σ is allowed wrt L (Assumption)

(8) w is allowed wrt L (5, 7)

(9) w is allowed wrt U (5, 6, 8, Progress constraint of *N using L offers U*)

(10) σ is allowed wrt U (5, 9)

[(1), (2), (7), (10) imply Progress constraints.]

End of proof for Progress constraints.

End of proof of Theorem 5(b).

Proof of Theorem 6 (Implementation Transitivity)

To prove Theorem 6, we need the following lemma:

Lemma 12. Let M_1, M_2 and M_3 be modules. If M_3 implements M_2 and M_2 implements M_1, then the following holds:

$\sigma \in Behaviors(M_3) \wedge \sigma$ safe wrt $M_1 \Rightarrow \sigma$ safe wrt M_2.

Lemma 12 and Theorem 6 can be proved exactly as Lemma 10 and Theorem 5(a), respectively, with *N offers I* replaced by *N implements I*.

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, Berlin: Springer-Verlag, 1990.

[2] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN Systems*, vol. 14, pp. 25–59, 1987.

[3] S. D. Brookes, C. A. R. Hoare and A. D. Roscoe, "A theory of communicating sequential processes," *JACM*, vol. 31, no. 3, pp. 560–590, 1984.

[4] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971.

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

[6] A. Kay and J. N. Reed, "A rely and guarantee method for timed CSP: A specification and design of a telephone exchange," *IEEE Trans. Software Eng.*, vol. 19, no. 6, pp. 625–639, June 1993.

[7] S. S. Lam and A. U. Shankar, "Protocol verification via projections," *IEEE Trans. Software Eng.*, vol. SE-10, no. 10, pp. 325–342, July 1984.

[8] ——, "Specifying modules to satisfy interfaces: A state transition system approach," *Distributed Comput.*, vol. 6, pp. 39–63, 1992.

[9] ——, "A relational notation for state transition systems," *IEEE Trans. Software Eng.*, vol. 16, pp. 755–775, July 1990.

[10] ——, "A composition theorem for layered systems," *Proceedings 11th Int. Symp. on Protocol Specification, Testing and Verification*, Stockholm, June 1991.

[11] ——, "Understanding interfaces," *Proc. 4th Int. Conf. on Formal Description Techniques (FORTE)*, Sydney, Australia, Nov. 1991.

[12] ——, "A theory of interfaces and modules II—Methodology," Tech. Rep., Department of Computer Sciences, University of Texas at Austin, in preparation.

[13] ——, and T. Y. C. Woo, "Applying a theory of modules and interfaces to security verification," *Proc. Symp. on Research in Security and Privacy*, IEEE Computer Society, May 1991.

[14] L. Lamport, "A simple approach to specifying concurrent systems," *Comm. ACM*, vol. 32, pp. 32–45, Jan. 1989.

[15] N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," *Proc. of the ACM Symp. on Principles of Distributed Comput.*, Vancouver, BC, Canada, Aug. 1987.

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

[17] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Trans. Software Eng.*, vol. SE-7, no. 4, pp. 417–426, July 1981.

[18] A. Pnueli, "In transition from global to modular temporal reasoning

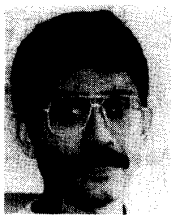
about programs," NATO ASI Series, vol. F13, *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. Berlin: Springer-Verlag, 1984.

[19] A. U. Shankar and S. S. Lam, "A stepwise refinement heuristic for protocol construction," *ACM TOPLAS*, vol. 14, no. 3, pp. 417–461, July 1992.

S. S. Lam (S'69–M'74–SM'80–F'85) received the B.S.E.E. degree with distinction from Washington State University in 1969 and the M.S. and Ph.D. degrees in engineering from the University of California at Los Angeles in 1970 and 1974, respectively.

He is chair of the Department of Computer Sciences, University of Texas, Austin, TX, and holds two endowed professorships. Prior to joining the University of Texas at Austin faculty in 1977, he was a research staff member at the IBM T. J. Watson Research Center, Yorktown Heights, NY, from 1974 to 1977. His research interests are in the areas of computer networks, communication protocols, performance evaluation, formal verification, and network security.

Dr. Lam is a recipient of the 1975 Leonard G. Abraham Prize Paper Award from the IEEE Communications Society. He organized and was program chair of the first ACM SIGCOMM Symposium held at the University of Texas at Austin in 1983. He presently serves on the editorial boards of *IEEE Transactions on Software Engineering* and *IEEE/ACM Transactions on Networking*.



A. Udaya Shankar (S'81–M'82) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1976; the M.S. degree in computer engineering from Syracuse University, Syracuse, NY, in 1978; and the Ph.D. degree in electrical engineering from the University of Texas at Austin in 1982.

Since January 1983, he has been on the faculty of the University of Maryland, College Park, MD, where he is now an Associate Professor of computer science. Since September 1985, he has held a temporary appointment with the Institute for Advanced Computer Studies at the University of Maryland. His research interests include the modeling and analysis of distributed systems and network protocols, from both correctness and performance aspects. He is currently working on assertional methods for layered systems and real-time systems, and on routing protocols for networks and internetworks.