MIT/LCS/TR-342

FOUNDATIONS OF A THEORY OF SPECIFICATION FOR DISTRIBUTED SYSTEMS

Eugene W. Stark

Foundations of a Theory of Specification for Distributed Systems

by

Eugene William Stark

B.E.S. The Johns Hopkins University
(1977)

S.M. Massachusetts Institute of Technology
(1980)

Submitted to the Department of Electrical Engineering
and Computer Science in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August, 1984

Signature of Author____
Department of Electrical Engineering and Computer Science
August 24, 1984

Certified by____
Prof. Nancy A. Lynch, Thesis Supervisor

Accepted by____
Prof. Arthur C. Smith, Chairman., E.E.C.S. Department Committee
on Graduate Students

Foundations of a Theory of Specification for Distributed Systems

by
Eugene William Stark

Submitted to the
Department of Electrical Engineering and Computer Science
on August 24, 1984 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy.

Abstract

This thesis investigates a particular approach, called *state-transition specification*, to the problem of describing the behavior of modules in a distributed or concurrent computer system. A state-transition specification consists of: (1) a state machine, which incorporates the safety or invariance properties of the module, and (2) *validity conditions* on the computations of the machine, which capture the desired liveness or eventuality properties. The theory and techniques of state-transition specification are developed from first principles to a point at which it is possible to write example specifications, to check the specifications for consistency, and to perform correctness proofs using the specifications. The utility of the techniques is demonstrated through examples.

Major contributions of the thesis include: (1) the definition of a semantic model that incorporates *hierarchy of abstraction* and *modular decomposition* as fundamental notions; (2) specification and proof techniques that smoothly handle both safety and liveness properties; (3) techniques that use liveness properties stated in *rely-/guarantee-condition* form to obtain simple proofs of correctness; (4) an interesting and useful notion of consistency for specifications involving liveness properties.

Keywords: state-transition specification, verification, concurrency, hierarchy, modularity, temporal logic, safety, liveness, rely/guarantee conditions

Thesis Supervisor: Nancy A. Lynch

Title: Associate Professor of Computer Science and Engineering

Acknowledgements

I am deeply indebted to my thesis adviser, Nancy Lynch, but for whom this thesis would likely never have been completed. Nancy read many, many difficult drafts of this work with enthusiasm and promptness that went far beyond the mere call of duty. She always seemed to manage not only to identify the most troublesome portions of each draft, but to make insightful suggestions for improvement as well. I am also grateful to John Guttag, Barbara Liskov, and Albert Meyer for their suggestions on improving the presentation. Discussions with Bill Weihl helped to formulate ideas during the early stages.

In an entirely different category are my parents, Joan S. Stark and William L. Stark, Jr., who made it seem natural that I should seek and complete graduate education, and whose love and support during this endeavor I cannot adequately acknowledge. Julian Stanley of Johns Hopkins made it possible for me to pursue the undergraduate portion of my education at an accelerated rate.

Finally, I would like to thank the chessplayers at Au Bon Pain in Harvard Square for providing a much-needed diversion during the past year.

CONTENTS

1. Introduction

The purpose of this thesis is to investigate a particular approach, called *state-transition specification*, to the problem of describing the behavior of modules in a concurrent or distributed computer system. In the state-transition approach, the desired behavior is described in terms of a kind of state machine whose computations generate records of event occurrences, called *observations*. A state-transition specification consists of two parts: (1) the definition of the state machine, which incorporates the "safety" or invariance properties of the module, and (2) the definition of some *validity conditions* on the computations of the machine, whose purpose is to capture the desired module "liveness" or eventuality properties. A state-transition specification defines a set of "acceptable" observations, namely the observations produced by valid computations of the state machine. A module behavior satisfies such a specification if the module behavior contains only acceptable observations.

The idea of describing module behavior with the help of state machines is not new, having already been proposed in various forms by other authors, [e.g. Parnas72, Yonezawa77, Lamport83]. However, previous work seems to be concerned primarily with *how* to write module specifications, and *how* to use proof rules to prove the correctness of implementations. The important issues of what constitutes the *meaning* of a specification, and what it *means* for an implementation to be correct, have not received satisfactory treatment. As a result, it is impossible to answer important questions such as: "What rules are *sound* for proving the correctness of an implementation," and "When is a specification *consistent*?"

This thesis improves upon previous work by systematically developing the theory and techniques of specification from "first principles" to a point at which it is possible to write example specifications, to prove implementations correct, and to check specifications for consistency. The theory incorporates an underlying semantic model within which one can formulate language-independent definitions of the notions of "implementation" and "correctness." The meaning of state-transition specifications is defined in terms of the model, and all proof techniques are shown to be sound with respect to the model.

The major contributions of this thesis are:

(1) The definition of a semantic model that incorporates *hierarchy of abstraction* and *modular decomposition* as fundamental notions.

(2) Specification and proof techniques that smoothly handle both safety and liveness properties.

(3) Techniques that use liveness properties stated in *rely-/guarantee-condition* form to obtain simple proofs of correctness.

(4) An interesting and useful notion of consistency for specifications involving liveness properties.

(5) Illustration of the utility of the ideas developed through specifications, implementations, and correctness proofs for three examples:

(a) a *synchronizer* module, which is implemented by a ring-structured network of *synchronizer component* modules,

(b) a *resource management* module, which is implemented by a tree-structured network of *local resource manager* modules,

(c) a *message transmission* module, which is implemented by unreliable *transmission line* modules, a *send protocol module*, and a *receive protocol module*, which together obey the *alternating bit protocol*.

1.1 Scope of the Thesis

A *specification* is a piece of text whose purpose is to describe the desired operation of a module in a computer system. Specifications form an integral part of a "top-down" design method in which design proceeds by the successive decomposition of a module to be implemented into a collection of interacting component modules [Liskov79, Wirth71]. The purpose of specifications in such an approach is to serve as a contract between the user and the implementer of a module. This helps to limit complexity by permitting a system to be decomposed into modules of reasonable size, such that each module depends only upon the specifications, and not the implementations, of the modules with which it interacts.

To permit the possibility of rigorous reasoning about specifications, a specification language should be given a formal semantics in terms of an underlying mathematical *semantic domain*. In this thesis, we use the term *behavior* to refer to the elements of a semantic domain, since the purpose of these elements is to serve as a mathematical model of the behavior of a portion of a real-world computer system. The semantics of a specification language describe how each specification denotes a set of behaviors that *satisfy* the specification. If the semantics of a programming language are defined so that each important program fragment denotes a behavior, then it is possible to derive syntactic rules for proving that (the denotations of) program fragments satisfy (the denotations of) specifications. The purpose of this thesis is not to propose particular formal specification or programming languages, but rather to investigate a collection of language-independent semantic concepts upon which particular specification and programming languages might be based. We therefore assume that specification and programming languages can have their meanings defined in terms of behaviors, and do not concern ourselves with the precise method by which this is accomplished.

In this thesis, we are concerned with concurrent or distributed systems. By this we mean systems that are most naturally viewed as a collection of independent, communicating modules, such that effects of concurrent operation of the various modules form a significant part of the description of system behavior. This thesis is primarily concerned with the concurrency aspect of distributed computing; while the model and techniques do not rule out the possibility of treating other aspects such as node crashes and network failures, no special structure to deal with these problems is included. The techniques of this thesis have been developed primarily with the idea that they would be applied to the problem of describing and reasoning about distributed algorithms. The examples presented are of this kind.

1.2 An Example

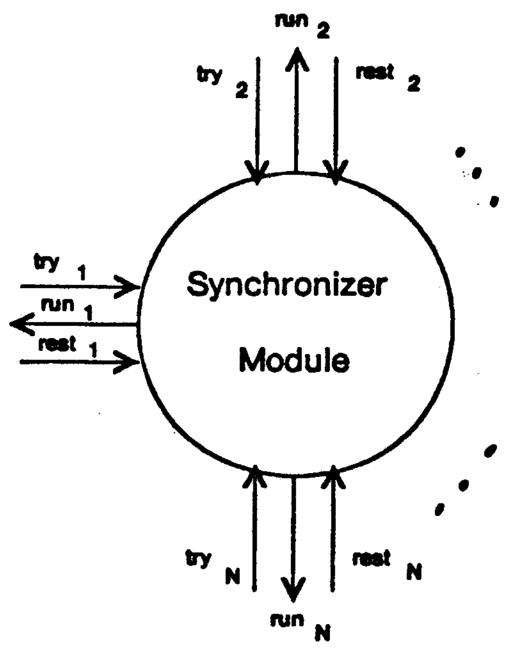
In this section, an example specification problem will be used to introduce informally the fundamental ideas about specification on which this thesis is based.

1.2.1 The Synchronizer Module

Consider the following scenario: A number of processes in a computer system require the use of a single resource to accomplish their respective tasks; however, because of limitations inherent in the resource, at most one process can be allowed to access the resource at any instant of time. To enforce this restriction, a *synchronizer module* is introduced, and the processes, which we will refer to as the *user* processes, are required to obtain permission from the synchronizer module before accessing the resource. It is the job of the synchronizer module to produce correct synchronization of the user processes' accesses to the resource. Our problem is to describe precisely the synchronizer module behaviors that are "acceptable" in the sense that they always produce "correct synchronization." This precise description is the specification of the synchronizer module.

When a user process desires to access the resource, it issues a *try* request to the synchronizer module. The user process is then supposed to wait until it receives a *run* response from the synchronizer module. When the user process is finished using the resource, it issues a *rest* response to the synchronizer module. We can capture these decisions in diagrammatic form as shown in Figure 1, in which the synchonizer module is depicted as a circle, and the possible requests and responses are drawn as arcs incident on and exiting from the circle, respectively. We assume that there are a total of *N* user processes accessing the synchronizer module, and have used a subscripted process number to distinguish the requests and responses corresponding to different processes.

Fig. 1. The Synchronizer Module



The set of all possible requests and responses for the synchronizer module can be thought of as an "alphabet" or "syntax" for describing the interaction of the synchronizer module with its environment. We call this set the *interface* of the synchronizer module, and refer to its elements as *events*. By observing the synchronizer module during an execution, we can obtain a record of the events that occurred during the execution. We call this record of event occurrences an *observation*, and assume that it takes the form of a finite or infinite sequence of events.

By fixing the interface of the synchronizer module to be a particular set of events, we determine a universe of possible observations. We next consider how to describe which observations in this universe are "acceptable." We must include in our description the idea that at most one user process at a time may access the resource. Also, we wish to require that the synchronizer module be *fair* in the sense that every *try* request by a user process is eventually answered by a *run* response, if it is possible to do so without violating the mutual exclusion property.

A natural way of describing which observations are acceptable is through the use of *conceptual states*. With this technique, we imagine that at any instant of time the synchronizer is in one of a number of possible internal states. These states may or may not have anything to do with the actual internal state of the synchronizer module; they are merely a tool for describing its observable behavior. After defining the set of initial states, we then describe for each event the preconditions required for that event to occur, and how the conceptual state of the synchronizer changes as a result of the occurrence of that event.

The conceptual state of the synchronizer module at any instant of time is a vector that tells for each user process what the synchronizer module thinks that user is currently doing with respect to the resource, based on the requests and responses that have occurred so far. The possibilities are that the user is either trying to obtain permission to access the resource (trying), is actively using the resource (running), is done using the resource (resting), or has failed to correctly follow the protocol (error).

---

1. The formal definition of observation used in this thesis is slightly more complicated than a finite or infinite sequences of events (see Chapter 2). This is done for technical reasons that are unimportant for the present, informal discussion.

Initially, the synchronizer module believes that each user process is resting. The state changes and preconditions are as follows: a *try* event for a process causes the state of that process to change to "trying" if it was previously resting, and to "error" otherwise; a *run* event for a process can occur only if that process is trying and no processes are currently running, and causes the state for that process to change to "running;" a *rest* event for a process causes the state of that process to change to "resting" if it was previously running, otherwise to "error."

A particular observation for the synchronizer module satisfies the description of the previous paragraph if to each finite prefix of the observation we can assign a conceptual state in such a way that each state change satisfies the conditions enumerated in the previous paragraph. For example, assuming there are only two user processes, the observation

$try_1\ try_2\ run_1\ rest_1\ run_2\ rest_2$

satisfies the conditions above since we can assign internal states as follows:

<resting, resting> try_1 <trying, resting> try_2 <trying, trying> run_1 <running, trying> $rest_1$ <resting, trying> run_2 <resting, running> $rest_2$ <resting, resting>.

However, the observation

$try_1\ try_2\ run_1\ run_2\ rest_1\ rest_2$

does not represent a correct functioning of the synchronizer module since

<resting, resting> try_1 <trying, resting> try_2 <trying, trying> run_1 <running, trying> run_2 <running, running> $rest_1$ <resting, running> $rest_2$ <resting, resting>,

which is the only assignment of states that satisfies the state change requirements, has the property that the precondition for the run_2 event is not satisfied by the state <running, trying>. We will use the term "history" to refer to an observation that has been annotated with states.

The state-transition description above tells us a significant amount about what are the correct observations of the synchronizer module, but it does not say everything that should be said. In particular, the requirement that every request by a user process should eventually be satisfied, if possible, is not captured by the state-transition description. Informally, the reason is that a state-transition description captures only properties of histories that are "local" in the sense that they involve only adjacent states, whereas the fairness property we would like is a "global" property that involves possibly widely separated portions of the history. If the conceptual state technique is to work, we must find some way to state global properties in a form compatible with the statement of the local properties. In Chapter 4 it will be shown how global properties can be expressed in the language of *temporal logic.*

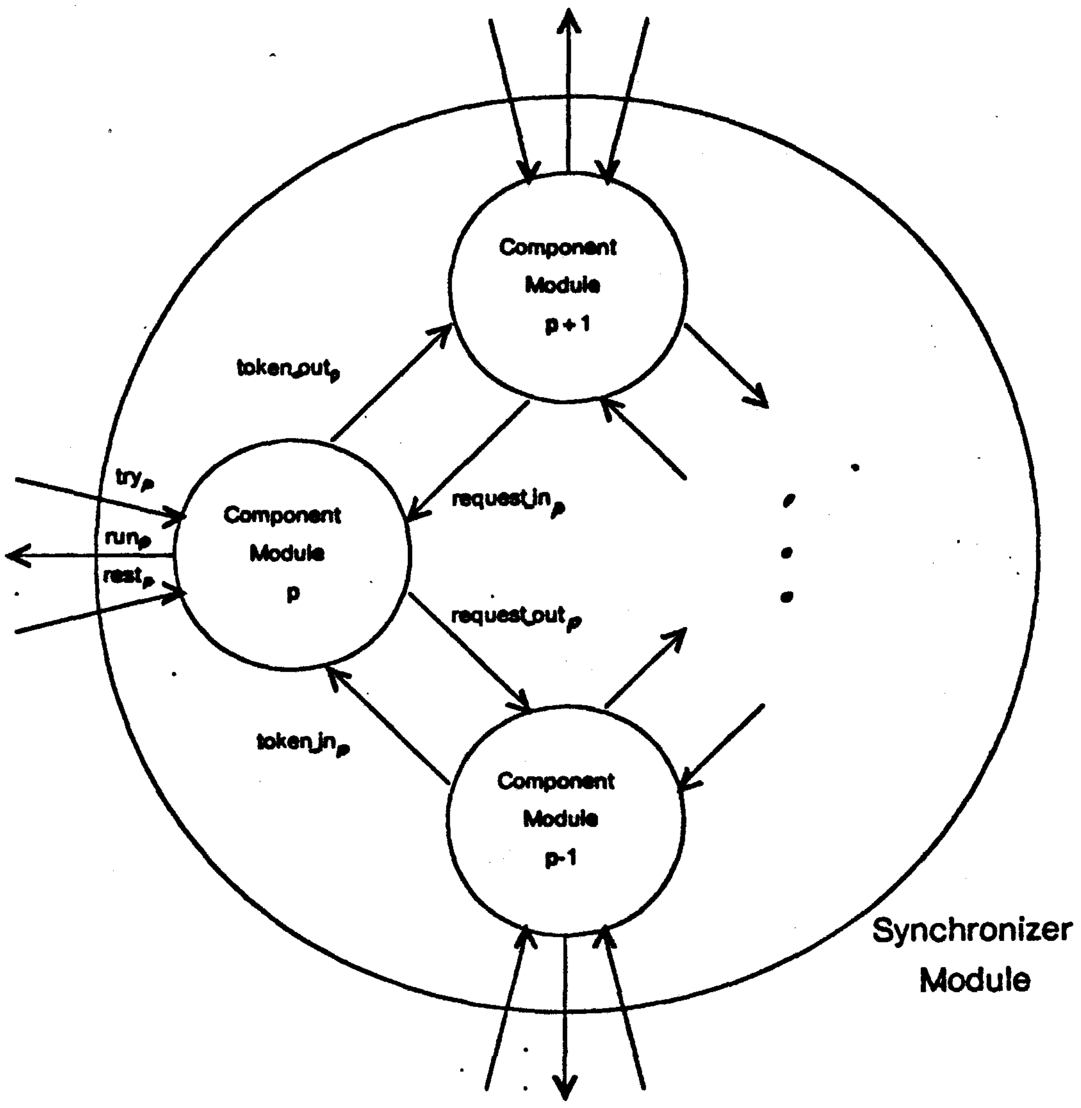
A specification of the synchronizer module via the conceptual state approach therefore consists of a state-transition description of the local properties that must be satisfied by acceptable observations, plus a description of additional global properties satisfied by such observations. A particular synchronizer module behavior is said to *satisfy* the synchronizer module specification if it contains only acceptable observations.

1.2.2 Implementation, Abstraction, and Composition

Now let us consider how the synchronizer module might be implemented. A possible organization is shown in Figure 2. In Figure 2, the synchronizer module is shown to be composed of a number of "synchronizer component" modules connected in a ring-like fashion. Each synchronizer component module interacts with exactly one user process and with its neighboring synchronizer component modules. The implementation operates as follows: There is a single conceptual token that circulates around the ring in the clockwise direction. A synchronizer component module must possess the token whenever it grants its associated user permission to access the resource. In addition to the *try*, *run*, and *rest* events with which communication with the user is accomplished, a synchronizer component module may pass the token to its clockwise neighbor with a *token_out* event, may receive the token from its counterclockwise neighbor with a *token_in* event, may request the token from its counterclockwise neighbor with a *request_out* event, and may accept a request from its clockwise neighbor with a *request_in* event.

We resolve the implementation relationship between the synchronizer component modules and the synchronizer module into two separate operations on systems: a *composition* operation, which takes a number of component modules and combines them into a larger system, and an *abstraction* operation, which takes the larger system and throws away internal details that are not of interest in the more abstract view. In the synchronizer example the composition operation takes a collection of synchronizer

Fig. 2. Implementation of the Synchronizer Module



component modules and connects them into a ring network, and the abstraction operation throws away the details of the internal communication between the component modules, saving only the events that make up the interface with the user processes.

1.2.3 Correctness of an Implementation

Suppose we are given a specification for the synchronizer module, and specifications for each of the synchronizer component modules. Each specification determines a set of behaviors that satisfy it. The implementation is "correct" with respect to these specifications if, no matter what behaviors we "plug in" for the synchronizer component modules, as long as each component behavior satisfies its specification, then the resulting synchronizer module behavior, constructed from the components via the operations of composition and abstraction, satisfies the synchronizer module specification.

1.2.4 Summary

The ideas presented in this section can be summarized as follows:

(1) Every module in a system has a well defined *interface*, which is the syntax with which it interacts with other modules in the system.

(2) An interface defines a universe of *observations*, which are records of operation that might be produced by a module with that interface. These observations constitute the possible "functionings" of the module. The set of all observations that can be produced by a particular module instance serves as the *behavior* of that module instance.

(3) A module can be specified by describing a set of "acceptable" observations. A module behavior "satisfies" such a specification if it contains only acceptable observations.

(4) An implementation of an abstract module in terms of a collection of component modules consists of a *composition operation* for combining component module behaviors to form a "composite" behavior, and an *abstraction operation* for deleting information from the composite behavior to obtain a behavior of the abstract module.

(5) An implementation is correct with respect to given specifications if, whenever we apply the composition operation of the implementation to a collection of behaviors that satisfy the component module specifications, and then apply the abstraction operation of the implementation to the resulting composite behavior, we obtain a behavior that satisfies the abstract module specification.

1.3 Outline of the Thesis

This thesis is an attempt to elaborate and make more precise the ideas illustrated informally in the previous section. In particular, an attempt will be made to answer the questions:

(1) What is an appropriate mathematical framework that adequately captures the notions of interface, observation, composition, abstraction, implementation, specification, and correctness discussed above? (Chapter 2)

(2) How can we translate, in a natural and systematic way, an intuitive understanding of the function to be performed by a module into a precise specification? (*State-Transition Specifications*, Chapter 3)

(3) Once we have obtained such a specification, how can we be sure that it says something meaningful? (*Consistency* of specifications, Chapter 5)

(4) How can we show, in a systematic way, that a particular implementation of an abstract module by a collection of component modules is correct with respect to given specifications? (*Correctness Proofs*, Chapters 3, 4, Appendix II)

(5) What general principles can we learn about how to organize specifications and proofs of correctness? (*Rely/Guarantee-Conditions*, Chapters 3, 4, Appendix II)

(6) How might the specification and proof techniques developed in this thesis be formalized to permit the use of mechanical aids? (*Event/State Algebras*, Appendix I).

This thesis is organized as follows: Chapter 2 introduces formal definitions of the notions of interface, observation, abstraction, composition, implementation, and correctness. Some of the modeling choices embodied in these definitions are discussed.

In Chapter 3, the basic definitions of Chapter 2 are used to define formally the notion of a state-transition specification. The main result of Chapter 3 is the Correctness Theorem (Theorem 3.9), which shows how the structure of state-transition specifications can be exploited to obtain a systematic method for performing correctness proofs. Secondary results of Chapter 3 (Lemma 3.11, Lemma 3.12) suggest how the proof method embodied in the Correctness Theorem can be further systematized if module liveness specifications are expressed in terms of *rely-/guarantee-conditions*.

Chapter 4 applies the theory of Chapters 2 and 3 to the synchronizer example. The complete specifications of the synchronizer and synchronizer component modules are presented, and the synchronizer implementation is proved correct with respect to these specifications. The language of *temporal logic* is used as a notation for expressing liveness properties.

Chapter 5 is concerned with finding an appropriate notion of consistency of specifications that include nontrivial liveness properties. Intuitively, a specification ought to be consistent if and only if it is satisfiable by some behavior. However, if by the term "behavior" we mean "arbitrary set of observations," then we obtain a notion of consistency that is much too liberal. To obtain stronger notions of consistency, we must restrict our attention to "realizable" or "computable" behaviors. Chapter 5 introduces a particular class of computable behaviors, the "I/O-behaviors," that is based on an underlying model of asynchronous concurrent computation called "I/O-systems." The corresponding notion of "I/O-consistency" is found to be useful for distinguishing between "obviously realizable" and "obviously unrealizable" liveness specifications. Chapter 5 develops a technique for proving state-transition specifications to be I/O-consistent and applies this technique to examples.

In Chapter 6 a kind of completeness result is proved (the Completeness Theorem, Theorem 6.4), which gives sufficient conditions under which a correct implementation has a proof by the Correctness Theorem. The statement and proof of Theorem 6.4 uses in a crucial way the existence of a "specification domain," which is a class of behaviors, like the I/O-behaviors, with certain closure properties.

Finally, Chapter 7 summarizes what has been accomplished and suggests avenues for future investigation.

Additional important material is contained in Appendices I, II, and III. Appendix I provides a formal semantics for the temporal logic language used informally in Chapters 4-6, and shows the correctness and consistency proof techniques developed in the thesis can be formalized within this language. Appendix II considers two additional examples: a distributed resource management system, and a reliable message transmission system based on the alternating bit protocol. Both of these systems are specified and proved correct using the techniques developed in the main body of the thesis. Appendix III is an index of definitions.

1.4 Related Work

The rather large body of work related to this thesis can be divided roughly into the following categories:

(1) Specification of sequential programs/abstract data types.

(2) Models of distributed/concurrent computation.

(3) Temporal logic specification techniques.

(4) Specification of communication protocols.

(5) Other distributed/concurrent system specification techniques.

Each of these categories will be discussed below. Further discussion is included at appropriate points in this thesis.

1.4.1 Specification of Sequential Programs/Abstract Data Types

Work in the area of specification of sequential programs can be classified into two categories: that concerned with the specification of the *function* to be performed by a program or program fragment, and that concerned with the specification of the *data types* manipulated by a program.

Sequential Program Function Specification

Specification of the function to be performed by a program or program fragment is a problem that must be addressed by any work on program correctness. In the sequential case, the semantics of a programming language assigns to each program fragment (statement, procedure, etc.) some mathematical object (denotation) representing the effect of executing that fragment. Typically, (see, e.g. [Jones81]) this denotation takes the form of a partial function or a binary relation on program states. A specification for a program fragment consists of some properties that must be satisfied by the denotation of that fragment.

Often function specifications are expressed in the form of Floyd/Hoare *partial correctness assertions* (PCA's) [Floyd67, Hoare69], consisting of a *precondition* and a *postcondition*, which are predicates on states. A program fragment satisfies a PCA if, whenever execution of the fragment is begun in a state satisfying the precondition, then execution will terminate only in a state satisfying the postcondition. Thus, if binary relations are used as denotations of fragments, a PCA is satisfied by any relation R such that if $\langle q, r\rangle \in R$ and q satisfies the precondition of the PCA, then r satisfies the postcondition.

Besides being convenient for specifying the function that must be satisfied by a program fragment, partial correctness assertions can be used to construct a formal deductive system for reasoning about the behavior of program fragments. For a good overview of these "Hoare logics" of programs, see [Apt81].

The partial correctness assertion technique has been generalized with some success to systems of concurrent processes [e.g. Owicki76]. However these techniques suffer from a lack of modularity in the sense that there is no notion of the behavior of a single process in isolation. Thus it is possible to specify the function of a complete parallel program, but not the behavior of its constituent processes. Although a logic of partial correctness assertions is used to prove that the behavior of a program satisfies its specification, the truth of PCA's associated with one process cannot be determined, except within the context of the PCA's for all other processes.

Partial correctness assertions are capable of expressing only safety properties of the form: "Whenever control is at point P, then relation R holds on the program variables. In general, one is interested in liveness specifications as well. For sequential programs, often the only liveness specifications of interest are statements that the program is guaranteed to terminate under certain conditions. Liveness properties of this simple form can be handled by incorporating termination into PCA's, as in Dijkstra's calculus of "weakest preconditions" [Dijkstra76], or by techniques completely outside of PCA's, such as Floyd's well-founded set technique [Floyd67]. For distributed or concurrent programs, it is almost always the case that more general liveness properties than simple termination are of interest, and these require alternative techniques.

Data Type Specification

The problem of describing the data objects manipulated by a program, especially the user-defined data objects, is usually referred to as "specification of abstract data types." There are actually two quite different problems that are addressed in the literature on abstract data type specification: the specification of *immutable* abstract data types, whose objects do not change their state during execution, and the specification of *mutable* abstract data types, whose objects have changeable state.

Specification of immutable abstract data types is the problem of describing and reasoning about static collections of values, functions, and relations. Usually a collection of interdependent immutable abstract data types is identified with the mathematical notion of a *heterogeneous algebra*, and algebras are described either axiomatically, as in [Guttag78, Goguen78, Kapur80], or via set-theoretic constructions, as in [Abrial80]. Specification of mutable abstract data types, on the other hand, can be thought of as the problem of describing and reasoning about the dynamic behavior of a collection of objects that can be manipulated using a limited set of procedures [Guttag80, Wing83]. Berzins [Berzins79] models a mutable abstract data type as a kind of *state machine*, which describes how the states of the mutable data objects evolve as a result of the invocation of the procedures.

The problem of specifying *immutable* abstract data types is not addressed by this thesis. In fact, the specification and proof techniques presented in this thesis assume as a prerequisite the ability to describe heterogeneous algebras and to perform reasoning about such algebras once they have been described. On the other hand, the problem of specifying *mutable* abstract data types can be viewed as a special case of the general problem of module specification considered in this thesis, by thinking of a mutable abstract data type in terms of a "type manager" module, which encapsulates the objects of the data type and which performs manipulations on these objects in response to requests by the environment. Viewed in this way, the purpose of a mutable abstract data type specification is to describe the correct "observations" for the type manager module. The notion of observation appropriate here is that of a history of "events," where each event records either a request for the type manager to perform some manipulation on the objects, or a reply indicating the results of some previously requested manipulation.

1.4.2 Models of Concurrent Computation

Quite a number of models have been proposed for investigating concurrent and distributed computer programs [Brock83, Hoare81b, Hoare81a, Greif75, Hewitt77, Kahn74, Keller76, Lynch81, Pratt82, Rounds81]. In this thesis as well, specific assumptions are made about how to model the behavior of such systems. It is necessary to make these assumptions to reach a point at which concrete example specifications can be written and correctness proofs performed. However, a conscious effort has been made to assume no more structure than is necessary for the results of this thesis. An attempt has been made to identify a few fundamental concepts that are required of *any* model, if it is to serve as a semantic foundation for the theory of specification developed here.

The fundamental concepts identified in this thesis are the notions of *interface*, *observation*, *behavior*, *abstraction*, and *composition*. These concepts, which have already been informally discussed, are given formal definitions in Chapter 2. In this section, we will briefly review the features of a number of extant models of concurrency and attempt to identify the notions of event, interface, observation, behavior, abstraction, and composition used here with corresponding notions in each of the models. We will also be interested in whether each model is suitable as a semantic basis for a specification language -- in particular, whether the model can model is useful for specifications involving liveness properties.

Kahn-MacQueen Processes

A rather elegant model of concurrent computation is the *stream processing* model of Kahn [Kahn74] and Kahn and MacQueen [Kahn77]. In this model, a process communicates with its environment through a collection of named *channels*. A process uses each channel either as an *input* channel or an *output* channel, but never as both. During execution, a process can read input values from input channels and emit output values on output channels. We can imagine observing a process throughout an entire execution and recording the sequence of values transmitted on each channel. Such a sequence of values, which can be either finite or infinite, is called a *stream*. A process is modeled by a *continuous function* from tuples of input streams to tuples of output streams. The notion of continuity used here is derived from the fact that streams under the prefix ordering form a partially ordered set which is complete under limits of increasing chains. Processes are *deterministic* in the sense that to each input tuple *I*, there is precisely one output tuple *O* that can be produced by a particular process, when that process is supplied with input *I*. This is a consequence of the fact that processes are modeled by functions.

In the stream processing model, the sets of input and output channels used by a process serve as the interface of that process. The role of an observation of a process is played by a pair <*I*, *O*>, where *I* is a tuple of streams corresponding to the input channels, and *O* is a tuple of streams corresponding to the output channels. The usual identification of a function with its graph permits us to view a process behavior *f* as the set of all observations of the form ⟨*I*, *f*(*I*)⟩.

A *process network* describes how to compose a collection of processes to form a composite system. Formally, a process network defines a kind of fixed point construction that maps a collection of component process behaviors to a behavior for the composite network. These fixed point constructions comprise the composition operations. The composition operations used by Kahn and MacQueen include features of both composition and abstraction as defined here, in the sense that once two processes have been connected by a communication channel, the stream of values transmitted over that channel is no longer of interest, and is ignored.

The Kahn/MacQueen model is unsuitable for the purposes of this thesis because it is incapable of representing processes with nondeterministic behavior.

Nondeterministic Process Nets

There have been several attempts to generalize the stream processing model of Kahn and MacQueen to incorporate nondeterminism. One such attempt is reported by Brock in [Brock83] (superseding the earlier version [Brock81] by Brock and Ackermann), where references to other attempts are given. In [Brock83], it is shown that the straightforward attempt to generalize the model of Kahn and MacQueen by permitting process behaviors to be *relations*, rather than functions, is doomed to failure. Intuitively, the reason is that the behavior of nondeterministic processes depends, in general, on the relative orders in which inputs are received and outputs produced. In essence, Brock's approach is to replace the ⟨*I*, *O*⟩ observations used by Kahn and MacQueen by *scenarios*. Scenarios include, in addition to the streams of values transmitted on each of the channels, a partial ordering that records some of the information concerning the temporal order in which values were transmitted. The behavior of a process is defined to be the set of all scenarios that the process can produce in its various executions. Brock shows how composition operations on scenario sets can be defined, in analogy to the operations on continuous functions defined by Kahn and MacQueen.

Pratt's [Pratt82] "repackages" Brock's model into a general framework for modeling processes and their composition, in which the behavior of a process is represented by the set of all *traces* (partially ordered multisets of events) it is capable of producing. As in the models of Kahn/MacQueen and Brock, the interface of a process can be identified with the set of all events in which the process can participate. The notion of trace plays the role of an observation. The notion of the *restriction* of a trace to a subset of its events is used to define composition of process behaviors. Restriction mappings on traces play essentially the same role in Pratt's model as decomposition maps play in the model of this thesis.

The models of Brock and Pratt admit the possibility of infinite scenarios or traces, and therefore do not *a priori* rule out the possibility of modeling processes that satisfy nontrivial liveness properties. However, this possibility is not addressed by either Brock or Pratt. Since we are interested in modeling processes with liveness properties, the models of Brock and Pratt are not suitable in their present state of development.

Communicating Sequential Processes

An important class of models of concurrency [Francez79, Hoare81a, Hoare81b, Rounds81] has been developed through attempts to give a formal semantics to the language of "Communicating Sequential Processes" (CSP) defined in [Hoare78]. In each of these models, the behavior of a process describes the *traces* (finite sequences of *communication events*) in which the process is willing to participate as it executes. The set of all events in which a process can ever participate plays the role of the interface of that process. The notion of a trace plays the role of an observation. Although the particular notion of process behavior is different for different models, each of the models of CSP contains a collection of algebraic operations on process behaviors, which are used to define the meaning of the various constructs of CSP. In particular, each model has some sort of "restriction" or "hiding" operations, which cause events to be deleted from a process behavior, and some sort of "relabeling" operations, which allow events of a process to be renamed. These operations are used for essentially the same purpose as the abstraction operations used in this thesis. Each model also has one or more "composition" operations (composition by intersection, composition by interleaving, or a mixture of the two) corresponding to the composition operators of this chapter, whose effect is to combine process behaviors in various ways.

The important considerations for models of CSP derive from a feature peculiar to that language. A CSP process can *refuse* to communicate with its environment. If a CSP process refuses to perform any of the communications offered by its environment, then *deadlock* is the result. The different definitions of process behaviors in the various models of CSP arise from attempting to deal with (or to ignore) the subtleties of refusals and nondeterminism.

In [Hoare81b], a process behavior is a prefix-closed set of traces, which can be viewed equivalently as a behavior of the kind defined in this thesis. There are operations in [Hoare81b] for deleting and renaming the events of a process. These operations are examples of the abstraction operators used in this thesis. Process behaviors are composed by the parallel composition operator $\|$, which is defined as follows: If A is the behavior of a process with interface E and B is the behavior of a process with interface F, then $A \| B$ is the set of all traces u formed from events in $E \cup F$ such that the restriction of u to E is in A and the restriction of u to F is in B. This notion of composition is a particular example of the composition operators defined in this thesis.

Hoare, Brookes, and Roscoe [Hoare81a] extend the work of [Hoare81b] to deal with the problems of refusals and nondeterminism. They do this by permitting behaviors to be more highly structured objects than just sets of traces. In particular, a behavior is a set of pairs $\langle s, X\rangle$, where s is a trace, and X is a set of events that can be refused by the process after the trace s has been produced. Although they use a single universal set of events for all processes, we can imagine designating the set of all events that actually appear in a process as the interface of that process. As in the model of [Hoare81b], traces play the role of observations. There are "concealment" operators for deleting events, and "inverse image" operators that permit renaming of events. There are no "direct image" operators, apparently because they are not as well behaved as the inverse image operators. Two kinds of parallel composition operations are defined: composition by *intersection*, in which events of the component processes are connected, and composition by *interleaving*, in which the events of the components remain independent.

Rounds and Brookes [Rounds81] attempt to justify and extend the work of [Hoare81a] in the following way: A definition of process behaviors is made that includes somewhat more information than that of [Hoare81a], and is based on supposedly more fundamental intuitive considerations. A number of algebraic operations, including composition and abstraction, are defined on behaviors. A notion of "observable equivalence" of behaviors is defined, and is shown to be a congruence. The quotient of the algebra of behaviors with respect to this congruence is then shown to be isomorphic to the model of [Hoare81a], thus providing evidence that this model exactly captures the externally observable properties of processes.

There seem to be problems associated with the use of models of CSP as a semantic basis for specification languages. These problems center around the following two questions: (1) Do traces represent a "complete" record of execution of a process, or simply some finite portion of such a record? (2) What is the meaning of a liveness specification such as "eventually event *a* will occur," if a process can be placed in an environment that refuses to permit the occurrence of event *a*?

With respect to question (1), it is difficult to see how the designers of the CSP models could have intended traces to represent complete observations. This is because in general a complete observation will be infinite, but the CSP models provide no method for extracting infinite traces from behaviors. Without a distinction between complete and incomplete observations, we have no way to determine whether a particular CSP process satisfies a liveness specification. It is clearly ridiculous to require that a specification such as "eventually event *a* will occur" be satisfied by all "incomplete" as well as all "complete" observations.

Question (2) arises from a desire to "assign the blame" for an unsatisfied liveness specification, either to a process or its environment. If a process can always be placed in an environment that can prevent the occurrence of event *a*, then the only reasonable conclusion we can draw is that the specification "eventually *a* will occur" is too strong (i.e. inconsistent). However, it is not clear how to weaken such a specification so that it can be regarded as consistent.

The above problems associated with the models of CSP have been avoided here as follows: First, it is assumed here that the observations in a behavior represent *complete* records of execution. Second, we accept the obvious conclusion that the specification "eventually *a* will occur" is inconsistent with respect to a model (such as the model of [Hoare81b]) that admits the possibility of refusals. Instead of trying to find ways to weaken specifications like this so that they can be regarded as consistent even in the face of refusals, though, we construct a model in which refusals are not allowed. This is the idea behind the I/O-behaviors constructed in Chapter 5 of this thesis.

Calculus of Communicating Systems

Rather similar to the models of CSP discussed above is the "Calculus of Communicating Systems," (CCS) of [Milner80]. As in CSP, the notions of a communication event and a sequence of communication events are the fundamental concepts for describing the behavior of a process. The role of a process interface is played, in CCS as in CSP, by the set of communication events in which the process is capable of participating. The CCS notion of an observation is a sequence of events; in contrast to CSP, CCS admits the possibility of infinite observations.

To represent the behavior of a process, Milner introduces the notion of a *communication tree* whose paths represent all possible complete histories of communication for a process. In a communication tree there can be multiple arcs emanating from a single node, labeled with with the same communication event, and arcs can be labeled with the special symbol τ, which represents an internal action of a process not associated with any communication event. Communication trees therefore contain more information about a process than just a simple set of traces. In fact, communication trees contain more information about a process than can be detected through composition with other processes. Milner addresses this problem by defining several notions of "observable equivalence" of communication trees, and shows that these relations are congruences for an algebra of processes whose operations include operations of composition and abstraction. He suggests that the class of process behaviors be obtained by forming the quotient of the algebra of communication trees with respect to one of these congruences. He is unable to reach a conclusion, though, as to which of the congruences is "best," or to give explicit characterizations (not involving quotient constructions) of the quotient algebras.

Although communication between two processes in CCS, as in CSP, is *synchronized* in the sense that it is represented by the simultaneous occurrence of communication events for the participating processes, communication in CCS is unlike that in CSP in the sense that a CCS process cannot prevent another process from performing an event. This is because the definition of the composition operation in CCS states that, if process A can perform an event a, and process A' can perform the "complementary" event a', then the composition $A \parallel A'$ can perform *either* a, *or* a', *or* the communication represented by the simultaneous occurrence of both a and a'.

The fact that observations can be infinite in CCS raises the question of whether it is possible to define CCS processes that satisfy interesting liveness properties. However, it seems that this possibility is ruled out by Milner's composition operation. Milner's composition operation is "unfair" in the sense that there are paths in the communication tree corresponding to the composition of two processes along which only one of the component processes gets to run. This means that no process can satisfy a specification of the form: "eventually a will occur," in an environment that has the capability of producing an infinite observation.

Actors

One of the earlier event-based models of computation is the actor model [Greif75, Hewitt77]. An actor system consists of a collection of primitive computing agents (actors), that communicate by passing messages. A computation for an actor system is a partially ordered set of events, where an event marks the arrival of a message at its target. Receipt of a message *activates* the target actor, and may cause additional messages to be issued. The partial order represents a kind of temporal "precedes" relationship between events, formed by taking the transitive closure of the union of the "causes" relation and the "arrival" ordering, the latter of which linearly orders all events with the same target. Hewitt and Baker [Hewitt77] postulate certain laws that must be satisfied by the various orders.

The actor model was originally applied [Greif75] to the specification of synchronization problems such as the mutual exclusion and readers/writers problem. The specifications are written as axioms that constrain the possible computations of a system. The language used, although not formally defined, is essentially a propositional calculus in which the propositions are of the form "$e \rightarrow e'$," which means that event e must precede event e' in any computation of a system satisfying the specification. Although no notion of state was used in the specifications, the language has nevertheless sufficient expressive power to handle several important examples.

Subsequent work concentrated on applying the actor model to the specification of more complex systems, both distributed and centralized [Yonezawa77]. In contrast to the work of Greif, Yonezawa's specifications have a decidedly state-transition flavor, and although proponents of the actor model consistently argue that global state is not a well-defined notion for distributed systems, the "situations" used in Yonezawa's correctness proofs appear to be just such global states.

In the actor model, the notion of an actor is generally defined by informal axioms and description, which are insufficient to answer the question: "What *is* an actor?" We must know the answer to this question if we wish to obtain a meaningful notion of the collection of all actors that satisfy a given specification, and to show the validity of rules for deriving consequences of specifications of actor systems. The question of what actors are has only recently been dealt with by Clinger [Clinger81], who defines actors and their computations directly in terms of set-theoretic constructs. It is interesting to note that, although actor enthusiasts like to point out that viewing computations as partially ordered sets of events captures "true" concurrency better than linearly ordered computations, Clinger shows that the laws of Hewitt and Baker are in fact equivalent to the existence of a global linear ordering of events in a computation.

To relate the actor model to the model used in this thesis, we can attempt to identify notions of interface, observation, behavior, abstraction, and composition in the actor model. There seems to be no obvious notion of the interface of an actor. The notion of a partially ordered set of events plays the role of an observation. Roughly speaking, Clinger defines the behavior of an actor to be a function that describes the actor's response (i.e. its state change and message transmissions) to the receipt of a message. Although we can imagine composing a collection of independent actors into a composite system, there seems to be no formal notion in the actor model corresponding to such an operation. As mentioned above, the existence of the arrival ordering prevents the definition of an abstraction operation.

The actor model has certain defects that render it unsuitable for a theory of specification. The major difficulty is that the actor model does not support abstraction of systems in a uniform way. There are notions of an actor and a system of actors, but no way to view a system abstractly as a single actor. The artificial "arrival ordering," imposed on all events that occur at a single actor, is the primary feature that prevents abstraction from being defined in a reasonable way. Another reason is the fact that every message must contain the name of its target actor, since this means that it is never possible to completely suppress the internal structure of an actor system.

Lynch/Fischer Processes

In the model of distributed computation proposed by Lynch and Fischer [Lynch81], the primitive objects are *variables* and *processes*, and *systems of processes*. A variable is a mailbox-like container for values, and a process is a kind of state machine that can perform input and output on variables. A system of processes consists of a collection of processes that communicate through variables. The variables of a system of processes are partitioned into *external* and *internal* variables. There is a kind of composition operation that combines a collection of systems of processes to form a larger system. There is also a kind of abstraction operation that transforms some of the external variables of a system into internal ones.

A correspondence between Lynch and Fischer's model and the model of this thesis can be established, if the notion of an event is identified with Lynch and Fischer's notion of a "variable action." A variable action describes the change in the value of a variable resulting from a single execution step. The interface of a system of processes is the set of all variable actions it can perform. The behavior of a system of processes is, as Lynch and Fischer define, the set of all finite and infinite sequences of variable actions the system is capable of performing. To view Lynch and Fischer's operation of composition of systems of processes as a special case of the composition operators defined here, it is necessary to account for the requirement that the actions on a single variable in the computation of a system have consistent values. This is easily accomplished if variables are thought of as active entities with an interface and a behavior. The interface of a variable is the set of all variable actions that can be performed on it. The behavior of a variable is the set of all finite and infinite sequences of variable actions in which the value read in each variable action equals the value written in the immediately preceding variable action.

In terms of modeling power, the model of this thesis and that of Lynch and Fischer appear equivalent. Lynch and Fischer's model is certainly capable of handling nondeterminism and liveness properties. The main advantage of the model of this thesis over that of Lynch and Fischer is that the former contains fewer primitive concepts. It is not necessary to draw distinctions between variables, processes, and systems of processes, and the definitions of composition and abstraction are simplified by avoiding these distinctions.

1.4.3 Temporal Logic Specification

Several authors [Hailpern80, Lamport83, Schwartz81] have proposed the use of temporal logic as a specification language and a vehicle for expressing correctness proofs. The use of temporal logic as a specification language evolved gradually from its use as an assertion language, that is, as a language for expressing properties of program executions [Pneuli77, Lamport80]. There is a subtle difference, though, between the semantics appropriate for temporal logic used as an assertion language and temporal logic used as a specification language. This difference, which has not been explicitly addressed in the literature,can be summarized as follows: Whereas temporal formulas as assertions express properties of *single* computations of a *fixed* program, temporal formulas as specifications express properties of the *set* of computations of an *undetermined* program. Stated another way, whereas a model for a temporal formula used as an assertion about a fixed program is a single computation of that program, a model for a temporal formula used as a specification is the set of all computations that can be produced by some program. This distinction has important ramifications for what notion of consistency is appropriate in each case. A temporal formula used as an assertion about the computations of a fixed program is consistent if and only if there exists a computation of that program that satisfies the formula. A temporal formula used as a specification is consistent if and only if there exists a program, all of whose computations satisfy the formula.

Another important issue that is not addressed explicitly in literature on temporal logic specification is the ability to specify a single module in isolation from particular program context.[1] The notion of a program module satisfying a specification in isolation *must* be meaningful if specifications are to effect the beneficial separation between module use and implementation. Since extant work does not include the notion of the meaning of a specification in isolation, there has been no discussion of the following important question: How can we combine independent module specifications to perform

1. Recent work [Barringer83], performed independently of the work described in this thesis, has begun to address some of the same issues, in particular: (1) temporal specifications express properties of *sets* of computations, rather than single computations, (2) specifications should have meaning that is independent of an enclosing context.

a proof of correctness? In particular, in what common language can the proof of correctness be expressed, and what deductions in this language are sufficient to imply the correctness of the implementation?

Among the papers on temporal specification of concurrent program modules, the approach developed by Lamport [Lamport83] contemporarily with work on this thesis, results in specifications that appear most similar to the state-transition specifications described here. In Lamport's approach, a specification consists of three parts: (1) A list of *state functions*, which define salient features of the program state; (2) A list of *initial conditions*, which represent assumptions on the initial values of the state functions; (3) A list of *properties*, which constitute the main body of the specification, and which can be viewed as standing for a collection of temporal logic formulas. The properties are of two kinds: *safety properties* and *liveness properties*. Safety properties describe the state transitions that are *permissible* for a program satisfying the specification, and liveness properties describe situations under which transitions are *required*.

The way one writes a specification in Lamport's approach is quite similar to the way one writes state-transition specifications as described in this thesis. At the semantic level, though, Lamport's approach seems rather different. The difference can be summed up briefly as follows: In Lamport's work, specifications for program modules play the role of assertions about the computations of a complete program in which the module appears. Whether or not a particular program module satisfies a specification can only be determined in such a context. In the framework presented in this thesis, whether a program module satisfies a specification can be determined without reference to any contextual information.

The meaning of the state functions used in Lamport's approach is obscure. Lamport says that state functions in a specification "should describe information that must be contained in the program state of any real implementation." This statement apparently implies that the value of the state functions is part of the observable behavior of the module being specified, and in this sense is just as important a part of a module specification as the relationship between the arguments passed and results returned from an invocation of an operation on the module. Choosing state functions that provide too detailed a view of the internal operation of a module can result in overspecification, since an implementer wishing to satisfy the specification is constrained to include enough information in the state so that the state functions can be defined.

This thesis resolves the problem of overspecification by introducing the notion of an *interface*. By defining a module interface, one fixes a particular class of module instances (i.e. the behaviors of that interface) which serves as a domain of discourse for the temporal specifications. In this thesis, a module interface is a set of events. An interface does *not* contain any notion of module state. States are used merely as a device for increasing the expressive power of the specification language to permit the desired properties of observations to be expressed in a convenient and natural way. Since states are not part of the module interface, the state set in a state-transition specification can be chosen on the basis of convenience, without danger of overspecification.

Schwartz and Melliar-Smith have also proposed the use of temporal logic as a specification language. In [Schwartz80], specifications are developed for the alternating bit communication protocol. Appearing in these specifications are uninterpreted symbols such as "InQ" and "OutQ." These symbols, like the state functions used by Lamport, are evidently intended to refer to portions of the state that must be identifiable in any program satisfying the specifications. Schwartz and Melliar-Smith present collections of temporal axioms which they claim completely characterize the send and receive processes supporting the alternating bit protocol. There is little basis for this claim, since it is impossible to determine what a process is, much less determine whether the specifications characterize a particular process or class of processes.

The axioms presented by Schwartz and Melliar-Smith involve complicated derived temporal operators such as "latches-until," which make the resulting specifications quite difficult to understand. The specifications have an *ad hoc* flavor, and it is difficult to obtain insight into how specifications for different examples would be obtained. In contrast, the state-transition approach discussed in this thesis suggests a systematic way of proceeding from an intuitive conception of the desired module behavior to a precise specification. Schwartz and Melliar-Smith present no proof that their send and receive process specifications correctly implement the service specification for the alternating bit protocol. Experience gained from the examples presented in this thesis suggests that specifications that have not been used in a proof of correctness are quite likely to contain errors.

Hailpern and Owicki [Hailpern80] propose a style of temporal logic specification that is different from the styles of Lamport and of Schwartz and Melliar-Smith. Hailpern and Owicki also use the alternating bit protocol as an example to illustrate their approach to specification. In addition to symbols representing components of the internal states of processes in the system, Hailpern and Owicki introduce the notion of a *history variable*, whose value at any instant of time represents the entire history of communication between two processes up until that instant of time. They state explicitly that history variables are simply a descriptive tool, and are not intended to be implemented. History variables appear to be quite useful for writing high-level, nonprocedural specifications. For example, the safety properties satisfied by a transmission line could be expressed by stating that the history of messages delivered is always a prefix of the history of messages sent.

The state-transition approach to specification presented in this thesis takes the history variable idea to its logical conclusion, by allowing arbitrarily structured history information (in the form of states), to be introduced into a specification, together with operations for manipulating this information. This can be done differently for each specification, without change to the underlying semantic model. For example, the specification of the reliable transmission module presented in Chapter 6 uses the notion of the history of all messages input to the reliable transmission module. In the specification of the send protocol module in Chapter 6 it is convenient to define the notion of "the history of all messages for which acknowledgements have been received." This history is a subhistory of the history of *all* messages transmitted by the send protocol module, and would not be directly accessible in the model of Hailpern and Owicki.

1.4.4 Specification of Communication Protocols

The problem of specification of communication protocols has received a good deal of attention, and can be viewed as a special case of the more general problem, investigated here, of specification of modules in a distributed system. Two surveys of the protocol specification literature, written from different vantage points, can be found in [Sunshine78] and [Hailpern81].

Of the numerous papers on protocol specification and verification, that of Bochmann [Bochmann78] appears to be most directly relevant to this thesis. Bochmann models a system as a collection of finite-state machines that affect each other through coupled state transitions. This is highly analogous to the definition, given here, of composition of behaviors by identifying events. Bochmann also has a notion of abstraction by ignoring uninteresting transitions, which matches the concept of abstraction of behaviors used here.

Schwabe [Schwabe81a, Schwabe81b] exploits the analogy between the instantaneous state of a communication protocol and a value of an abstract data type, to translate state-transition specifications of protocols into equational axioms that define an abstract data type. This translation enables him to verify correctness properties of communication protocols using an automated verifier (AFFIRM) originally intended for proving properties of abstract data types. However, only certain kinds of correctness properties can be stated and proved using his technique. In particular, liveness properties cannot be handled. Schwabe pays little attention to the semantics of his specifications, leaving some ambiguity as to what objects satisfy a specification, and what consitutes correctness of a protocol.

It is interesting that the notions of hierarchy and modularity of systems, and the prerequisite concept of the interface of a system with its environment, are much more prominent in the literature on protocol specification than they are in the literature on specification in general. In protocol specification, a system is viewed as a nested set of layers: the bottom level corresponds to the communication hardware, and each layer provides an abstract service to the next higher layer. The top level implements the service provided to the "end user." Typically the service provided by a level can be viewed as an abstract communication network connecting two users, which often have an asymmetric sender/receiver relationship. Higher levels of abstraction are implemented by interposing *protocol processes* between the users and the communication service provided by the next lower level. The interface between the users and a service comprises the set of operations (e.g. open connection, send message), they can perform. A distinction is drawn between the specification of an abstract service provided to a user (the *service specification*) and a description of the protocol processes (the *protocol specification*).

There are only a few specific correctness properties of interest for communication protocols: freedom from deadlock, completeness (i.e. definedness of the protocol in every reachable state), progress, and stability in the face of unexpected perturbations of the protocol. These properties are certainly also of interest for more general kinds of distributed systems. All verification techniques in the communication protocol literature are ultimately based on representing the protocol processes and abstract communication media as finite-state machines, constructing a combined state-transition graph for the implementation, and performing various analyses on this graph. The state-transition approach to specification and verification is a natural generalization of this technique. It should be noted, however, that the machines used in the state-transition specifications in this thesis are not necessarily finite-state, and that reachability analysis of a system is performed by proving predicates to be invariant, rather than by explicit construction of the combined state-transition graph. This means that the proof techniques discussed in this thesis need not be subject to the combinatorial explosion problem often referred to in the literature on protocol verification.

1.4.5 Other Concurrent System Specification Techniques

Chen [Chen81, Chen82] develops a concurrent system specification language called EBS (Event-Based Specification Language), and gives specifications for a number of examples, including the alternating bit protocol. The EBS language can be thought of as a generalized version of the language used in [Greif75] to specify various synchronization problems. An EBS specification expresses properties of an event history, which is a partially ordered set of events. The EBS notion of an event history corresponds to the notion of an observation used in this thesis.

Chen's work seems to be motivated by a number of the same concerns that motivated this thesis. In particular, Chen discusses the distinction between the user's view and the designer's or implementer's view of a system, and introduces a notion of interface to capture the way in which a system interacts with its environment. In Chen's approach, a module interface consists of a collection of *ports*. There is a notion of module interconnection by identifying ports, which is reminiscent of the composition operations used in this thesis. Chen's work does not, apparently, include a notion of behavior, or the idea that a module specification has meaning except with respect to a complete program context. Chen does not have a semantic definition of the correctness of an implementation from which the soundness of proof techniques can be derived. Rather, the notion of correct implementation seems to be identified with the notion of logical consequence.

An interesting property of Chen's specifications is that they tend to be "orthogonal." An orthogonal specification is a specification that is composed of a collection of independent subspecifications. For example, Chen defines a number of different properties of a reliable transmission system, such as "no loss of messages," "no duplication of messages," and "no erroneous messages." It is not obvious how the state-transition technique presented in this thesis could support the writing of specifications with a comparable orthogonality property.

The Gypsy system [Good79, Good82] has some capability for the specification and verification of distributed systems. In the Gypsy model, a distributed system is viewed as a collection of independent processes that communicate through message buffers. Specifications of the communication function performed by a process are expressed in terms of properties of "buffer histories," which represent the sequences of messages transmitted on, or received from message buffers. Gypsy seems capable of handling only safety properties.

Correctness proofs in Gypsy are performed by deriving a collection of verification conditions from annotated program text, and then proving the validity of these verification conditions using a semi-automatic theorem prover. Evidently the validity of the verification conditions is taken as the definition of correctness; the literature shows no attempt to justify the sufficiency of the verification conditions in terms of any fundamental model of computation. Reasoning about the behavior of a system of processes in Gypsy is done in terms of relationships between buffer histories. The approach appears similar to Hailpern and Owicki's history variable approach.

An outgrowth of the Gypsy work is the work of DiVito [DiVito82], which is concerned with the description and mechanical verification of communication protocols. DiVito's specifications contain liveness properties only, and are expressed in a decision table style that captures much the same information as the definitions of state-transition relations presented in this thesis. The purpose of DiVito's work seems to be to quickly reach a point at which experimentation with mechanical verification is possible. His focus is primarily on linguistic issues, rather than their semantics.

Lansky and Owicki [Lansky83] have developed a language, called GEM, for the specification and verification of properties of concurrent systems. The underlying model of computation is an event-oriented model similar to the actor model [Greif75, Hewitt77], in which a computation of a system is represented as a set of events plus various relations on this set. The *enable* relation captures the notion of necessary temporal precedence, or causality, between events. The *element* partial ordering captures the notion of incidental temporal precedence, where one event precedes another because they happen to occur at the same point in space. The *temporal* partial ordering is the transitive closure of the union of the enable relation and the element ordering. Besides the notion of an event and the relations on events discussed above, GEM includes a number of additional primitive notions. An *element* corresponds to a locus of activity or point in space. A *group* is a set of elements and other groups, which is used to collect semantically related objects. *History sequences* are certain increasing sequences of computation prefixes, and are used as a domain of interpretation for temporal logic formulas. *Threads* are a mechanism for dynamically grouping a sequence of related events.

The issues considered by GEM seem largely orthogonal to those examined in this thesis. The design of GEM seems to have been motivated primarily by a desire to describe, within a common framework, the semantics of a number of primitives of concurrent programming languages. For example, monitors and the CSP communication primitives are discussed. In contrast, this thesis is not concerned with the description of programming language primitives, although this is a problem that must ultimately be addressed. A GEM specification describes constraints on computations of a single program, whereas in this thesis a specification is viewed as describing constraints on the entire set of computations of an undetermined program. GEM apparently does not include any notion of behavior, composition, or abstraction.

Yonezawa [Yonezawa77] develops techniques for the specification and verification of parallel programs, based on the actor model of computation. The central concepts used in these techniques are the notions of a *conceptual state*, and a *situation*. A conceptual state is a summary of the past communication history of an actor, and corresponds closely to the conceptual states used in the state-transition specifications of this thesis. A situation assigns a conceptual state to each actor in a system, and is used in verification in much the same way as the state of the "composite machine" is used in this thesis (see Chapter 3). The notion of an implementation invariant appears in Yonezawa's work, and plays roughly the same role there as it does in this thesis. Yonezawa's model seems to incorporate a notion of hierarchy of abstraction, in the sense that it is possible to view a system both at a more detailed level, where there is a larger collection of events and more detailed states, and at a less detailed level, where only a subset of the events is considered and less information is contained in the states.

Yonezawa's specifications look very much like the definitions of state-transition relations used here, in the sense that a specification describes, for each possible event, a precondition on the state that must hold for an event to occur, and a postcondition that describes the state that results after the event occurrence. The semantics of the event/precondition/postcondition triples used by Yonezawa seems to differ from their counterparts in this thesis, in the sense that if the precondition of an event ever holds, then that event must eventually occur. Thus, Yonezawa's formalism appears, to a certain extent, to be capable of expressing liveness properties.

There are three major deficiencies with Yonezawa's work, which are improved upon in this thesis:

(1) The semantics of Yonezawa's specifications are defined informally in terms of the actor model, whose precise definition is somewhat obscure. It is therefore not possible to address rigorously the question of what constitutes correctness in Yonezawa's model, and to show that his proof techniques suffice to prove correctness.

(2) The actor model lacks a useful notion of modular decomposition. In particular, there is no reasonable way to view a system of actors as a single actor.

(3) Yonezawa's techniques can handle only a very limited form of liveness property in specifications and proofs; namely, those of the form: "If the precondition of an event holds, then eventually that event must occur."

2. Framework for a Theory of Specification

The purpose of this chapter is to construct a framework of definitions that is suitable as a foundation for a theory of specification. We present and motivate formal definitions of the notions, discussed informally in Chapter 1, of "interface," "observation," "abstraction," "decomposition," "implementation," and "correctness."

2.1 Interfaces, Observations, and Behaviors

An *event* is an observable instantaneous occurrence during the operation of a computer system. If one were to examine a particular computer system in microscopic detail, the events of a system could be identified with physical events, such as voltage changes on signal lines. However, we are generally not interested in such a large amount of detail, and instead regard large classes of physical events as equivalent and indistinguishable. Examples of such equivalence classes are: the event in which process A submits a message to a transmission system for delivery to process B, the event in which the variable x is set to three, and the event in which the synchronizer module receives a *try* request from user process p.

The first step in modeling a particular system is to identify and classify the interesting instantaneous occurrences. As a result of this procedure, we associate with each system and each particular level of abstraction at which the system is to be viewed, an "interface," which represents the set of all possible instantaneous occurrences of interest at the given level of abstraction, plus a single element λ, which represents all uninteresting occurrences. Lower levels of abstraction (those that incorporate more detail) are characterized by larger interfaces, corresponding to finer classifications of the instantaneous occurrences, whereas higher levels of abstraction are associated with smaller interfaces, corresponding to coarser classifications.

Definition - An *interface* is a structure $\langle E, \lambda_E, ...\rangle$, where E is a set whose elements are called *events*, λ_E is a distinguished element of E called the *null event*, and the ellipsis indicates that further structure may be present. ∎

We use the symbol E to denote both the entire structure and the underlying set of events. When the interface E is clear from the context, we will omit subscripts, writing λ instead of λ_E.

In general, an interface E will have additional structure besides the distinguished element λ_E. For example, in Chapter 5 we will be concerned with interfaces of the form $\langle E, \lambda_E, \mathrm{In}_E, \mathrm{Out}_E \rangle$, where In_E and Out_E are subsets of E called the sets of "input events" and "output events," respectively. Except for the material in Chapter 5, the only structure required is the existence of the distinguished null event λ_E.

If E is an interface, then let E^* denote the set of all finite strings, and E^∞ the set of all finite and infinite strings, on the alphabet $E - \{\lambda_E\}$. It is convenient to view E as a subset of E^* and E^∞, where the element λ_E of E is identified with the unique string of length zero, and each non-λ element e of E is identified with the corresponding string e of length one.

In the synchronizer example, the interfaces are defined as follows. Let Proc be the set of user processes. The synchronizer module has interface $E^{SM} = \{\mathrm{try}_p, \mathrm{run}_p, \mathrm{rest}_p : p \in \mathrm{Proc}\} \cup \{\lambda\}$. A synchronizer component module has interface $E^{SC} = \{\lambda$, try, run, rest, token_in, token_out, request_in, request_out$\}$.

To describe the functioning of a system during a single execution, we postulate the existence of an omniscient observer, outside of the system under consideration. The observer is able to watch the operation of the system and compile a complete record of the events that occur, along with their time of occurrence. We refer to this record, the structure of which will be precisely defined below, as an "observation." An observation is a function that maps each instant of time t in the interval $[0, \infty)$ to the event that occurs at time t.

We assume that at most finitely many non-λ events can occur in any bounded interval. This assumption, which is used to permit inductive reasoning about observations, seems reasonable if we think of a computer as executing in discrete steps taken at a finite rate. The fact that an observation is a (single-valued) function implies that at most one event occurs at each instant of time. This is not to be interpreted as a fact about real-world systems, but rather as part of the definition of the term "event." That is, by definition no more than one event occurs at any instant. To model a situation in which a number of primitive occurrences can happen simultaneously, we must use an interface that contains one event for each possible combination of primitive occurrences.

The reason why we define observations as functions from $[0, \infty)$ to events rather than simply as sequences of events (and in Chapter 3 define computations on $[0, \infty)$ as well), is a technical one. We shall often be interested in composing a collection of observations, one for each component module in a system of modules, to obtain a single observation of the composite system. If observations are defined to be sequences of events, then composition of observations corresponds (in the special case that the component modules do not interact) to interleaving of sequences. For example, if a module M_1 can produce the sequence of events *ab* and module M_2 can produce the sequence of events *cd*, then the composite system consisting of modules M_1 and M_2 can produce the interleaved sequence of events *acbd*. The feature of interleaving that is inconvenient for our purposes is the fact that the indices of events change under interleaving. That is, the event *b* appears as the second event in the sequence *ab*, but as the third event in the sequence *acbd*. The definitions of observation and composition we use have the more convenient property that an event appearing at time *t* in an observation for module in isolation always corresponds to the event appearing at time *t* in a composite observation.

Definition - An *observation* over an interface E is a function x: $[0, \infty) \rightarrow E$, such that $x(t) \neq \lambda$ for at most finitely many t in each bounded interval. ▮

Let Λ denote the identically λ observation, and let Obs(E) denote the set of all observations over E. If $x \in$ Obs(E), and $a \in [0, \infty)$, then let $[x]$ denote the function that maps each $t \in [0, \infty)$ to the the (finite) string of non-λ events that occur during the interval $[0, t)$ in x. Let suffix$_a(x)$ be the observation $y \in$ Obs(E) such that $y(t) = x(t + a)$ for all $t \in [0, \infty)$.

By collecting the set of all observations that can be produced by a system in various environments, we obtain the "behavior" of that system.

Definition - A *behavior* of interface E is a subset of Obs(E).

Let Beh(E) be the set of all behaviors of interface E.

2.2 Abstraction, Decomposition, and Interconnection

In this section, we show how the concepts of hierarchy of abstraction and modular decomposition can be captured through the use of certain mappings between interfaces, which we call "translations," and the corresponding mappings they induce on observations.

Definition - A *translation* from an interface E to an interface F is a function h: $E \rightarrow F$ such that $h(\lambda_E) = \lambda_F$. A translation h from E to F extends in a natural way to a function h: Obs(E) $\rightarrow$ Obs(F), under the definition $h(x) = h \circ x$. ∎

The concept of an "interconnection," defined below, is the formal notion corresponding to a diagram like Figure 2. Intuitively, an interconnection consists of of an "abstraction map," which captures the relationship between a more concrete and a more abstract view of a system, and a "decomposition map," which captures the relationship between a composite system and its component modules. An abstraction map is simply a translation from the interface corresponding to the concrete view, to the interface corresponding to the abstract view. A decomposition map is a collection of translations that shows how the events for the composite system are decomposed into events for the component modules.

Definition - An *interconnection* is a pair $\mathfrak{I} = \langle \alpha^{\mathfrak{I}}, \langle \delta_i^{\mathfrak{I}} \rangle_{i \in I} \rangle$, where $\alpha^{\mathfrak{I}}$: $E^{\mathfrak{I}} \rightarrow D^{\mathfrak{I}}$ is a translation, I is a finite index set, and each $\delta_i^{\mathfrak{I}}$: $E^{\mathfrak{I}} \rightarrow F_i^{\mathfrak{I}}$ is a translation. The interfaces $F_i^{\mathfrak{I}}$ are the *component interfaces* of $\mathfrak{I}$, the interface $E^{\mathfrak{I}}$ is the *composite interface* of $\mathfrak{I}$, and the interface $D^{\mathfrak{I}}$ is the *abstract interface* of $\mathfrak{I}$. The translation $\alpha^{\mathfrak{I}}$ is the *abstraction map* of $\mathfrak{I}$, and the vector $\langle \delta_i^{\mathfrak{I}} \rangle_{i \in I}$ is the *decomposition map* of $\mathfrak{I}$. ∎

In the sequel, underlining will be used to denote a vector of objects; thus we write $\underline{\delta}^{\mathfrak{I}}$ for the vector $\langle \delta_i^{\mathfrak{I}} \rangle_{i \in I}$.

The synchronizer implementation yields an example of an interconnection. The content of Figure 2 is formalized by the interconnection $\langle \alpha^{\mathrm{SMI}}, \underline{\delta}^{\mathrm{SMI}} \rangle$, where E^{SMI}, α^{SMI}: $E^{\mathrm{SMI}} \rightarrow E^{\mathrm{SM}}$ and δ_p^{SMI}: $E^{\mathrm{SMI}} \rightarrow E^{\mathrm{SC}}$, $p \in$ Proc, are defined below.

The composite interface for the synchronizer module implementation is $E^{\mathrm{SMI}} = \{\mathrm{try}_p, \mathrm{run}_p, \mathrm{rest}_p, \mathrm{token}_p, \mathrm{request}_p : p \in \mathrm{Proc}\} \cup \{\lambda\}$.

The decomposition map $\underline{\delta}^{\mathrm{SMI}}$ projects or decomposes each event for the composite interface into corresponding events for the synchronizer component modules. The events try_p, run_p, and $rest_p$ in E^{SMI} decompose to *try*, *run*, and *rest* events of the pth synchronizer component module. The events $token_p$ and $request_p$ of E^{SMI} represent interaction between the pth synchronizer component module and its neighbors in the ring. Specifically, the event $token_p$ represents the joint occurrence of a *token_out* event for the pth synchronizer component module, and and a *token_in* event for the $p+1$st synchronizer component module. Similarly, the event $request_p$ represents the joint occurrence of a *request_out* event for the pth synchronizer component module and a *request_in* event for the $p-1$st synchronizer component module. Formally,

$$\begin{aligned}
\delta_p^{\mathrm{SMI}}(e) &= \text{try}, && \text{if } e = \text{try}_p \\
&= \text{run}, && \text{if } e = \text{run}_p \\
&= \text{rest}, && \text{if } e = \text{rest}_p \\
&= \text{token\_in}, && \text{if } e = \text{token}_{p-1} \\
&= \text{token\_out}, && \text{if } e = \text{token}_p \\
&= \text{request\_in}, && \text{if } e = \text{request}_{p+1} \\
&= \text{request\_out}, && \text{if } e = \text{request}_p \\
&= \lambda, && \text{otherwise.}
\end{aligned}$$

The abstraction map α^{SMI} preserves events in which the system of synchronizer component modules interact with the user processes, but deletes (i.e. maps to λ) events corresponding to internal interaction between synchronizer component modules. Formally,

$$\begin{aligned}
\alpha^{\mathrm{SMI}}(e) &= e, && \text{if } e \in \{\text{try}_p, \text{run}_p, \text{rest}_p : p \in \text{Proc}\}, \\
&= \lambda, && \text{if } e \in \{\text{token}_p, \text{request}_p : p \in \text{Proc}\} \cup \{\lambda\}.
\end{aligned}$$

We assign intuitive significance to some of the operators on behaviors that are naturally induced by abstraction and decomposition maps.

The direct image operator associated with an abstraction map takes a behavior of a system viewed at a more concrete level, and produces the corresponding behavior of that system viewed at a more abstract level.

Definition - The *abstraction operator* associated with a translation $\alpha: E \rightarrow D$, is the function, also denoted by α, that maps each behavior $B \in \mathrm{Beh}(E)$ to the direct image $\alpha(B) \in \mathrm{Beh}(D)$. We refer to the behavior $\alpha(B)$ as the *abstraction of B under α*. ∎

The inverse image operator induced by a decomposition map models the operation of composing a collection of component module behaviors to produce the corresponding behavior of the composite system. Intuitively, if S is a system consisting of component modules $\langle M_i \rangle_{i \in I}$, then S can produce all and only those observations x that, when decomposed, match observations that each M_i can produce.

Definition - The *composition operator* associated with the vector $\underline{\delta}$ of translations is the function, denoted by $\underline{\delta}^{-1}$, that maps a vector $\langle B_i \rangle_{i \in I}$, where $B_i \in \mathrm{Beh}(F_i)$ for each $i \in I$, to a behavior $\delta^{-1}(\underline{B}) \in \mathrm{Beh}(E)$, under the definition:

$$\delta^{-1}(\underline{B}) = \{x \in \mathrm{Obs}(E) \colon \delta_i(x) \in B_i \text{ for all } i \in I\}.$$

Thus, the set $\underline{\delta}^{-1}(\underline{B})$ contains an observation $x \in \mathrm{Obs}(E)$ iff $\delta_i(x) \in B_i$ for all $i \in I$. We call this set the *composition of* $\underline{B}$ *under* $\underline{\delta}$. ∎

2.3 Specification, Implementation, and Correctness

In practice a specification will take the form of a string of symbols in a formal specification language, since it must be possible to write down a specification. However, since this thesis is not concerned with the details of a particular formal language in which specifications are to be expressed, it is convenient to adopt a more liberal view: A specification is any mathematical object that denotes, in a well-defined way, an interface and a set of behaviors of that interface.

Definition - A *specification language* is a triple $\langle \mathrm{Specs}, \mathfrak{E}, \mathfrak{B} \rangle$, where *Specs* is a set of *specifications*, $\mathfrak{E}$ is a mapping that assigns an interface $\mathfrak{E}(S)$ to each specification $S \in \mathrm{Specs}$, and $\mathfrak{B}$ is a mapping that assigns a set $\mathfrak{B}(S) \subseteq \mathrm{Beh}(\mathfrak{E}(S))$ to each specification $S \in \mathrm{Specs}$. We say that S is a *specification of interface* $\mathfrak{E}(S)$, and that each $B \in \mathfrak{B}(S)$ *satisfies* S. ∎

An interconnection describes the pattern of interaction between modules in a system in analogy to the way a program scheme describes the flow of control between uninterpreted statements. It makes no sense to speak of an interconnection as "correct" or "incorrect," since an interconnection includes no information about the behaviors of the component or abstract modules. However, if we provide an interpretation for the modules by augmenting an interconnection with specifications of the abstract and component module interfaces, it does become meaningful to speak of correctness. We use the term "implementation" for an interconnection augmented with specifications.

Definition - An *implementation* is a tuple $\langle \mathfrak{I}, S_{\text{abs}}, \langle S_i \rangle_{i \in I} \rangle$, where $\mathfrak{I}$ is an interconnection, S_{abs} is a specification of interface $D^{\mathfrak{I}}$, and S_i is a specification of interface $F_i^{\mathfrak{I}}$, for each $i \in I$. ▮

An implementation is correct if, whenever acceptable behaviors are plugged in for the component modules, then the resulting abstract module behavior is also acceptable. The composition and abstraction operators associated with the interconnection formalize the notion of "plugging in."

Definition - An implementation $\langle \mathfrak{I}, S_{\text{abs}}, \langle S_i \rangle_{i \in I} \rangle$ is *correct* if $\alpha^{\mathfrak{I}} \circ (\underline{\delta}^{\mathfrak{I}})^{-1}(\underline{B}) \in \mathfrak{B}(S_{\text{abs}})$, whenever $B_i \in \mathfrak{B}(S_i)$ for each $i \in I$. ▮

3. State-Transition Specifications

In this chapter, we will investigate a particular approach, called "state-transition specification," to the derivation of module specifications. In this approach, we imagine that at any instant of time a module can be thought of as being in one of a number of conceptual states. Associated with each conceptual state is a collection of events that can occur in that state, and a description of the state change that results from the occurrence of each of those events. Thus, a state-transition specification describes the desired functioning of a module in terms of a kind of machine that generates an observation as it executes. It is important to note that the conceptual states in a state-transition specification are merely a tool for describing the desired functioning, and need not have anything to do with the "real" state present in any particular module instance that satisfies the specification.

The properties captured by the state-transition technique discussed here are divided into two classes: "local" properties, which concern the relationship between an event and the conceptual state of the module immediately preceding and immediately following the occurrence of that event, and "global" properties, which relate events and states perhaps distant from each other in time. Local properties are of the form: "An event *e* can occur only if the state of the module satisfies *P*, and if *e* occurs, then the old state and new state of the module are related by the binary relation *R*." Examples of global properties are "eventuality" conditions of the form: "If the module is now in a state with property *P*, then eventually event *e* will occur." Local properties are specified by a machine as mentioned above. Global properties are specified by defining a set of "validity conditions" on computations of the machine. The set of computations that satisfy the validity conditions is called the set of "valid" computations.

The reason for investigating state-transition specifications is that they appear to provide a natural, straightforward strategy for turning an intuitive understanding of the desired function of a module into a formal specification. This strategy consists of the following steps:

(1) Define an appropriate set of conceptual states. For example, in the specification of the abstract synchronizer module, a state is a vector that tells for each user process whether the synchronizer module thinks that process is trying, running, resting, or in error.

(2) Define a set of initial states, in which the module begins execution. For the

synchronizer module, there is a single initial state in which all user processes are resting.

(3) Define, for each event, the conditions required on the state for the occurrence of that event to be possible, and the state changes associated with an occurrence of that event. For example, a "run" event for process *p* can occur only if *p* is trying and no other process is currently running. Occurrence of a "run" event causes the state of *p* to change to "running" and leaves the states of all other processes unchanged.

(4) Define the desired global properties for the module. For the synchronizer module, we wish to require that every user request eventually result in a corresponding reply, if possible.

Besides serving as a natural vehicle for formalizing specifications, the state-transition approach also provides a strategy for performing correctness proofs. The Correctness Theorem (Theorem 3.9) gives sufficient conditions for correctness that exploit the machine structure of the specifications.

This chapter is organized as follows: In Section 3.1 the notion of "subset specifications," of which state-transition specifications are an example, is introduced. In Section 3.2 the machines used in state-transition specifications are defined, and in section 3.3 some tools for reasoning about their computations are developed. The notion of a state-transition specification is defined in Section 3.4. In Section 3.5 the Correctness Theorem, which is the main result of this chapter, is proved. Section 3.6 shows that the Correctness Theorem is a natural generalization of the "possibilities mapping" proof technique of Lynch [Lynch83] and Goree [Goree81]. Section 3.7 shows how the proof technique suggested by the Correctness Theorem can be further systematized in the case of state-transition specifications whose sets of valid computations have been defined by "rely-/guarantee-conditions."

3.1 Subset Specifications

As discussed in Chapter 2, a specification *S* of interface *E* defines a set $\mathfrak{B}(S)$ of behaviors of interface *E*. In general, we might look for specification techniques that are capable of expressing arbitrary properties of behaviors. However, in practice it appears that the properties of behaviors we wish to express in a specification are nearly always of a special form. That is, it is nearly always the case that we wish to express *universal* properties of the observations in a behavior, of the form: "Every observation x in B has property P," where P is a property of observations. This means that in practice it is usually not necessary to have a specification technique that is powerful enough to express arbitrary properties of behaviors. Rather, a less powerful technique, which is capable only of expressing properties of *observations*, suffices. The state-transition specification technique introduced in this chapter is of this less powerful variety.

Definition - A specification S of interface E is a *subset specification* if there exists a set $\mathcal{O}(S) \subseteq \mathrm{Obs}(E)$ such that $\mathfrak{B}(S) = \{B \in \mathrm{Beh}(E): B \subseteq \mathcal{O}(S)\}$. ▮

For the rest of this thesis we will be concerned only with subset specifications. To see what we give up by restricting our attention to subset specifications, let us consider some examples. Examples of properties of behaviors that can be expressed as subset specifications, that is, as universal statements about observations in a behavior B, are the following:

- Every observation in B contains at most finitely many occurrences of non-λ events (that is, computation always quiesces).
- In every observation in B, either each occurrence of a try event for process p is ultimately followed by a run event for process p, or else there is a point in time after which some process is in the "running" state forever.

Examples of properties of behaviors that cannot be expressed as subset specifications, and hence cannot be captured by the state-transition approach discussed here are:

- There exists an observation in B that contains at most finitely many occurrences of non-λ events (there exists a quiescing computation).
- If x is an observation in B and $t \in [0, \infty)$, such that $[x](t) = u$, then there is an observation $y \in B$ and a $t' \in [0, \infty)$ such that $[y](t') = ue$. (if the module is capable of doing u, then it is also capable of doing ue).
- If x is an observation in B, and f is an order-isomorphism from $[0, \infty)$ to $[0, \infty)$, then $x \circ f$ is also an observation in B (the module is *asynchronous*, or timing-independent).

Because the properties of behaviors defined by subset specifications are really just "lifted" properties of observations, the definition of correctness of an implementation that involves subset specifications has an equivalent statement in terms of observations.

Lemma 3.1 - Suppose that $\langle \mathcal{I}, S_{abs}, \langle S_i \rangle_{i \in I} \rangle$ is an implementation, where S_{abs} and each S_i is a subset specification. Then $\langle \mathcal{I}, S_{abs}, \underline{S} \rangle$ is correct iff $\alpha^{\mathcal{I}} \circ (\underline{\delta}^{\mathcal{I}})^{-1}(\langle O(S_i) \rangle_{i \in I}) \subseteq O(S_{abs})$.

Proof - => Suppose $\langle \mathcal{I}, S_{abs}, \underline{S} \rangle$ is correct. Suppose that $x \in \mathrm{Obs}(E^{\mathcal{I}})$ is such that $\delta_i^{\mathcal{I}}(x) \in O(S_i)$ for each $i \in I$. Then the behavior $\{x\}$ is the composition under $\underline{\delta}^{\mathcal{I}}$ of the vector of behaviors $\langle \{\delta_i^{\mathcal{I}}(x)\} \rangle_{i \in I}$, and the behavior $\{\alpha^{\mathcal{I}}(x)\}$ is the abstraction under $\alpha^{\mathcal{I}}$ of the behavior $\{x\}$. Since the behavior $\{\delta_i^{\mathcal{I}}(x)\}$ satisfies S_i for each $i \in I$, it follows by correctness that the behavior $\{\alpha^{\mathcal{I}}(x)\}$ satisfies S_{abs}. Thus $\alpha^{\mathcal{I}}(x) \in O(S_{abs})$.

<= Suppose that $\alpha^{\mathcal{I}} \circ (\underline{\delta}^{\mathcal{I}})^{-1}(\langle O(S_i) \rangle_{i \in I}) \subseteq O(S_{abs})$. For each $i \in I$, suppose that B_i is behavior that satisfies S_i. Then $B_i \subseteq O(S_i)$. Let $B_{abs} = \alpha^{\mathcal{I}} \circ (\underline{\delta}^{\mathcal{I}})^{-1}(\underline{B})$. Then B_{abs} is a subset of $O(S_{abs})$ by hypothesis, and hence Hence B_{abs} satisfies S_{abs}. ∎

3.2 Machines and Computations

In this section, we define a kind of nondeterministic machine that generates an observation in each of its computations.

Definition - A *nondeterministic event machine* (or just "machine" for short) M consists of:

- An interface E_M
- A set Q_M of *states.*
- A nonempty set $\mathrm{Init}_M \subseteq Q_M$ of *initial states.*
- A relation $\mathrm{Trans}_M \subseteq \mathrm{Steps}(E_M, Q_M) = Q_M \times E_M \times Q_M$, called the

state-transition relation, such that for all $q \in Q_M$, the *null step* $\langle q, \lambda, q \rangle \in \mathrm{Trans}_M$. ∎

If $E_M = E$, then we say that M is a machine *of interface E.*

The state-transition relation Trans_M of a machine M has a natural extension $\mathrm{Trans}_M{}^*$ that applies to strings of events, rather than just single events. Formally, define $\mathrm{Trans}_M{}^* \subseteq Q_M \times E_M{}^* \times Q_M$ to be the least relation containing Trans_M, and having the following closure property: If $\langle q, u, r \rangle \in \mathrm{Trans}_M{}^*$ and $\langle r, v, s \rangle \in \mathrm{Trans}_M{}^*$, then $\langle q, uv, s \rangle \in \mathrm{Trans}_M{}^*$. (Recall from Section 2.1 that we identify the null event λ_E with the empty string.)

Definition - A state $q \in Q_M$ is *reachable by M* if there exists a state $q_0 \in \text{Init}_M$ and a string $u \in E_M{}^*$ such that $\langle q_0, u, q\rangle \in \text{Trans}_M{}^*$.

Suppose that $R \subseteq Q_M$. Then R is *inductive* for M if

(1) $\text{Init}_M \subseteq R$.

(2) For all $\langle q, e, r\rangle \in \text{Trans}_M$, if $q \in R$ then $r \in R$.

We say that R is *invariant* for M if it contains all reachable states of M. The following extremely important *induction principle* is a standard technique (see, e.g. [Keller76]) for proving properties of reachable states.

Lemma 3.2 (Induction Principle) - Suppose M is a machine, and that $R \subseteq Q_M$. If R is inductive for M then R is invariant for M.

Proof - Straightforward. ∎

Ordinarily, a computation of a machine might be defined to be a pair consisting of a state sequence $q_0, q_1, \ldots$, and an event sequence $e_0, e_1, \ldots$, such that each step $\langle q_k, e_k, q_{k+1}\rangle$ satisfies the state-transition relation. Intuitively, q_k and q_{k+1} represent the states "just before" and "just after" the occurrence of the event e_k, respectively. To define a computation in which the notion of an event sequence has been replaced by that of an observation, we generalize the notion of a state sequence to that of a "state function," which assigns a state to each nonnegative real number, in such a way that the notion of state "just before" and "just after" each point $t \in [0, \infty)$ is meaningful.

Definition - A *state function* over a set of states Q is a function f: $[0, \infty) \rightarrow Q$ such that for all $t \in [0, \infty)$, there exists $\epsilon_t > 0$ such that f is constant on the intervals $[t-\epsilon_t, t] \cap [0, \infty)$ and $(t, t+\epsilon_t]$. ∎

We write $f(t^+)$ as an abbreviation for the constant value of f on the interval $(t, t+\epsilon_t)$, which intuitively represents the state "just after" time t. The state at and also "just before" time t is represented by the value $f(t)$.

Definition - A *history* over an interface E and state set Q is a pair $X = \langle \text{Obs}_X, \text{State}_X\rangle$, where Obs_X is an observation over E, and State_X is a state function over Q. Let $\text{Hist}(E, Q)$ denote the set of all histories over interface E and state set Q. ∎

If $X \in \text{Hist}(E, Q)$ and $t \in [0, \infty)$, then define the step occurring at time t in X by:

$\text{Step}_X(t) = \langle \text{State}_X(t), \text{Obs}_X(t), \text{State}_X(t^+) \rangle$.

The generalization of the ordinary definition of a computation is now straightforward.

Definition - A *computation* of a machine M is a history $X \in \text{Hist}(E_M, Q_M)$ such that

(1) $\text{State}_X(0) \in \text{Init}_M$.

(2) $\text{Step}_X(t) \in \text{Trans}_M$ for all $t \in [0, \infty)$.

Let Comp(M) denote the set of all computations of M. ▮

If V is a set of computations of M, then define Obs(V), the set of all observations *generated* by V, by $\text{Obs}(V) = \{\text{Obs}_X: X \in V\}$.

3.3 Properties of Histories

The purpose of this section is to develop some machinery for passing back and forth between histories and "history skeletons," which are sequences of steps plus timing information. Each history skeleton naturally defines a unique history. Conversely, given a history X we can extract (though not in a unique way) a history skeleton by restricting Step_X to a suitable subset T of $[0, \infty)$. Whereas histories have convenient behavior under projection, history skeletons are more useful for performing computational induction arguments.

Definition - A *skeletal sequence* is a monotone increasing sequence $t_0 < t_1 < \ldots$ of elements of $[0, \infty)$, such that $t_0 = 0$ and $t_k \to \infty$ as $k \to \infty$. A skeletal sequence $T = \langle t_k \rangle_{k \in \mathcal{N}}$ *spans* a history X if for each $k \in \mathcal{N}$, Obs_X is identically λ and State_X is constant on the interval (t_k, t_{k+1}). ▮

Note that by the properties of a state function, if State_X is constant on the open interval (t_k, t_{k+1}), then State_X is also constant on the right-closed interval $(t_k, t_{k+1}]$.

Lemma 3.3 - Suppose X is a history. Then there exists a skeletal sequence that spans X.

Proof - Let $T = \mathcal{N} \cup \{t \in [0, \infty): \text{Step}_X(t) \text{ is nonnull}\}$. The proof that T is a skeletal sequence that spans X uses the defining properties of observations and state functions, plus the compactness property of the closed, bounded subsets of $[0, \infty)$. The details are omitted. ∎

Corollary 3.4 - Suppose $\langle X_i \rangle_{i \in I}$ is a finite collection of histories. Then there is a skeletal sequence T that spans all the X_i.

Proof - For each $i \in I$, let T_i be a skeletal sequence that spans X_i, and define $T = \cup_{i \in I} T_i$. The finiteness of I implies that T has order type ω, and is hence a skeletal sequence. It is obvious that T spans each X_i. ∎

Definition - A *history skeleton* over an interface E and a state set Q is a function f: $T \rightarrow$ Steps(E, Q), where $T = \langle t_k \rangle_{k \in \mathcal{N}}$ is a skeletal sequence, such that if $f(t_k) = \langle q_k, e_k, r_k \rangle$ for each $k \in \mathcal{N}$, then $r_k = q_{k+1}$ for all $k \in \mathcal{N}$. The history skeleton f *spans* a history X if T spans X and f is the restriction of Step_X to T. ∎

Lemma 3.5 - Suppose that f is a history skeleton over E and Q. Then there is a unique history X over E and Q such that f spans X.

Proof - Suppose f: $T \rightarrow$ Steps(E, Q), where $T = \langle t_k \rangle_{k \in \mathcal{N}}$. Suppose $f(t_k) = \langle q_k, e_k, q_{k+1} \rangle$. The requirement that f spans X defines X uniquely:

$$\begin{aligned} \text{Obs}_X(t) &= e_k, && \text{if } t = t_k \\ &= \lambda, && \text{otherwise.} \\ \text{State}_X(t) &= q_0, && \text{if } t = 0 \\ &= q_{k+1}, && \text{if } t \in (t_k, t_{k+1}]. \end{aligned}$$

It is easy to see that X is a history. ∎

Lemma 3.6 - Suppose X is a history over E and Q. If $T = \langle t_k \rangle_{k \in \mathcal{N}}$ is a skeletal sequence that spans X, then the restriction of Step_X to T is a history skeleton that spans X.

Proof - Let f denote the restriction of Step_X to T, and suppose that $f(t_k) = \langle q_k, e_k, r_k \rangle$. If f is a history skeleton, then f spans X by definition. To see that f is a history skeleton, we must show that $r_k = q_{k+1}$ for all $k \in \mathcal{N}$. Fix $k \in \mathcal{N}$. By definition of a state function, we can select $\varepsilon > 0$ such that State_X is constant on the interval $(t_k, t_k + \varepsilon]$. Then $r_k = \text{State}_X(t_k + \varepsilon)$. Since State_X is constant on the interval $(t_k, t_{k+1}]$ by the fact that T is a skeletal sequence of X, it follows that $r_k = q_{k+1}$. ∎

The following consequence of Lemma 3.3 and Lemma 3.6 says that every state appearing in a computation is reachable.

Corollary 3.7 - Suppose X is a computation for a machine M. Then $\mathrm{State}_X(t)$ is reachable for M, for all $t \in [0, \infty)$.

Proof - Use Lemma 3.3 to obtain a skeletal sequence $T = \langle t_k \rangle_{k \in \mathcal{N}}$ that spans X. By Lemma 3.6, the restriction of Step_X to T is a history skeleton that spans X. The result follows by an inductive proof that the constant value of State_X on each set $\{t_0\}$, $(t_1, t_2]$, $(t_2, t_3]$, ... is reachable for M. The details are straightforward, and are omitted. ▮

3.4 State-Transition Specifications

Definition - A *state-transition specification S of interface E* is a pair $\langle M_S, V_S \rangle$, where M_S is a machine of interface E and V_S is a set of computations of M_S, which we call the set of *valid computations*. ▮

If S is a state-transition specification of interface E, then the set of behaviors that *satisfy* S is defined as follows:

$$\mathfrak{B}(S) = \{B \in \mathrm{Beh}(E): B \subseteq \mathrm{Obs}(V_S)\}$$

It is clear from this definition that state-transition specifications are subset specifications.

As a concrete example of a state-transition specification, consider the specification for the synchronizer module. The interface for the synchronizer module is defined by:

$$E^{SM} = \{\lambda\} \cup \{\mathrm{try}_p, \mathrm{run}_p, \mathrm{rest}_p: p \in \mathrm{Proc}\}.$$

The state set Q^{SM} for the synchronizer module specification is defined by

$$Q^{SM} = \Pi_{p \in \mathrm{Proc}} \{\mathrm{trying}, \mathrm{running}, \mathrm{resting}, \mathrm{error}\}.$$

Thus each element of the state set Q^{SM} is a vector that tells, for each process $p \in$ Proc, what the synchronizer module thinks that process is currently doing. If $q \in Q^{SM}$ and $p \in$ Proc, then let $q(p)$ denote the component of q corresponding to process p. If $v \in$ {trying, running, resting, error}, then let $q[v/p]$ denote the state $r \in Q^{SM}$ that is identical to q except that $r(p) = v$.

Next, we define the initial state set Init^{SM} and state-transition relation Trans^{SM} for the synchronizer specification. The initial state set Init^{SM} consists of the single state q that assigns the value "resting" to each $p \in \text{Proc}$. The state-transition relation Trans^{SM} contains a step $\langle q, e, r\rangle$ iff either $e = \lambda$ and $q = r$, or one of the conditions (try), (run), or (rest) below is satisfied for some $p \in \text{Proc}$:

(try) $e = \text{try}_p$, and either
$q(p) = \text{resting}$ and $r = q[\text{trying}/p]$, or
$q(p) \neq \text{resting}$ and $r = q[\text{error}/p]$.

(run) $e = \text{run}_p$, $q(p) = \text{trying}$, $q(p') \neq \text{running}$ for all $p' \in \text{Proc} - \{p\}$,
and $r = q[\text{running}/p]$.

(rest) $e = \text{rest}_p$, and either
$q(p) = \text{running}$ and $r = q[\text{resting}/p]$, or
$q(p) \neq \text{running}$ and $r = q[\text{error}/p]$.

We have defined the machine $M^{SM} = \langle E^{SM}, Q^{SM}, \text{Init}^{SM}, \text{Trans}^{SM}\rangle$ for the synchronizer module specification. To complete the state-transition specification of the synchronizer module, we must define the set V^{SM} of valid computations of M^{SM}. The intuitive property we wish to capture by this definition is that the synchronizer must eventually grant all requests, if possible. The qualification "if possible" is required since if one user process remains in the "running" state forever, then it will be impossible for the synchronizer module to grant any further requests, without violating the mutual exclusion property. We can informally state the defining property of V^{SM} as follows: "If, for all user processes p, every instant of time at which p is running is eventually followed by an instant of time at which p is not running, then, for all p, every instant of time at which p is trying is eventually followed by an instant of time at which p is running."

The validity condition for the synchronizer module is relatively simple, but already the locutions used to precisely define this condition are somewhat awkward. To deal with more complex specifications, we require a more compact notation that can be systematically applied, as opposed to the *ad hoc* approach taken above. Such a notation is developed in the next chapter, where the constructs of temporal logic are used to express properties of histories.

3.5 The Correctness Theorem

In this section we consider the problem of how to prove the correctness of an implementation with respect to state-transition specifications. The fundamental result of this section is the Correctness Theorem. This theorem shows how the correctness of an implementation follows from certain properties of a composite machine, which is a kind of a kind of product of the machines for the component module specifications and the machine for the abstract module specification. Associated with this product construction are projection maps that take each computation for the composite machine to a corresponding computation for the abstract module machine and for each component module machine.

The Correctness Theorem states that, for an implementation to be shown correct, it suffices to show that two conditions hold for the composite machine. We call these conditions the "maximality" condition and the "validity" condition. The maximality condition concerns the relationship between the state-transition relations of the component module machines and the state-transition relation of the abstract module machine. The validity condition concerns the relationship between the set of valid computations for the component modules and the set of valid computations for the abstract module.

If the inclusion of the machine from the abstract module specification as a part of the composite machine seems somewhat strange, consider the following analogy: In proofs of concurrent program correctness using Hoare-like deductive systems [Apt81, Owicki76], it is well known that it is sometimes necessary to introduce "ghost variables," which have no effect on the execution of the program, but merely serve to capture information about the state of program execution not reflected in the values of the program variables. The abstract module part of the composite machine serves the same function as ghost variables: namely to capture information about the history of system execution possibly not reflected in the states of the component module machines.

The proof technique suggested by the Correctness Theorem seems closely related to the "data refinement proofs" of [Jones81]. Jones shows how the correctness of implementations of data abstractions can be performed via "representation relations," which relate the states of abstract data objects to states of their concrete representations. Representation relations capture the same information as the "implementation invariants" defined below, and the "possibilities mappings" of Lynch [Lynch83] and Goree [Goree81] (see Section 3.6).

We now define precisely the notion of the composite machine for an implementation. Suppose $\langle \mathcal{I}, S_{\mathrm{abs}}, \langle S_i \rangle_{i \in I} \rangle$ is an implementation, where $S_{\mathrm{abs}} = \langle M_{\mathrm{abs}}, V_{\mathrm{abs}} \rangle$ and $S_i = \langle M_i, V_i \rangle$, for each $i \in I$, are state-transition specifications.

Definition - The *composite machine* M for the implementation $\langle \mathcal{I}, S_{\mathrm{abs}}$ and $\underline{S} \rangle$ is defined as follows:

$$E_M = E^{\mathcal{I}}$$
$$Q_M = Q_{M_{\mathrm{abs}}} \times \Pi_{i \in I} Q_{M_i}.$$

Let π_{abs} and π_i, be the canonical projection maps from the cartesian product Q_M onto the factors $Q_{M_{\mathrm{abs}}}$ and Q_{M_i}, for each $i \in I$.

$$\mathrm{Init}_M = \mathrm{Init}_{M_{\mathrm{abs}}} \times \Pi_{i \in I} \mathrm{Init}_{M_i}.$$
$$\mathrm{Trans}_M = \{\langle q, e, r \rangle \in \mathrm{Steps}(E_M, Q_M): \langle \pi_{\mathrm{abs}}(q), \alpha(e), \pi_{\mathrm{abs}}(r) \rangle \in \mathrm{Trans}_{M_{\mathrm{abs}}} \text{ and } \langle \pi_i(q), \delta_i(e), \pi_i(r) \rangle \in \mathrm{Trans}_{M_i} \text{ for all } i \in I\}. \blacksquare$$

Suppose that $X \in \mathrm{Hist}(E_M, Q_M)$. Then associated with X is its *canonical projection* $X^{(\mathrm{abs})}$ onto $\mathrm{Hist}(E_{M_{\mathrm{abs}}}, Q_{M_{\mathrm{abs}}})$, defined by

$$\mathrm{Obs}_{X^{(\mathrm{abs})}} = \alpha \circ \mathrm{Obs}_X$$
$$\mathrm{State}_{X^{(\mathrm{abs})}} = \pi_{\mathrm{abs}} \circ \mathrm{State}_X.$$

In a similar way, we associate with X its *canonical projection* $X^{(i)}$ onto $\mathrm{Hist}(E_{M_i}, Q_{M_i})$, defined by

$$\mathrm{Obs}_{X^{(i)}} = \delta_i \circ \mathrm{Obs}_X$$
$$\mathrm{State}_{X^{(i)}} = \pi_i \circ \mathrm{State}_X.$$

It is easily verified that the projections $X^{(\mathrm{abs})}$ and $X^{(i)}$ defined above are, in fact, histories. Also, it is easily checked that if X is a computation of M, then $X^{(\mathrm{abs})}$ is a computation of M_{abs}, and $X^{(i)}$ is a computation of M_i, for each $i \in I$.

Next, we state the conditions that are shown by the Correctness Theorem to be sufficient for $\langle \mathcal{I}, S_{\mathrm{abs}}, \underline{S} \rangle$ to be correct. Intuitively, the maximality condition states that the abstract machine can perform any event that can be performed by the system of component module machines. The validity condition states that a computation that is valid for each of the component modules is also valid for the abstract module.

Definition - The *maximality condition holds* for the implementation $\langle \mathcal{I}, S_{abs}, \underline{S} \rangle$ if for all states q reachable for the composite machine M, and all $e \in E$, if $\delta_i(e)$ is enabled for M_i in state $\pi_i(q)$ for each $i \in I$, then $\alpha(e)$ is enabled for M_{abs} in state $\pi_{abs}(q)$. ∎

Definition - An *implementation invariant* for the implementation $\langle \mathcal{I}, S_{abs}, \underline{S} \rangle$ is a set $Inv \subseteq Q_M$, such that Inv is inductive for the composite machine M. ∎

Note that an implementation invariant is indeed invariant for M by the Induction Principle (Lemma 3.2).

Since an implementation invariant contains all reachable states of the composite machine, it is sufficient to use "$q \in Inv$, where Inv is an implementation invariant," in place of "q reachable for the composite machine," in proving that the maximality condition holds.

Definition - The *validity condition holds* for the implementation $\langle \mathcal{I}, S_{abs}, \underline{S} \rangle$ if: Whenever X is a computation for the composite machine M with the property that $X^{(i)} \in V_i$ for all $i \in i$, then $X^{(abs)} \in V_{abs}$ as well. ∎

We now come to the main technical lemma (Lemma 3.8 below) used to prove the Correctness Theorem. The intuitive content of this lemma is as follows: Suppose we are given a collection $\underline{X}$ of computations for the component module machines, which are "coherent" in the sense that there is a single observation $x \in \text{Obs}(E)$ such that each Obs_{X_i} is the image of x under the mapping δ_i. The vector $\underline{X}$ of computations can be thought of as a computation of the system of machines, obtained by juxtaposing the machines for the component module specifications, and "interconnecting" their events as specified by the decomposition map $\underline{\delta}$. Lemma 3.8 asserts that, if the maximality condition holds, then it is possible to construct a computation X for the composite machine M, such that $\text{Obs}_X = x$, and furthermore, such that the projections $X^{(i)}$ of the computation X are the given original computations X_i. Since $X^{(abs)}$ must be a computation of the abstract module machine (because every computation of M projects to a computation of M_{abs}), it follows that every coherent collection $\underline{X}$ of computations for the component module machines, "simulates" some computation of the abstract module machine.

Formally, suppose that X_i is a computation of M_i, for each $i \in I$. Given an observation $x \in \mathrm{Obs}(E)$, we say that the collection $\underline{X}$ is *x-coherent* if $\mathrm{Obs}_{X_i} = \delta_i(x)$ for each $i \in I$. The point of this definition is that a vector $\underline{X}$ cannot be used to form a computation X of M unless the observations of each of the X_i are in agreement.

Lemma 3.8 - Let $\langle \mathfrak{I}, S_{\mathrm{abs}}, \langle S_i \rangle_{i \in I} \rangle$ be an implementation, where $S_{\mathrm{abs}} = \langle M_{\mathrm{abs}}, V_{\mathrm{abs}} \rangle$, and that $S_i = \langle M_i, V_i \rangle$, for each $i \in I$ are state-transition specifications. Suppose that the maximality condition holds for $\langle \mathfrak{I}, S_{\mathrm{abs}}, \underline{S} \rangle$. Suppose that $x \in \mathrm{Obs}(E^{\mathfrak{I}})$, and that X_i is a computation of M_i for each $i \in I$, such that the collection $\underline{X}$ is x-coherent. Then there exists a computation X of the composite machine M such that $\mathrm{Obs}_X = x$, and such that $X^{(i)} = X_i$ for each $i \in I$.

Proof - By Corollary 3.4, there exists a skeletal sequence $T = \langle t_k \rangle_{k \in \mathcal{N}}$ that spans each of the X_i. We assume without loss of generality that T includes all points t for which $x(t) \neq \lambda$. Let $e_k = x(t_k)$ for each k. We will use the maximality condition to construct a sequence $q_0, q_1, \ldots$ of elements of Q_M such that $\pi_i(q_k) = \mathrm{State}_{X_i}(t_k)$ for all $i \in I$ and all $k \in \mathcal{N}$, and such that $\langle q_k, e_k, q_{k+1} \rangle \subset \mathrm{Trans}_M$ for all $k \in \mathcal{N}$. Then the function $f: T \rightarrow \mathrm{Steps}(E_M, Q_M)$ that takes t_k to $\langle q_k, e_k, q_{k+1} \rangle$ is a history skeleton for M. By Lemma 3.5 there is a unique history X for M such that f spans X. It is easy to see that X is a computation of M with $\mathrm{Obs}_X = x$ and $X^{(i)} = X_i$ for each $i \in I$.

The q_k are constructed by induction on k. At the kth stage of the construction ($k \geq 0$), we assume that q_k has been constructed so that q_k is reachable and $\pi_i(q_k) = \mathrm{State}_{X_i}(t_k)$ for all $i \in I$. We construct q_{k+1} so that $\pi_i(q_{k+1}) = \mathrm{State}_{X_i}(t_{k+1})$ for all $i \in I$ and so that $\langle q_k, e_k, q_{k+1} \rangle \in \mathrm{Trans}_M$. It follows by definition of reachability that q_{k+1} is reachable.

Basis: Let q_0 be an arbitrary element of $\{q \in \mathrm{Init}_M: \pi_i(q) = \mathrm{State}_{X_i}(0) \text{ for all } i \in I\}$. Note that this set is nonempty since it is a cartesian product of nonempty sets. Clearly $q_0 \in$ *Inv* and $\pi_i(q_0) = \mathrm{State}_{X_i}(0)$ for all $i \in I$.

Induction: Suppose, for some $k \in \mathcal{N}$, that q_k has been defined so that $q_k \in$ *Inv* and $\pi_i(q_k) = \mathrm{State}_{X_i}(t_k)$ for all $i \in I$. Since X_i is a computation for M_i, for each $i \in I$, we know that $\delta_i(e_k)$ is enabled for M_i in state $\pi_i(q_k)$, for each $i \in I$. Since q_k is reachable for M, the maximality condition implies that $\alpha(e_k)$ is enabled for M_{abs} in state $\pi_{\mathrm{abs}}(q_k)$. Hence

$\{q \in Q_M: \langle q_k, e_k, q\rangle \in \mathrm{Trans}_M$ and $\pi_i(q) = \mathrm{State}_{X_i}(t_{k+1})$ for all $i \in I\}$ is nonempty. Let q_{k+1} be an arbitrary element of this set. ∎

The Correctness Theorem is an easy consequence of the preceding lemma.

Theorem 3.9 (Correctness Theorem) - If the maximality and validity conditions hold for an implementation, then the implementation is correct.

Proof - Suppose $\langle \mathcal{I}, S_{\mathrm{abs}}, \langle S_i\rangle_{i\in I}\rangle$ is an implementation, where $S_{\mathrm{abs}} = \langle M_{\mathrm{abs}}, V_{\mathrm{abs}}\rangle$ and that $S_i = \langle M_i, V_i\rangle$ for each $i \in I$ are state-transition specifications. Suppose that the maximality and validity conditions hold. Let M be the composite machine. Suppose that $x \in \mathrm{Obs}(E)$ is such that $\delta_i(x) \in \mathcal{O}(S_i)$ for all $i \in I$. By Lemma 3.1, it suffices to show that $\alpha(x) \in \mathcal{O}(S_{\mathrm{abs}})$. Since $\delta_i(x) \in \mathcal{O}(S_i)$ for each $i \in I$, we know that for each $i \in I$ there is a computation $X_i \in V_i$, such that $\mathrm{Obs}_{X_i} = \delta_i(x)$. Since the collection $\underline{X}$ is x-coherent, by Lemma 3.8 there exists a computation X for the composite machine M, such that $\mathrm{Obs}_X = x$ and such that $X^{(i)} = X_i$ for all $i \in I$. Using the validity condition, we then conclude that $X^{(\mathrm{abs})} \in V_{\mathrm{abs}}$. It follows that $\alpha(x) = \alpha(\mathrm{Obs}_X) \in \mathcal{O}(S_{\mathrm{abs}})$, as required. ∎

3.6 Possibilities Mappings

In this section we show that the Correctness Theorem is a natural generalization of the "possibilities mapping" proof technique proposed by Lynch [Lynch83] and Goree [Goree81].

Lynch and Goree define a possibilities mapping to be a function that assigns a set of abstract module machine states to each vector of states for the component module machines, in such a way that the initial state set and state-transition relation are preserved. The fact that each vector of component module states is mapped to a *set* of abstract states, rather than to a *single* abstract state, means that possibilities mappings are a generalization of the usual notions of simulation or machine homomorphism. Intuitively, the value of the simulation mapping on a vector of component states is the set of "possible" abstract states that correspond to the given component states -- hence the name "possibilities mapping."

Lynch and Goree's proof technique can be stated as follows: "If there exists a possibilities mapping for an implementation, then the implementation is correct." Interpreted in the framework of this thesis, Lynch and Goree's technique applies only to implementations that involve state-transition specifications $\langle M, V\rangle$ for which V = Comp(M). For such implementations, the validity condition required by the Correctness Theorem is vacuous. Theorem 3.10 below shows that the existence of a possibilities mapping is equivalent to the maximality condition required by the Correctness Theorem, and thus the Correctness Theorem includes Lynch and Goree's proof technique as a special case.

To define the notion of a possibilities mapping, suppose $\langle \mathcal{I}, S_{abs}, \langle S_i\rangle_{i\in I}\rangle$ is an implementation. Suppose $S_{abs} = \langle M_{abs}, V_{abs}\rangle$ and $S_i = \langle M_i, V_i\rangle$, for each $i \in I$. Let M be the composite machine.

Definition - A *possibilities mapping* for the implementation $\langle \mathcal{I}, S_{abs}, \langle S_i\rangle_{i\in I}\rangle$ is a function f: $\Pi_{i\in I} Q_{M_i} \rightarrow \mathcal{P}(Q_{M_{abs}})$, with the following properties:

(1) $\mathrm{Init}_{M_{abs}} \subseteq f(\langle q_i\rangle_{i\in I})$ whenever $q_i \in \mathrm{Init}_{M_i}$ for all $i \in I$.

(2) For all $q \in Q_M$, if $\pi_{abs}(q) \in f(\langle \pi_i(q)\rangle_{i\in I})$, then:

(a) Whenever $r \in Q_M$ and $e \in E_M$ are such that $\langle q, e, r\rangle \in \mathrm{Trans}_M$, then $\pi_{abs}(r) \in f(\langle \pi_i(r)\rangle_{i\in I})$.

(b) For all $e \in E_M$, if $\delta_i(e)$ is enabled in state $\pi_i(q)$ for each $i \in I$, then $\alpha(e)$ is enabled for M_{abs} in state $\pi_{abs}(q)$. ∎

Theorem 3.10 - Suppose that $\langle \mathcal{I}, S_{abs}, \langle S_i\rangle_{i\in I}\rangle$ is an implementation, where S_{abs} and S_i for each $i \in I$ are state-transition specifications. Then the following are equivalent:

(1) There exists a possibilities mapping for $\langle \mathcal{I}, S_{abs}, \underline{S}\rangle$.

(2) The maximality condition holds for $\langle \mathcal{I}, S_{abs}, \underline{S}\rangle$.

Proof - Suppose that $S_{abs} = \langle M_{abs}, V_{abs}\rangle$, and that $S_i = \langle M_i, V_i\rangle$, for each $i \in I$. Let M be the composite machine for the implementation $\langle \mathcal{I}, S_{abs}, S\rangle$.

(1) => (2): Suppose that f is a possibilities mapping for $\mathcal{I}$, S_{abs} and $\underline{S}$. Define

$$Inv = \{q \in Q_M: \pi_{abs}(q) \in f(\langle \pi_i(q)\rangle_{i\in I})\}.$$

Condition (1) in the definition of a possibilities mapping implies that $\mathrm{Init}_M \subseteq Inv$. Condition (2)(a) in the definition of a possibilities mapping implies that Inv is inductive, and hence by Lemma 3.2 contains all states reachable for M. The maximality condition now follows from condition (2)(b) in the definition of a possibilities mapping.

(2) => (1): Conversely, suppose that the maximality condition holds. Define f: $\Pi_{i\in I} Q_i \rightarrow$

Q_{abs} as follows: $f(\langle q_i \rangle_{i \in I})$ is the set of all $q_{\text{abs}} \in Q_{\text{abs}}$ such that there exists a reachable state q for M with $\pi_{\text{abs}}(q) = q_{\text{abs}}$ and $\pi_i(q) = q_i$ for all $i \in I$. We claim that f is a possibilities mapping.

Condition (1) in the definition of a possibilities mapping holds, since given $\langle q_i \rangle_{i \in I} \in \Pi_{i \in I} \text{Init}_{M_i}$, then every $q_{\text{abs}} \in \text{Init}_{M_{\text{abs}}}$ yields a state $\langle q_{\text{abs}}, \langle q_i \rangle_{i \in I} \rangle$ that is in Init_M and hence is reachable for M.

To show that condition (2) holds, suppose that q is a state of M such that $\pi_{\text{abs}}(q) \in f(\langle \pi_i(q) \rangle_{i \in I})$. Then q is reachable for M by definition of f. To see that (2)(a) holds, note that if $\langle q, e, r \rangle \in \text{Trans}_M$, then r is reachable by definition of reachability, and hence $\pi_{\text{abs}}(r) \in f(\langle \pi_i(r) \rangle_{i \in I})$. The maximality condition implies that condition (2)(b) holds. ∎

3.7 Rely-/Guarantee-Conditions

In this section we will see how state-transition specifications whose sets of valid computations are defined by *rely-/guarantee-conditions* can be used to perform the validity part of a proof of correctness. The principle of rely-/guarantee-conditions states that the set of valid computations V in a state-transition specification $S = \langle M, V \rangle$ should be defined in the form: "*Rely* implies *Guar*," where *Rely* expresses the properties that the module being specified relies on its environment to provide, and *Guar* expresses the properties that the module guarantees to provide in return.

For the synchronizer module, we wish the validity conditions to capture the idea that every user's request should eventually result in a response, if possible. The tricky part is the precise formulation of the "if possible" condition. Clearly if some user goes into the running state and remains in that state forever, then it will never be possible to allow any other user in the trying state to go to the running state, without violating the mutual exclusion property. This condition can be stated in rely-/guarantee-condition form as follows: "If every user process obeys the requirement that, once in the running state, it will eventually leave the running state, then the synchronizer module guarantees that every user in the trying state will eventually leave the trying state (and hence advance to the running state.)"

We have two results, Lemma 3.11 and Lemma 3.12 below, that describe techniques for using rely-/guarantee-condition specifications in proofs of correctness. In both of these techniques, we are required to prove:

(*) Each component module's rely-condition is implied by the conjunction of the abstract module's rely-condition and the guarantee-conditions for some subset of the component modules.

Although the exact form taken by condition (*) is different for the two techniques, a proof by either of the techniques is simplest when the rely- and guarantee-conditions for the component modules are chosen so that the truth of condition (*) is obvious. Thus, rely- and guarantee-conditions serve to "cut" the interdependence of modules on each other, analogously to the way in which a loop invariant cuts the dependence of one iteration of a loop on the previous iteration. This observation is strong motivation for the suggestion that module specifications ought not to be derived in isolation, but rather with a proof of correctness in mind in which those specifications are used.

A correctness proof that makes use of Lemma 3.11 or Lemma 3.12 is rather different from one in which eventuality conditions (such as termination) are verified by the well-founded set techniques of [Floyd67, Keller76] and others. Proofs by the latter techniques tend to take the form of reasoning about the structure of a computation, whereas proofs by Lemma 3.11 and Lemma 3.12 tend to be arguments based on the communication structure of the modules in the system. Experience gained from the examples presented in this thesis suggests that arguments based on communication structure are simpler and more natural.

The use of rely- and guarantee-conditions has been proposed for safety specifications in [Jones83]. Independently of this thesis, Barringer and Kuiper [Barringer83] have proposed the use of liveness specifications that are partitioned into an "environment part," which captures assumptions made about the environment, and a "component part," which captures commitments made by the module being specified. Jones, as well as Barringer and Kuiper, exploit the rely-/guarantee-condition structure of specifications by defining inference rules for process composition that seem closely related to Lemma 3.11. Barringer and Kuiper's environment/component division seems essentially the same as the rely/guarantee division used in this thesis, except that Barringer and Kuiper apply the environment/component division to state-transition properties, as well as liveness properties.

Misra and Chandy [Misra81] have also used a kind of rely/guarantee distinction to develop proof techniques for safety properties. In that paper, a process *h* is specified by an assertion of the form $r|h|s$, where r and s are predicates on finite sequences (called *traces*) of communication events. Such an assertion is interpreted as: "The predicate s holds of the empty trace, and for all traces t that can be produced by process h, if r holds for all *proper* prefixes of t, then s holds for all prefixes (both proper and improper) of t. The predicates r and s can be thought of as roughly analogous to rely- and guarantee-conditions, respectively, although the former are properties of finite prefixes of traces rather than properties of infinite computations. Misra and Chandy's proof technique is a "Theorem of Hierarchy," which gives conditions under which specifications of a collection of components can be used to infer a specification of the network formed by interconnecting the components. Their proof technique can be stated as follows: To show that the specification $R0|H|S0$ for the network H is a consequence of the specifications $r_i|h_i|s_i$ $(i \in I)$ for the components, it suffices to show that:

(1) S implies $S0$,

(2) $R0$ and S implies R,

where R and S denote the conjunction of the r_i and s_i, respectively. These conditions are syntactically similar to the conditions (1) and (2) of Lemma 3.11, although their meaning is quite different. The proof of Misra and Chandy's Theorem of Hierarchy is by induction on computation prefixes, whereas the proof of Lemma 3.11 is by structural induction using a well-founded dependency relation.

In [Misra82], the techniques of [Misra81] are extended to encompass a weak form of liveness specification in which an additional predicate q is used to state conditions under which a process trace is guaranteed to be extended. The Theorem of Hierarchy is augmented with additional conditions to permit its application to these more general specifications. The additional conditions do not appear to relate in a simple way to any conditions used in this thesis.

To state Lemma 3.11 and Lemma 3.12, the following notation is convenient: If R and G are subsets of a universe U, then define $R \rightarrow_U G$ (read R *implies* G in U) to be the subset $(U - R) \cup G$ of U. In applications of Lemma 3.11 and Lemma 3.12, the set U will be the set Comp(M) of computations of a machine M, and R and G will be the sets of computations of M that satisfy rely-conditions and guarantee-conditions, respectively.

Lemma 3.11 below says that to prove that the validity condition holds, it suffices to prove:

(1) The abstract module's guarantee condition is implied by the conjunction of

the guarantee conditions for the component modules.

(2) There exists a well-founded partial ordering (a "depends on" relation) of the component modules in the system, such that each component module's rely-condition is implied by the conjunction of the abstract module's rely-condition and the guarantee-conditions for the modules on which the component depends.

Lemma 3.11 - (Rely/Guarantee Technique I) - Suppose U is a set and that R_{abs}, G_{abs}, and R_i, G_i for each $i \in I$ are subsets of U. Suppose $V_{\mathrm{abs}} = R_{\mathrm{abs}} \rightarrow_U G_{\mathrm{abs}}$ and $V_i = R_i \rightarrow_U G_i$, for each $i \in I$. Suppose

(1) $\bigcap_{i \in I} G_i \subseteq G_{\mathrm{abs}}$,

(2) There exists a well-founded partial order $<$ on I such that for all $i \in I$,

$$R_{\mathrm{abs}} \cap (\bigcap_{j<i} G_j) \subseteq R_i.$$

Then $\bigcap_{i \in I} V_i \subseteq V_{\mathrm{abs}}$.

Proof - Suppose $X \in R_{\mathrm{abs}} \cap (\bigcap_{i \in I} V_i)$. Suppose further, to obtain a contradiction, that $X \notin G_{\mathrm{abs}}$. Then by hypothesis (1) we know that $X \notin G_{i_0}$ for some $i_0 \in I$.

Since $X \notin G_{i_0}$, and since $X \in V_{i_0}$ by assumption, it must be the case that $X \notin R_{i_0}$. By hypothesis (2) and the assumption that $X \in R_{\mathrm{abs}}$, there exists $i_1 < i_0$ such that $X \notin G_{i_1}$. Repeating this argument yields an infinite descending sequence $i_0 > i_1 > \dots$, in contradiction with the well-foundedness of $<$. ∎

An example of the use of Lemma 3.11 can be found in the proof of correctness of the transmission module implementation in Appendix II.

The existence of the "depends on" relation required to satisfy hypothesis (2) of Lemma 3.11 is a rather stringent condition. In some cases, for example the synchronizer implementation, all of the component modules in the system are symmetric in their relationship to each other, and it is hard to see how a suitable dependency relation might be found. Lemma 3.12 below shows that an alternative "acyclicity" condition can be used, in case the component module rely- and guarantee-conditions can be factored in a certain way. Specifically, Lemma 3.12 assumes that the rely-condition for module i can be expressed as the conjunction of what module i relies on the external environment and on each component module j to provide, and that the guarantee-condition for module i can be expressed as the conjunction of what module i guarantees to the external environment and to each component module j.

In Lemma 3.12 below, one should think of $R_{\mathbf{abs}}$, $G_{\mathbf{abs}}$ as the rely- and guarantee-conditions for the abstract module, and of R_i, G_i as the rely- and guarantee-conditions for component module i. The hypotheses of Lemma 3.12 require us to find $\{RG_{i,j}: i, j \in I + \{\text{abs}\}\}$. (RG stands for "rely/guarantee.") Intuitively, if $i, j \in I$, then $RG_{i,j}$ expresses what module i guarantees to module j, and also what module j relies on module i to provide. $RG_{\mathbf{abs},j}$ expresses what the external environment of the entire system guarantees to component module j, and also what module j relies on the external environment to provide. $RG_{i,\mathbf{abs}}$ expresses what component module i guarantees to the external environment, and also what the external environment relies on module i to provide.

Condition (1)(a) and (1)(b) in Lemma 3.12 state, intuitively, that the abstract module's rely-condition implies what each of the component modules rely on the external environment to provide, and that the abstract module's guarantee condition is implied by the conjunction of what each of the component modules guarantees to provide to the external environment. Condition (2)(a) states that each component module's rely-condition is implied by the conjunction of what that component relies on the external environment to provide and on what that component relies on the component modules in the system to provide. Condition (2)(b) states that the guarantee-condition for component module i implies what module i guarantees to the external environment and what module i guarantees to each of the component modules in the system. Condition (3) in Lemma 3.12 is an acyclicity condition, which states that there can be no unbroken cyclic dependency between component modules.

If I is a set, then define a *cycle* of I to be a nonempty subset of $I \times I$ of the form: $\{\langle i_0, i_1\rangle, \langle i_1, i_2\rangle, \dots, \langle i_{n-1}, i_n\rangle\}$, such that $i_n = i_0$.

Lemma 3.12 (Rely/Guarantee Technique II) - Let I be a finite index set. Suppose that U is a set and that $R_{\mathbf{abs}}$, $G_{\mathbf{abs}}$, and R_i, G_i for each $i \in I$ are subsets of U. Suppose $V_{\mathbf{abs}} = R_{\mathbf{abs}} \rightarrow_U G_{\mathbf{abs}}$ and $V_i = R_i \rightarrow_U G_i$, for each $i \in I$. If there exists, for each $i, j \in I + \{\text{abs}\}$, a set $RG_{i,j} \subseteq U$ such that (1)-(3) below hold, then $\bigcap_{i \in I} V_i \subseteq V_{\mathbf{abs}}$.

(1)(a) $R_{\mathbf{abs}} \subseteq \bigcap_{j \in I} RG_{\mathbf{abs},j}$.

(b) $\bigcap_{i \in I} RG_{i,\mathbf{abs}} \subseteq G_{\mathbf{abs}}$.

(2)(a) $\bigcap_{i \in I + \{\text{abs}\}} RG_{i,j} \subseteq R_j$, for all $j \in I$.

(b) $G_i \subseteq \bigcap_{j \in I + \{\text{abs}\}} RG_{i,j}$, for all $i \in I$.

(3) Whenever $\{\langle i_0, i_1\rangle, \langle i_1, i_2\rangle, \ldots, \langle i_{n-1}, i_n\rangle\}$ is a cycle of I, then

$$U = \bigcup_{k=0}^{n-1} RG_{i_k, i_{k+1}}.$$

Proof - Suppose (1)-(3). Suppose further, to obtain a contradiction, that there exists $X \in U \cap R_{\text{abs}} \cap (\bigcap_{i \in I} V_i)$ such that $X \notin G_{\text{abs}}$. We perform an inductive construction to obtain a cycle $\{\langle i_m, i_{m+1}\rangle, \ldots, \langle i_n, i_{n+1}\rangle\}$ of I such that $X \notin \bigcup_{k=m}^{n-1} RG_{i_k, i_{k+1}}$. This contradicts hypothesis (3).

As the induction hypothesis at stage k of the construction, we assume that $i_1, i_2, \ldots, i_k$ have been constructed so that $X \notin R_{i_k}$, and that $X \notin \bigcup_{j=1}^{k-1} RG_{i_j, i_{j+1}}$.

Basis: From (1)(a) and the assumption that $X \in R_{\text{abs}}$, we know that $X \in RG_{\text{abs}, j}$ for all $j \in I$. Since $X \notin G_{\text{abs}}$, by (1)(b) we know that $X \notin RG_{i_1, \text{abs}}$ for some $i_1 \in I$. By (2)(b) we know that $X \notin G_{i_1}$, and from the assumption that $X \in V_{i_1}$, we conclude $X \notin R_{i_1}$.

Induction: Assume the induction hypothesis holds for some $k \geq 1$. By (2)(a) we know that $X \notin RG_{i_k, i_{k+1}}$ for some $i_{k+1} \in I$. If $i_{k+1} = i_m$ for some m with $1 \leq m \leq k$, then we have obtained the desired cycle and the construction terminates. Otherwise, by (2)(b) we know that $X \notin G_{i_{k+1}}$, and from the assumption that $X \in V_{i_{k+1}}$, we conclude that $X \notin R_{i_{k+1}}$. This establishes the induction hypothesis for $k+1$.

Since the set I is finite by hypothesis, we cannot extend the sequence $i_1, i_2, \ldots, i_k$ indefinitely without creating a cycle. ∎

Examples of the use of Lemma 3.12 can be found in the proof of correctness of the synchronizer implementation in Chapter 4, and in the proof of correctness of the resource manager implementation in Appendix II.

4. The Synchronizer Implementation

In this chapter, the theory developed in Chapter 3 is applied to obtain complete specifications and a proof of correctness for the synchronizer example. In Section 4.1 we review the synchronizer module specification which has already been developed. It is shown how the set of valid computations for this specification can be given a concise definition using the language of temporal logic. In Section 4.2, the synchronizer component module specification is presented. In Section 4.3, the definition of the synchronizer module implementation is reviewed. In Section 4.4, the Correctness Theorem is used to prove the correctness of the synchronizer implementation.

4.1 Notation

This section introduces the notation we will use to express state-transition specifications, and in particular, the temporal logic notation we use to define the sets of valid computations. We use this notation in this chapter in a highly informal fashion, and do not concern ourselves with precise syntax and semantics. The reader who is interested in a careful treatment of the notation we use is referred to Appendix I.

To define a state-transition specification S, we first define the interface E_S and state set Q_S of the machine M_S. As discussed in detail in Appendix I, we regard these two sets as two distinguished sorts *Events* and *States* in a many-sorted algebra A_S. We associate a first-order language $\mathcal{L}(S)$ with the algebra A_S in the usual way. The language $\mathcal{L}(S)$ is used to define the initial state set $Init_S$ and the state-transition relation $Trans_S$ of the machine M_S. In this chapter, we often use constructions that are not part of a first-order language. Appendix I shows how the use of these constructions can be justified.

From the first-order language $\mathcal{L}(S)$, we obtain a temporal language $\mathfrak{T}(S)$ by augmenting $\mathcal{L}(S)$ with the temporal operators $\Box$ (read "henceforth") and $\Diamond$ (read "eventually"), which are applied to formulas to obtain new formulas. In addition, three new atomic terms are added to the language: **Now** and **After**, which behave syntactically like constant symbols of sort *States*, and **Occurs**, which behaves like a constant symbol of sort *Events*. The meanings of the symbols **Now**, **Occurs**, and **After** depend upon the particular instant of time under consideration, and thus are altered by the action of temporal operators $\Box$ and $\Diamond$ in a way that is detailed below. Intuitively, if the particular instant of time under consideration is t, then **Occurs** denotes the event that occurs at time t, **Now** denotes the state at time t, and **After** denotes the state "just after" time t.

The semantics of the temporal language associated with a specification $S = \langle M, V\rangle$ are captured by the binary relation $\models$ (read "satisfies"), which tells when a formula of the temporal language is satisfied by a particular history over E_M and Q_M. To assert that the history X satisfies a particular temporal formula φ, we write $X \models \varphi$. Satisfaction is defined informally as follows: If φ is a formula that contains no occurrences of temporal operators, then $X \models \varphi$ iff φ holds in the usual sense of first-order logic, with the symbols **Now**, **Occurs**, and **After** interpreted as $\mathrm{State}_X(0)$, $\mathrm{Obs}_X(0)$, and $\mathrm{State}_X(0^+)$, respectively. If φ is a formula of the form $\Box\psi$, then $X \models \varphi$ iff $\mathrm{suffix}_t(X) \models \psi$ for *all* $t \in [0, \infty)$. If φ is of the form $\Diamond\psi$, then $X \models \varphi$ iff $\mathrm{suffix}_t(X) \models \psi$ for *some* $t \in [0, \infty)$. Note that the semantics we use are essentially the "linear time" semantics of [Lamport80], and hence the $\Diamond$ operator is equivalent to the compound operator $\neg\Box\neg$.

We say that a formula φ is a *consequence* of a set of formulas Ψ, written $\Psi \models \varphi$, if $X \models \varphi$ whenever $X \models \psi$ for all $\psi \in \Psi$. A formula φ is *valid*, written $\models \varphi$, if it is a consequence of the null set of formulas.

The temporal language $\mathcal{T}(S)$ of a specification $S = \langle M_S, V_S\rangle$ contains an important sentence to which we shall refer extensively. This is the sentence

$$\mathrm{Comp}_S \equiv \mathrm{Init}_S(\mathbf{Now}) \wedge \Box\mathrm{Trans}_S(\mathbf{Now}, \mathbf{Occurs}, \mathbf{After}).$$

Intuitively, $X \models \mathrm{Comp}_S$ iff X is a computation for the machine M_S.

4.2 Specification of the Synchronizer Module

In this section, we review the state-transition specification $S^{SM} = \langle M^{SM}, V^{SM}\rangle$ of the synchronizer module, which has already been developed in Chapter 3.

Let Proc be a finite set of user processes.

Interface:

$$E^{SM} = \{\lambda\} \cup \{\mathrm{try}_p, \mathrm{run}_p, \mathrm{rest}_p : p \in \mathrm{Proc}\}.$$

In anticipation of Chapter 5, we classify each event in the synchronizer module interface as either an input event, an output event, or both (the null event λ is the only event that is both an input and an output event).

$\mathrm{In}^{SM} = \{\lambda\} \cup \{\mathrm{try}_p, \mathrm{rest}_p: p \in \mathrm{Proc}\}$
$\mathrm{Out}^{SM} = \{\lambda\} \cup \{\mathrm{run}_p: p \in \mathrm{Proc}\}$.

Although our theoretical framework so far draws no formal distinction between input and output, in Chapter 5 such a distinction is introduced to obtain a useful test for consistency of liveness specifications. Input events should be thought of intuitively as stimuli that are applied to a module by its environment, and output events as responses applied by a module to its environment. A module does not have the capability of regulating the application of input stimuli to it.

Machine:

The state set for the synchronizer module machine is defined by

$Q^{SM} = \Pi_{p \in \mathrm{Proc}} \{\mathrm{trying, running, resting, error}\}$.

To ease later discussion, let us say that process p is *resting* (resp. *trying*, *running*, *in error*) in state q if $q(p)$ = resting (resp. trying, running, error).

The set of initial states for the synchronizer module machine is defined by

$\mathrm{Init}^{SM} = \{q \in Q^{SM}: q(p) = \text{resting for all } p \in \mathrm{Proc}\}$.

A step $\langle q, e, r\rangle$ is in the state-transition relation Trans^{SM} for the synchronizer module machine iff either $e = \lambda$ and $q = r$, or one of the conditions (try), (run), or (rest) below is satisfied for some $p \in \mathrm{Proc}$.

A *try* event for process p can occur at any time. If process p was previously resting then it advances to the trying state, otherwise to the error state. The states of all other processes are unaffected.

(try) $e = \mathrm{try}_p$, and either
$q(p) = \text{resting and } r = q[\text{trying}/p]$, or
$q(p) \neq \text{resting and } r = q[\text{error}/p]$.

A *run* event for process p can occur only if process p is trying, and no other processes are currently running. Process p advances to the running state, and the states of all other processes are unaffected.

(run) $e = \mathrm{run}_p$, $q(p) = \text{trying}$, $q(p') \neq \text{running for all } p' \in \mathrm{Proc} - \{p\}$,
and $r = q[\text{running}/p]$.

A *rest* event for process p can occur at any time. If process p was previously running, then it advances to the resting state, otherwise to the error state. The states of all other processes are unaffected.

(rest) $e = \text{rest}_p$, and either
$q(p) = \text{running}$ and $r = q[\text{resting}/p]$, or
$q(p) \neq \text{running}$ and $r = q[\text{error}/p]$.

Validity Conditions:

We wish the validity condition for the synchronizer module to capture the idea that every user's request should eventually result in a response, if possible. This condition can be stated in the rely-/guarantee-condition form as follows: "If every user process obeys the requirement that, once in the running state, it will eventually leave the running state, then the synchronizer module guarantees that every user in the trying state will eventually leave the trying state (and hence advance to the running state)." We can express this condition concisely as a temporal sentence.

$$\text{Valid}^{SM} \equiv \text{Rely}^{SM} \rightarrow \text{Guar}^{SM}$$

where

$$\text{Rely}^{SM} \equiv \Box(\forall p \in \text{Proc})(\mathbf{Now}(p) = \text{running} \rightarrow \Diamond(\mathbf{Now}(p) \neq \text{running}))$$
$$\text{Guar}^{SM} \equiv \Box(\forall p \in \text{Proc})(\mathbf{Now}(p) = \text{trying} \rightarrow \Diamond(\mathbf{Now}(p) \neq \text{trying})).$$

4.3 Specification of the Synchronizer Component Module

A synchronizer component module communicates with an associated user process via the *try*, *run*, and *rest* events, with its neighboring synchronizer component module in the clockwise direction via *token_out* and *request_in* events, and with its neighboring synchronizer component module in the counterclockwise direction via *token_in* and *request_out* events. The conceptual state of the module contains a count of the number of tokens the module possesses, plus information concerning the state of the associated user process. The synchronizer component module can allow the user process to enter the *running* state only if it possesses a token, and must retain a token throughout the entire period during which the user is in the running state. We would like the synchronizer component module to be "fair" in the sense it eventually grants each user request, if possible, and eventually responds to each request for the token by its clockwise neighbor in the ring, if possible.

The specification of the synchronizer component module is parameterized by the number of tokens it possesses in the initial state. Thus the specification presented below actually is a specification schema that represents a family $\{SC_k\colon k \in \mathcal{N}\}$ of related specifications, where SC_k is the specification for the synchronizer component module with k tokens in the initial state. The only place the initial number of tokens appears in the specification is in the definition of the initial state set.

Interface:

The first task in the construction of the synchronizer component module specification is the description of its interface.

E^{SC} = {λ, try, run, rest, token_in, token_out, request_in, request_out}.

The sets of input and output events are defined by:

In^{SC} = {λ, try, rest, token_in, request_in}

Out^{SC} = {λ, run, token_out, request_out}.

Machine:

A state for the synchronizer component module contains a "token" component, whose value represents the number of tokens the module possesses, and a "ustate" component, which tells what state the synchronizer component module thinks the user process is in.

Q^{SC} = token: $\mathcal{N} \times$ ustate: {trying, running, resting, error}.

The "tags" *token* and *ustate* are used as selectors; if $q \in Q^{SC}$, then q(token) denotes the *token* component of q and q(ustate) denotes the *ustate* component.

In an initial state the synchronizer component module SC_k has k tokens and the user process is resting.

Init^{SC_k} = {$q \in Q^{SC}$: q(token) = $k \wedge q$(ustate) = resting}.

A step $\langle q, e, r\rangle$ is in the state-transition relation Trans^{SM} for the synchronizer component module machine iff either $e = \lambda$ and $q = r$, or one of the conditions (try), (run), or (rest), (token_in), (token_out), (request_in), (request_out) below is satisfied:

A *try* event can occur at any time. If the user process was previously resting, then it advances to the trying state, otherwise to the error state.

(try) e = try and either

q(ustate) = resting and r = q[trying/ustate], or
q(ustate) ≠ resting and r = q[error/ustate].

A *run* event can occur only if the user process is trying and the synchronizer component module currently possesses a token. The user process advances to the running state.

(run) e = run, q(ustate) = trying, q(token) ≠ 0, and r = q[running/ustate].

A *rest* event can occur at any time. If the user process was previously running, then it advances to the trying state, otherwise to the error state.

(rest) e = rest and either
q(ustate) = running and r = q[resting/ustate], or
q(ustate) ≠ running and r = q[error/ustate].

A *token_in* event can occur at any time, and causes the number of tokens possessed by the synchronizer component module to be increased by one.

(token_in) e = token_in and r = q[q(token) + 1/token]

A *token_out* event can occur only if the user process is currently not running, and the synchronizer component module possesses at least one token. The number of tokens possessed is decremented.

(token_out) e = token_out, q(ustate) ≠ running, q(token) ≠ 0, and
r = q[q(token)–1/token]

A *request_in* event can occur at any time, and has no direct effect on the state. The way in which a request_in event induces the synchronizer component module to eventually respond with a token_out event is captured by the validity conditions.

(request_in) e = request_in and r = q

A *request_out* event can occur only if the synchronizer component module currently does not possess a token. Occurrence of such an event has no effect on the state.

(request_out) e = request_out, q(token) = 0, and r = q

Validity Conditions:

We would like the synchronizer component module validity conditions to capture the following two ideas:

(1) A synchronizer component module always eventually satisfies a user's request, if possible.

(2) A synchronizer component module always responds to requests for the token issued by its clockwise neighbor, if possible.

We can state this in rely-/guarantee-condition form as follows: If all requests issued by the synchronizer component module to its counterclockwise neighbor are eventually granted, and the user process never remains forever in the running state, then all user requests and all requests for the token from the clockwise neighbor, will eventually be granted. Formally,

$$\mathrm{Valid}^{SC} \equiv \mathrm{Rely}^{SC} \rightarrow \mathrm{Guar}^{SC},$$

where

$$\begin{aligned}\mathrm{Rely}^{SC} \equiv\ & \Box(\mathbf{Now}(\mathrm{ustate}) = \mathrm{running} \rightarrow \Diamond(\mathbf{Now}(\mathrm{ustate}) \neq \mathrm{running})) \wedge \\ & \Box(\mathbf{Occurs} = \mathrm{request\_out} \rightarrow \Diamond(\mathbf{Now}(\mathrm{token}) \neq 0)) \\ \mathrm{Guar}^{SC} \equiv\ & \Box(\mathbf{Now}(\mathrm{ustate}) = \mathrm{trying} \rightarrow \Diamond(\mathbf{Now}(\mathrm{ustate}) \neq \mathrm{trying})) \wedge \\ & \Box(\mathbf{Occurs} = \mathrm{request\_in} \rightarrow \Diamond(\mathbf{Occurs} = \mathrm{token\_out}))\end{aligned}$$

4.3.1 The Synchronizer Implementation

To be able to describe and reason about the synchronizer implementation we must formalize the idea that the set *Proc* is a "ring-structured set of processes with a distinguished process." We assume that the set *Proc* is the set of integers modulo *N* for some *N*, and that *zero* is a distinguished process, which will be the process that initially possesses the token.

We first define the synchronizer interconnection

$$\mathfrak{I}^{SMI} = \langle E^{SMI}, \alpha^{SMI}, \langle \delta_p^{SMI} \rangle_{p \in Proc} \rangle.$$

The abstract interface D^{SMI} is the synchronizer module interface E^{SM}, and the pth component interface F_p^{SMI} is the synchronizer component module interface E^{SC}.

The composite interface for the synchronizer module implementation is defined by:

$$\begin{aligned}E^{SMI} &= \{\lambda\} + \{\mathrm{try}_p, \mathrm{run}_p, \mathrm{rest}_p, \mathrm{token}_p, \mathrm{request}_p : p \in \mathrm{Proc}\}. \\ \mathrm{In}^{SMI} &= \{\lambda\} + \{\mathrm{try}_p, \mathrm{rest}_p : p \in \mathrm{Proc}\} \\ \mathrm{Out}^{SMI} &= \{\lambda\} + \{\mathrm{run}_p, \mathrm{token}_p, \mathrm{request}_p : p \in \mathrm{Proc}\}.\end{aligned}$$

The *try*$_p$, *run*$_p$, and *rest*$_p$ events in the composite interface correspond under the decomposition map to *try*, *run*, and *rest* events for synchronizer component module p, and under the abstraction map to *try*$_p$, *run*$_p$, and *rest*$_p$ events for the synchronizer module. A *token*$_p$ event represents the transmission of a token from synchronizer component module p to synchronizer component module $p+1$ (i.e. in the clockwise direction around the ring), and a "request" event represents the transmission of a request from synchronizer component module p to synchronizer component module $p-1$ (i.e. in the counterclockwise direction). We capture this information formally by defining the abstraction map α^{SMI} and decomposition map $\underline{\delta}^{SMI}$.

$$\begin{aligned} \alpha^{SMI}(e) &= e, && \text{if } e \in \{\text{try}_p, \text{run}_p, \text{rest}_p : p \in \text{Proc}\}, \\ &= \lambda, && \text{if } e \in \{\text{token}_p, \text{request}_p : p \in \text{Proc}\} \cup \{\lambda\}. \end{aligned}$$

$$\begin{aligned} \delta_p^{SMI}(e) &= \text{try}, && \text{if } e = \text{try}_p \\ &= \text{run}, && \text{if } e = \text{run}_p \\ &= \text{rest}, && \text{if } e = \text{rest}_p \\ &= \text{token\_in}, && \text{if } e = \text{token}_{p-1} \\ &= \text{token\_out}, && \text{if } e = \text{token}_p \\ &= \text{request\_in}, && \text{if } e = \text{request}_{p+1} \\ &= \text{request\_out}, && \text{if } e = \text{request}_p \\ &= \lambda, && \text{otherwise.} \end{aligned}$$

To complete the description of the synchronizer implementation $\langle \mathcal{I}^{SMI}, S_{abs}^{SMI}, \underline{S}^{SMI} \rangle$, we must define the specifications S_{abs}^{SMI} and S_p^{SMI} for each $p \in$ Proc. The specification S_{abs}^{SMI} is the synchronizer module specification S^{SM}. The specification S_{zero}^{SMI} is the specification S^{SC_1} of the synchronizer component module with one initial token, and for all $p \in$ Proc $-$ {zero}, S_p^{SMI} is the specification S^{SC_0} of the synchronizer component module with no initial tokens.

4.4 Correctness of the Synchronizer Implementation

In this section, we use the techniques of Chapter 3 to show the correctness of the synchronizer implementation. Most of the proof consists of straightforward case analyses. The interesting content of the proof is contained in the use of Lemma 3.12 to prove that the validity condition holds.

4.4.1 Implementation Invariant

To prove the correctness of the synchronizer module implementation, we first need to find an implementation invariant that provides enough information about the reachable states of the composite machine so that we can prove the maximality condition. The implementation invariant will also be useful in the proof that the validity condition holds, and so in this section we define an implementation invariant that is strong enough for both the maximality and validity proofs.

For a set *Inv* to be an implementation invariant for an implementation means that it is inductive for the composite machine for the implementation. Formally, if *M* is the composite machine and *E* the composite interface, we must show:

(Basis) $(\forall q \in Q_M)(q \in \mathrm{Init}_M \rightarrow q \in \mathrm{Inv}))$

(Induction) $(\forall q,r \in Q_M, e \in E)(\langle q, e, r\rangle \in \mathrm{Trans}_M \rightarrow (q \in \mathrm{Inv} \rightarrow r \in \mathrm{Inv}))$.

It is generally convenient to define an implementation invariant *Inv* by a predicate

$\mathrm{Inv}(q) \equiv \mathrm{Rep}(q) \wedge \mathrm{Abs}(q)$,

where *Rep* is called the *representation invariant* and *Abs* is called the *abstraction relation*. A representation invariant describes a relationship that must hold at all times between the states of component modules in an implementation. Representation invariants serve roughly the same purpose here as what is called the "data type invariant" in the literature on abstract data types [e.g. Jones81, Jones83]. An abstraction relation describes the correspondence between the states of the component modules and the state of the abstract module. The abstraction relation plays the same role here as the "retrieve functions" of [Jones81], and the "representation functions" of [Hoare72].

The implementation invariant Inv^{SMI} for the synchronizer implementation is defined as follows:

$\mathrm{Inv}^{SMI}(q) \equiv \mathrm{Rep}^{SMI}(q) \wedge \mathrm{Abs}^{SMI}(q)$,

The abstraction relation Abs^{SMI} holds of state *q* iff in state *q*, the abstract synchronizer module's view of the state of the *p*th user process is identical to the *p*th synchronizer component module's view, for each *p* in Proc. Stated another way, the abstract synchronizer module state corresponding to a given collection of synchronizer component module states is obtained by throwing away all information, except for the *ustate* component, in the states of the component modules. Formally,

$$\mathrm{Abs}^{\mathrm{SMI}}(q) \equiv \wedge_{p \in \mathrm{Proc}}(q_{\mathrm{abs}}(p) = q_p(\mathrm{ustate})),$$

where we have used the notations q_{abs}, q_p as abbreviations for $\pi_{\mathrm{abs}}(q)$, $\pi_p(q)$, respectively.

The representation invariant $\mathrm{Rep}^{\mathrm{SMI}}$ is defined by:

$$\mathrm{Rep}^{\mathrm{SMI}}(q) \equiv \mathrm{Mutex}(q) \wedge \mathrm{Token}(q).$$

Mutex(q) states that if a user process is in the running state, then the corresponding synchronizer component module must possess a token. Formally,

$$\mathrm{Mutex}(q) \equiv \wedge_{p \in \mathrm{Proc}}(q_p(\mathrm{ustate}) = \mathrm{running} \rightarrow q_p(\mathrm{token}) \neq 0).$$

Token(q) asserts that the total number of tokens in the system at any time is precisely one.

$$\mathrm{Token}(q) \equiv \Sigma_{p \in \mathrm{Proc}}\, q_p(\mathrm{token}) = 1.$$

The proof that $\mathrm{Inv}^{\mathrm{SMI}}(q)$ is in fact an implementation invariant for the synchronizer implementation is a straightforward induction.

Basis: It follows directly from the initial state sets that if $\mathrm{Init}^{\mathrm{SMI}}(q)$ holds, then

$$\begin{array}{lll} q_{\mathrm{abs}}(p) & = \mathrm{resting} & \text{for all } p \in \mathrm{Proc} \\ q_{\mathrm{zero}} & = \langle \mathrm{token{:}}\ 1, \mathrm{ustate{:}}\ \mathrm{resting}\rangle & \\ q_p & = \langle \mathrm{token{:}}\ 0, \mathrm{ustate{:}}\ \mathrm{resting}\rangle & \text{for all } p \in \mathrm{Proc}-\{\mathrm{zero}\}. \end{array}$$

It is easily checked that these three conditions imply that $\mathrm{Abs}^{\mathrm{SMI}}(q)$, Mutex($q$), and Token($q$) all hold. We conclude that $\mathrm{Inv}^{\mathrm{SMI}}(q)$ holds for all $q \in \mathrm{Init}^{\mathrm{SMI}}$, as required.

Induction: We must show that for all $\langle q, e, r\rangle \in \mathrm{Trans}^{\mathrm{SMI}}$, if $\mathrm{Inv}^{\mathrm{SMI}}(q)$ holds then $\mathrm{Inv}^{\mathrm{SMI}}(r)$ does, too. Suppose that $\langle q, e, r\rangle \in \mathrm{Trans}^{\mathrm{SMI}}$ and $\mathrm{Inv}^{\mathrm{SMI}}(q)$ holds.

First of all, note that if $e = \lambda$, then $q = r$ and hence $\mathrm{Inv}^{\mathrm{SMI}}(r)$ follows trivially from $\mathrm{Inv}^{\mathrm{SMI}}(q)$. We therefore assume in what follows that $e \neq \lambda$. We consider separately the proofs of $\mathrm{Abs}^{\mathrm{SMI}}(r)$, Mutex($r$), and Token($r$).

To prove that $\mathrm{Abs}^{\mathrm{SMI}}(r)$ holds, there are two cases: (1) $e \in \{\mathrm{token}_p, \mathrm{request}_p{:}\ p \in \mathrm{Proc}\}$; and (2) $e \in \{\mathrm{try}_p, \mathrm{run}_p, \mathrm{rest}_p{:}\ p \in \mathrm{Proc}\}$. Case (1) is disposed of quickly by noting that if $e = \mathrm{token}_p$, or $e = \mathrm{request}_p$ for some $p \in \mathrm{Proc}$, then $r_{\mathrm{abs}}(p') = q_{\mathrm{abs}}(p')$ and $r_{p'}(\mathrm{ustate}) = q_{p'}(\mathrm{ustate})$ for all $p' \in \mathrm{Proc}$. Thus in this case $\mathrm{Abs}^{\mathrm{SMI}}(r)$ follows directly from $\mathrm{Abs}^{\mathrm{SMI}}(q)$. Case (2) is handled by a straightforward enumeration of the cases: $e = \mathrm{try}_p$, $e = \mathrm{run}_p$, $e = \mathrm{rest}_p$, and verifying that in each case, the occurrence of e results in identical values for $r_{\mathrm{abs}}(p')$ and $r_{p'}(\mathrm{ustate})$, for each $p' \in \mathrm{Proc}$.

We now consider the proof that Mutex(r) holds. Suppose not, then it must be the case that r_p(ustate) = running and r_p(token) = 0 for some $p \in$ Proc. By a case analysis on e it is straightforward to check that the only way this can happen is if either q_p(ustate) = running and e = token$_p$, or q_p(token) = 0 and e = run$_p$. Examination of the specification of the synchronizer component module shows that it is impossible for a *token*$_p$ event to occur if q_p(ustate) = running, and also for a *run*$_p$ event to occur if q_p(token) = 0.

Finally, we wish to show that Token(r) holds. A case analysis on e shows that the only events that affect the number of tokens in the system are those of the form *token*$_p$ for some $p \in$ Proc. Examination of the specifications shows that, when such an event occurs, r_p(token) = q_p(token) − 1, r_{p+1}(token) = q_{p+1}(token) + 1, and $r_{p'}$(token) = $q_{p'}$(token) for all $p' \in$ Proc − $\{p, p+1\}$. Thus $\Sigma_{p' \in \text{Proc}}\, r_{p'}$(token) = $\Sigma_{p' \in \text{Proc}}\, q_{p'}$(token), and hence Token(r) holds.

4.4.2 Proof of Maximality

We must show that for all $q \in Q^{\text{SMI}}$ and $e \in E^{\text{SMI}}$, if $Inv^{\text{SMI}}(q)$ holds and $\delta_p^{\text{SMI}}(e)$ is enabled in state q_p for all $p \in$ Proc, then $\alpha^{\text{SMI}}(e)$ is enabled in state q_{abs}.

Suppose $Inv^{\text{SMI}}(q)$ holds and that $\delta_p^{\text{SMI}}(e)$ is enabled in state q_p for all $p \in$ Proc. There are two cases: (1) e = *run*$_p$ for some $p \in$ Proc; and (2) e is not of this form. Examination of the synchronizer module specifications shows that case (2) is trivial, since $\alpha(e)$ is enabled in any state unless e = *run*$_p$ for some $p \in$ Proc.

Now consider case (1). Since $\delta_p^{\text{SMI}}(e)$ is enabled in state q_p, from the synchronizer component module specification we know that

(A) q_p(ustate) = trying and q_p(token) $\neq$ 0.

The assumption that Inv$^{\text{SMI}}$(q) holds implies that Token(q), Mutex(q), and Abs$^{\text{SMI}}$(q) all hold. From (A) and Abs$^{\text{SMI}}$(q) we infer that $q_{\text{abs}}(p)$ = trying. From (A) and Token(q) we know that $q_{p'}$(token) = 0 for all $p' \in$ Proc−$\{p\}$. From this and Mutex(q) we infer that $q_{p'}$(ustate) $\neq$ running for all $p' \in$ Proc−$\{p\}$. From this and Abs$^{\text{SMI}}$(q), we conclude that $q_{\text{abs}}(p') \neq$ running for all $p' \in$ Proc−$\{p\}$. We have shown that

(B) $q_{\text{abs}}(p)$ = trying $\wedge$ ($\bigwedge_{p' \in \text{Proc}-\{p\}} q_{\text{abs}}(p') \neq$ running).

holds. Examination of the synchronizer module specifications shows that (B) implies that $\alpha^{\text{SMI}}(e)$ is enabled in state q_{abs}, as desired.

4.4.3 Proof of Validity

To express the proof that the validity condition holds for the synchronizer implementation, we associate a temporal language $\mathcal{T}(S^{SMI})$ with the composite specification $S^{SMI} = \langle M^{SMI}, V^{SMI} \rangle$ in the same way as temporal languages were associated with the synchronizer module and synchronizer component module specifications. In addition, we must have some way of taking the temporal sentences, each expressed in its own temporal language, that define the sets of valid computations for the synchronizer module and synchronizer component module specifications, and "lifting" them to the common language $\mathcal{T}(S^{SMI})$. This can be accomplished by a simple syntactic translation, which we now define.

To each formula φ of $\mathcal{T}(S^{SM})$ we associate a corresponding "lifted" version $[\![\varphi]\!]_{abs}$ of $\mathcal{T}(S^{SMI})$, by replacing each occurrence of the symbol **Now** by the term $\mathbf{Now}_{abs}$, each occurrence of **After** by the term $\mathbf{After}_{abs}$, and each occurrence of **Occurs** by the term $\alpha^{SMI}(\mathbf{Occurs})$. Similarly, to each formula φ of $\mathcal{T}(S^{SC})$ and each $p \in$ Proc, we associate a corresponding formula $[\![\varphi]\!]_p \in \mathcal{T}(S^{SMI})$, by replacing each occurrence of **Now** by $\mathbf{Now}_i$, each occurrence of **After** by $\mathbf{After}_i$, and each occurrence of **Occurs** by $\delta_i^{SMI}(\mathbf{Occurs})$.

The precise relationship between a formula and its lifted version is captured by Lemma I.2 in Appendix I. Informally, if $\varphi \in \mathcal{T}(S^{SM})$, then a history X for the composite machine M^{SMI} satisfies the formula $[\![\varphi]\!]_{abs} \in \mathcal{T}(A^{SMI})$ iff the canonical projection $X^{(abs)}$ of X satisfies the formula φ. Similarly, if $\varphi \in \mathcal{T}(S^{SC})$, then a history X for M^{SMI} satisfies $[\![\varphi]\!]_p$ iff $X^{(i)}$ satisfies φ. An analogous result is stated in [Wolper82], where the process of "lifting" specifications of individual processes to obtain specifications of a system of processes is called "relativization."

In the proof that the validity condition holds for the synchronizer implementation, we must have some way of making use of the information contained in the state-transition relation of the composite machine. We do this by using the implementation invariant according to the following rule of inference: If *Inv* has been shown to be invariant, then

$$\text{Comp} \models \Box \text{Inv}(\mathbf{Now})$$

holds. This rule, whose validity follows from Corollary 3.7, will be used extensively in our correctness proofs.

To show that the validity condition holds for the synchronizer implementation, we must show that:

$$\mathrm{Comp}^{\mathrm{SMI}} \models \bigwedge_{p\in\mathrm{Proc}} [\![\mathrm{Valid}^{\mathrm{SC}}]\!]_p \rightarrow [\![\mathrm{Valid}^{\mathrm{SM}}]\!]_{\mathrm{abs}}.$$

In light of Lemma 3.12 it suffices to find, for each $i, j \in \mathrm{Proc} + \{\mathrm{abs}\}$, a temporal sentence $\mathrm{RG}_{i,j}$ such that conditions (SMI1)-(SMI3) below hold.

(SMI1)(a) $\quad \mathrm{Comp}^{\mathrm{SMI}} \models [\![\mathrm{Rely}^{\mathrm{SM}}]\!]_{\mathrm{abs}} \rightarrow \bigwedge_{j\in\mathrm{Proc}} \mathrm{RG}_{\mathrm{abs},j}$

(SMI1)(b) $\quad \mathrm{Comp}^{\mathrm{SMI}} \models \bigwedge_{i\in\mathrm{Proc}} \mathrm{RG}_{i,\mathrm{abs}} \rightarrow [\![\mathrm{Guar}^{\mathrm{SM}}]\!]_{\mathrm{abs}}$

(SMI2)(a) $\quad \mathrm{Comp}^{\mathrm{SMI}} \models \bigwedge_{j\in\mathrm{Proc}} (\bigwedge_{i\in\mathrm{Proc}+\{\mathrm{abs}\}} \mathrm{RG}_{i,j} \rightarrow [\![\mathrm{Rely}^{\mathrm{SC}}]\!]_j)$

(SMI2)(b) $\quad \mathrm{Comp}^{\mathrm{SMI}} \models \bigwedge_{i\in\mathrm{Proc}} ([\![\mathrm{Guar}^{\mathrm{SC}}]\!]_i \rightarrow \bigwedge_{j\in\mathrm{Proc}+\{\mathrm{abs}\}} \mathrm{RG}_{i,j})$

(SMI3) Whenever $\{\langle i_0, i_1\rangle, \langle i_1, i_2\rangle, \ldots, \langle i_{n-1}, i_n\rangle\}$ is a cycle of Proc, then

$$\mathrm{Comp}^{\mathrm{SMI}} \models \bigvee_{k=0}^{n-1} \mathrm{RG}_{i_k,i_{k+1}}.$$

The sentences $\mathrm{RG}_{i,j}$ express what is relied/guaranteed between each pair of synchronizer component modules or between a synchronizer component module and the external environment of the entire system. The synchronizer component module specifications have been chosen in such a way that the sentences $\mathrm{RG}_{i,j}$ can be obtained simply by "lifting" the synchronizer component module rely-/guarantee-conditions to the temporal language of the composite machine. The formal definitions are as follows:

For all $i, j \in \mathrm{Proc}$,

$$\mathrm{RG}_{i,\mathrm{abs}} \equiv \Box(\mathrm{Now}_i(\mathrm{ustate}) = \mathrm{trying} \rightarrow \Diamond(\mathrm{Now}_i(\mathrm{ustate}) \neq \mathrm{trying}))$$

$$\mathrm{RG}_{\mathrm{abs},j} \equiv \Box(\mathrm{Now}_j(\mathrm{ustate}) = \mathrm{running} \rightarrow \Diamond(\mathrm{Now}_j(\mathrm{ustate}) \neq \mathrm{running}))$$

For all $i \in \mathrm{Proc}$,

$$\mathrm{RG}_{i-1,i} \equiv \Box(\delta_i^{\mathrm{SMI}}(\mathbf{Occurs}) = \mathrm{request\_out} \rightarrow \Diamond(\mathrm{Now}_i(\mathrm{token}) \neq 0))$$

For all $i, j \in \mathrm{Proc}$ such that $i + 1 \neq j$,

$$\mathrm{RG}_{i,j} \equiv \mathbf{true}$$

Next, we verify (SMI1)-(SMI3). Assume $\mathrm{Comp}^{\mathrm{SMI}}$ throughout the remainder of the proof. The interesting intuitive content of the validity proof is contained in the proof that (SMI3) holds. The remaining cases are practically automatic.

Intuitively, hypothesis (SMI1)(a) says that the abstract module rely condition implies what each component module relies on the external environment to provide. Hypothesis (SMI1)(b) says that the conjunction of what each component module guarantees to the external environment implies the abstract module guarantee-condition. Formally, we must show:

(SMI1)(a) $[\![\mathrm{Rely}^{SM}]\!]_{abs} \rightarrow \bigwedge_{p \in Proc} \mathrm{RG}_{abs,p}$

(SMI1)(b) $\bigwedge_{p \in Proc} \mathrm{RG}_{p,abs} \rightarrow [\![\mathrm{Guar}^{SM}]\!]_{abs}$

We show (SMI1)(a), condition (SMI2)(b) is equally straightforward. From the synchronizer module specifications, we know that

$$[\![\mathrm{Rely}^{SM}]\!]_{abs} \equiv \Box \bigwedge_{p \in Proc}(\mathbf{Now}_{abs}(p) = \text{running} \rightarrow \Diamond(\mathbf{Now}_{abs}(p) \neq \text{running}))$$

Suppose that $[\![\mathrm{Rely}^{SM}]\!]_{abs}$ holds. By the invariance of the abstraction relation Abs^{SMI}, we infer

$$\Box \bigwedge_{p \in Proc}(\mathbf{Now}_p(\text{ustate}) = \text{running} \rightarrow \Diamond(\mathbf{Now}_p(\text{ustate}) \neq \text{running})).$$

Interchanging the $\Box$ and conjunction yields

$$\bigwedge_{p \in Proc} \mathrm{RG}_{abs,p},$$

as desired.

Intuitively, hypothesis (SMI2)(a) says that each component module's rely-condition is implied by the conjunction of what is guaranteed to it by the external environment and by each other component module. Hypothesis (SMI2)(b) says that each component module's guarantee-condition implies what the external environment and each other component module rely on it to provide. Formally, we must show:

(SMI2)(a) $\bigwedge_{i \in Proc+\{abs\}} \mathrm{RG}_{i,p} \rightarrow [\![\mathrm{Rely}^{SC}]\!]_p$, for all $p \in$ Proc

(SMI2)(b) $[\![\mathrm{Guar}^{SC}]\!]_p \rightarrow \bigwedge_{j \in Proc+\{abs\}} \mathrm{RG}_{p,j}$, for all $p \in$ Proc.

To show condition (SMI2)(a) is completely straightforward. Let $p \in$ Proc be fixed. It suffices to show that $\mathrm{RG}_{abs,p} \wedge \mathrm{RG}_{p-1,p} \rightarrow [\![\mathrm{Rely}^{SC}]\!]_p$. By definition

$$\mathrm{RG}_{abs,p} \equiv \Box(\mathbf{Now}_p(\text{ustate}) = \text{running} \rightarrow \Diamond(\mathbf{Now}_p(\text{ustate}) \neq \text{running}))$$

$$\mathrm{RG}_{p-1,p} \equiv \Box(\delta_p^{SMI}(\mathbf{Occurs}) = \text{request\_out} \rightarrow \Diamond(\mathbf{Now}_p(\text{token}) \neq 0)).$$

The conjunction of these two sentences is easily seen to be equivalent to $[\![\mathrm{Rely}^{SC}]\!]_p$ by inspection of the synchronizer component module specifications.

To show (SMI2)(b) is not completely trivial because of the fact that what component module p guarantees to module $p+1$ is not exactly what module $p+1$ relies on module p to provide. Specifically, module p guarantees always to eventually send the token in response to a request from module $p+1$. However, module $p+1$ relies not on the eventual occurrence of a *token_in* event, but rather on the eventual setting of the *token* component of its state to a nonzero value. The nontrivial portion of the proof is to use the state-transition relation for module $p+1$ to show that occurrence of a *token_in* event for that module implies the eventual setting of the *token* component of its state to a nonzero value.

Formally, to prove (SMI2)(b) it suffices to show that $[\![\mathrm{Guar}^{SC}]\!]_p \rightarrow \mathrm{RG}_{p,\mathrm{abs}} \wedge \mathrm{RG}_{p,p+1}$, since $\mathrm{RG}_{p,j} \equiv$ **true** by definition unless j = abs or $j = p+1$. By definition

$$\mathrm{RG}_{p,\mathrm{abs}} \equiv \Box(\mathrm{Now}_p(\mathrm{ustate}) = \mathrm{trying} \rightarrow \Diamond(\mathrm{Now}_p(\mathrm{ustate}) \neq \mathrm{trying}))$$

$$\mathrm{RG}_{p,p+1} \equiv \Box(\delta^{SMI}_{p+1}(\mathbf{Occurs}) = \mathrm{request\_out} \rightarrow \Diamond(\mathrm{Now}_{p+1}(\mathrm{token}) \neq 0)).$$

By inspection of the synchronizer component module specifications, we have

$$[\![\mathrm{Guar}^{SC}]\!]_p \equiv \Box(\mathrm{Now}_p(\mathrm{ustate}) = \mathrm{trying} \rightarrow \Diamond(\mathrm{Now}_p(\mathrm{ustate}) \neq \mathrm{trying})) \wedge$$
$$\Box(\delta^{SMI}_p(\mathbf{Occurs}) = \mathrm{request\_in} \rightarrow \Diamond(\delta^{SMI}_p(\mathbf{Occurs}) = \mathrm{token\_out})).$$

Assume $[\![\mathrm{Guar}^{SC}]\!]_p$. Then $\mathrm{RG}_{p,\mathrm{abs}}$ follows immediately from the first conjunct of $[\![\mathrm{Guar}^{SC}]\!]_p$. To show that the second conjunct of $[\![\mathrm{Guar}^{SC}]\!]_p$ implies $\mathrm{RG}_{p,p+1}$, note the definition of the state-transition relation for synchronizer component module p implies that

$$\Box(\delta^{SMI}_{p+1}(\mathbf{Occurs}) = \mathrm{token\_in} \rightarrow \Diamond(\mathrm{Now}_{p+1} \neq 0)).$$

From the second conjunct of $[\![\mathrm{Guar}^{SC}]\!]_p$, using the definition of the decomposition map $\underline{\delta}^{SMI}$, we obtain

$$\Box(\delta^{SMI}_{p+1}(\mathbf{Occurs}) = \mathrm{request\_out} \rightarrow \Diamond(\delta^{SMI}_{p+1}(\mathbf{Occurs}) = \mathrm{token\_in})).$$

Combining the preceding two sentences and applying temporal reasoning shows $\mathrm{RG}_{p,p+1}$, as desired.

The most interesting part of the proof is the proof that (SMI3) holds. To show (SMI3), we must show that

$$\mathrm{Comp}^{SMI} \models \vee_{k=1}^{n-1} \mathrm{RG}_{i_k,i_{k+1}}$$

holds for every cycle $\{\langle i_0, i_1\rangle, \ldots, \langle i_{n-1}, i_n\rangle\}$ from Proc. The only nontrivial case is the cycle $\{\langle \mathrm{zero}, \mathrm{zero}+1\rangle, \langle \mathrm{zero}+1, \mathrm{zero}+2\rangle, \ldots, \langle \mathrm{zero}+N-1, \mathrm{zero}\rangle\}$ that traverses the entire ring in the clockwise direction, since every other cycle from Proc contains a link $\langle i, j\rangle$ for which $\mathrm{RG}_{i,j} \equiv$ **true** by definition. Suppose, to obtain a contradiction, that (SMI3) fails for this cycle. Then for all $p \in$ Proc, the sentence $\mathrm{RG}_{p-1,p}$ does not hold. This means that

$$\wedge_{p \in \mathrm{Proc}} \Diamond(\delta^{SMI}_p(\mathbf{Occurs}) = \mathrm{request\_out} \wedge \Box(\mathrm{Now}_p(\mathrm{token}) = 0)).$$

That is, for each $p \in$ Proc, eventually a point is reached at which synchronizer component module p issues a request_out event, but never has the token after that point. This implies that

$$\wedge_{p \in \mathrm{Proc}} \Diamond\Box(\mathrm{Now}_p(\mathrm{token}) = 0).$$

Since Proc is a finite set, it is valid to interchange the conjunction and $\Diamond$ operator in the preceding formula, concluding that

$$\Diamond \wedge_{p \in \mathrm{Proc}} \Box(\mathrm{Now}_p(\mathrm{token}) = 0).$$

This asserts that there is some point after which no synchronizer component module ever possesses a token. This is a contradiction with the invariance of Token, which states that the total number of tokens in the system is always precisely one. ▮

5. Consistency of Specifications

In Chapter 3 it is suggested that module specifications ought to be expressed in rely-/guarantee-condition form, and that the rely- and guarantee-conditions for the component modules in a system ought to be selected so that each component module guarantees precisely the conditions relied upon by its neighbors in the system. In Chapter 4 the synchronizer example illustrates how adherence to this principle can result in a simple proof of the validity condition required by the Correctness Theorem. In practice, there seems to be considerable flexibility in the choice of rely- and guarantee-conditions. Often significant simplifications in a correctness proof can be effected simply by adjusting the component module specifications.

The apparent flexibility in the choice of rely- and guarantee-conditions in specifications raises the following somewhat disturbing question: What is to prevent us from writing component module specifications with extremely weak rely-conditions (e.g. true), and ridiculously strong guarantee-conditions (e.g. false), in order to simplify the proof of correctness? An implementation whose component module validity conditions are all of the form "true $\rightarrow$ false" makes the validity part of a correctness proof extremely simple, but also vacuous. We can also consider more subtle, but still problematic specifications in which a module "guarantees" the application of some input to it -- something that seems to contradict our intuitive notion of what it means to be an input.

Since a specification of the form "true $\rightarrow$ false," or a specification that guarantees the application of input ought to be regarded as meaningless, we should have some way of distinguishing these specifications from others that are meaningful. The theory we have set up so far provides no formal criteria for making such a distinction. What we require is a suitable notion of consistency of specifications, with respect to which obviously unrealizable specifications such as "true $\rightarrow$ false" are inconsistent, and apparently reasonable specifications, such as the synchronizer component module specification, are consistent.

In mathematical logic, a theory is consistent iff it has a model. Since the "models" of specifications are behaviors, it seems reasonable to define a specification to be consistent iff there is a behavior that satisfies it. If we take the term "behavior" in this definition to mean "arbitrary behavior," though, we do not obtain a stringent enough notion of consistency. For example, every subset specification is consistent in this sense, since the empty behavior $\emptyset$ satisfies every subset specification. To obtain more stringent notions of consistency, we must restrict our interpretation of the term "behavior" to mean "realizable" or "computable" behavior.

In this chapter, we examine a notion of consistency based on a model of concurrent computation called "I/O-systems." An I/O-system models a collection of concurrent processes that interact through coupled events. By viewing I/O-systems at various levels of abstraction we obtain the "I/O-behaviors," which we take as our class of computable behaviors. A specification is defined to be "I/O-consistent" iff there exists an I/O-behavior that satisfies it. The notion of I/O-consistency seems to be quite useful for distinguishing between meaningful and meaningless eventuality specifications. We develop a technique for proving state-transition specifications to be I/O-consistent and apply this technique to show the I/O-consistency of the synchronizer component module specification.

5.1 I/O-Systems

This section defines a model of asynchronous concurrent computation called "I/O-systems." An I/O-system is a system of nondeterministic processes that interact through coupled events. The nonnull events in which each process can participate are partitioned into "input events" and "output events." An input event for a process represents the stimulation of the process by its environment, and an output event for a process corresponds to the process responding to its environment. A process can choose whether or not it will produce output, but does not have the ability to control the application of input to itself. If a process wishes to produce output, then it cannot be prevented from doing so, although a process has no control over precisely when the output will be produced.

The coupling of the processes in an I/O-system is described by a "system interface," the elements of which are "system events." Each system event is a vector with one component for each process in the system, and represents a possible simultaneous occurrence in the computation of the system. No system event contains more than one component output event, modeling the idea that at most one process can produce an output at any instant of time.

To describe the execution of an I/O-system, it is helpful to imagine the existence of a "scheduler," who controls the path of execution of the system. For each step of the system, the scheduler chooses a system event from the system interface. All processes then simultaneously take steps corresponding to the chosen system event. By the constraint that there is at most one output component of each system event, at most one process produces an output event in each step, and the other processes perform input steps or null steps. We are only interested in computations of an I/O-system that are "fair" in the sense that the scheduler selects each process to perform output steps often enough.

We now give a formal definition of I/O-systems. We first define the notion of an I/O-interface, which is an interface whose non-λ events are partitioned into input events and output events.

Definition - An *I/O-interface* is an interface $\langle E, \lambda_E, \mathrm{In}_E, \mathrm{Out}_E \rangle$, where $\mathrm{In}_E \subseteq E$ is a set of *input events* and $\mathrm{Out}_E \subseteq E$ is a set of *output events*, such that the sets In_E, Out_E, $\{\lambda_E\}$ partition E. ▮

We next define the "asynchronous product" of a collection of I/O-interfaces. Intuitively, the asynchronous product $\otimes_{i \in I} F_i$ of the collection $\langle F_i \rangle_{i \in I}$ of I/O-interfaces represents the set of all possible simultaneous occurrences in a system of processes where process i has interface F_i. Each element of the asynchronous product interface is a vector of events from the component interfaces, such that at most one of the events in the vector is an output event. The fact that at most one event in each vector is an output event means that at most one process produces an output event at a time. This restriction is typical of asynchronous, interleaved execution models, and this is why the asynchronous product has been so named.

Definition - The *asynchronous product* $\otimes_{i \in I} F_i$ of a collection $\langle F_i \rangle_{i \in I}$ of I/O-interfaces is the interface F defined as follows:

$$\begin{aligned}
F &= \{\underline{f} \in \Pi_{i \in I} F_i \text{: at most one } f_i \text{ is an output event}\} \\
\lambda_F &= \langle \lambda_{F_i} \rangle_{i \in I} \\
\mathrm{In}_F &= \{\underline{f} \in F\text{: } \underline{f} \neq \lambda_F \text{ and no } f_i \text{ is an output event}\} \\
\mathrm{Out}_F &= \{\underline{f} \in F\text{: exactly one } f_i \text{ is an output event}\}.
\end{aligned}$$

The maps π_i: $\otimes_{i \in I} F_i \rightarrow F_i$, for $i \in I$, that take a vector $\underline{f}$ to its ith component, are called the *canonical projections* associated with $\otimes_{i \in I} F_i$. ▮

In general, a system interface will not be the entire asynchronous product of the process interfaces, but rather only a sub-interface of the asynchronous product. The reason for using a sub-interface of the asynchronous product as the system interface is to capture possible coupling of events between processes. One kind of coupling that can be modeled in this way is the identification of events of distinct processes. For example, if the output event *out* for process one is to be identified with the input event *in* for process two, then we would include in the system interface the vector $\langle out, in\rangle$, in which process one performs an *out* event at the same time as process two performs an *in* event, but we would exclude from the system interface the event $\langle out, \lambda\rangle$, in which process one performs an *out* event while process two does nothing, and the event $\langle \lambda, in\rangle$, in which process two performs an *in* event while process one does nothing. Other kinds of coupling can also be modeled. For example, if the input event *in* for process two always occurs along with an output event *out* for process one, but the event *out* for process one need not occur along with an *in* event for process two, then the system interface would include the events $\langle out, in\rangle$ and $\langle out, \lambda\rangle$, but would exclude the event $\langle \lambda, in\rangle$.

Our only requirement on the system interface is that to each event of a component process there is some system event that contains the given event as a component. This requirement ensures that each observation over a process interface has a faithful representation as an observation over the system interface.

Definition - An *embedding* of an I/O-interface E into an I/O-interface F is an injective translation $\gamma: E \rightarrow F$ such that $\gamma(\mathrm{In}_E) \subseteq \mathrm{In}_F$ and $\gamma(\mathrm{Out}_E) \subseteq \mathrm{Out}_F$. ∎

(Recall that the fact γ is a translation implies that $\gamma(\lambda_E) = \lambda_F$.)

Definition - A *system interface* for a collection $\langle F_i\rangle_{i\in I}$ of I/O-interfaces is an I/O-interface $E \subseteq \otimes_{i\in I} F_i$ such that

(1) The inclusion map $\gamma: E \rightarrow \otimes_{i\in I} F_i$ is an embedding.

(2) Each map $\pi_i \circ \gamma$ is onto F_i, where $\langle \pi_i\rangle_{i\in I}$ are the canonical projections associated with $\otimes_{i\in I} F_i$.

The collection of maps $\langle \pi_i \circ \gamma\rangle_{i\in I}$ is called the *canonical decomposition map* associated with E. ∎

Each process in an I/O-system is represented by an "I/O-machine," which is a machine that cannot prevent the occurrence of input events. The I/O-machines in an I/O-system are required to be "explicit" in the sense that each nonnull step results in the occurrence of some non-λ step. This assumption is justified because we think of an I/O-system as being a detailed, low-level model, in which all steps taken by processes result in explicit observable events. Later we will apply abstraction maps to the behaviors of I/O-systems to obtain less detailed, higher-level views of system behavior, in which steps can be taken that do not result in observable events.

Definition - An *I/O-machine of I/O-interface E* is a machine M of interface E that is *input-cooperative* in the following sense: For all $q \in Q_M$ and $e \in \mathrm{In}_E$, there exists $r \in Q_M$ such that $\langle q, e, r\rangle \in \mathrm{Trans}_M$. An I/O-machine M of interface E is *explicit* if every step $\langle q, \lambda, r\rangle \in \mathrm{Trans}_M$ has $r = q$. ∎

Definition - An *I/O-system* is a tuple $\mathcal{S} = \langle E, \langle M_i\rangle_{i\in I}\rangle$, where I is a finite, nonempty set of *process indices*, $E \subseteq \otimes_{i\in I} F_i$ is a system interface, and each M_i is an explicit I/O-machine of interface F_i. ∎

We associate with an I/O-system $\mathcal{S} = \langle E, \langle M_i\rangle_{i\in I}\rangle$, a *system machine M* defined as follows:

$$\begin{aligned}
E_M &= E\\
Q_M &= \Pi_{i\in I} Q_{M_i}\\
\mathrm{Init}_M &= \Pi_{i\in I} \mathrm{Init}_{M_i}\\
\mathrm{Trans}_M &= \{\langle \underline{q}, e, \underline{r}\rangle : \langle q_i, \delta_i(e), r_i\rangle \in \mathrm{Trans}_{M_i} \text{ for all } i \in I\},
\end{aligned}$$

where $\langle \delta_i\rangle_{i\in I}$ is the canonical decomposition map associated with E. ∎

Definition - A *computation* for an I/O-system is just a computation for its system machine. ∎

A computation X for an I/O-system projects to computations $X^{(i)}$ for each of its constituent machines in the obvious way.

We will be interested only in the "fair" computations of an I/O-system. To formally define the notion of fairness, suppose $\mathcal{S} = \langle E, \langle M_i\rangle_{i\in I}\rangle$ is an I/O-system and M is the system machine. Suppose $\underline{q} \in Q_M$ is a system state. We say that process i *runs* in a step $\langle \underline{q}, e, \underline{r}\rangle \in \mathrm{Trans}_M$ if $\delta_i(e)$ is an output event for process i. We say that *process i is* *enabled in system state* $\underline{q}$ if there is a step $\langle \underline{q}, e, \underline{r} \rangle \in \text{Trans}_M$ in which process i runs.

Suppose X is a computation for M. Process i is *repeatedly enabled* in X if for all $t \in [0, \infty)$ there exists $t' \in [t, \infty)$ such that process i is enabled in $\text{State}_X(t)$. Process i *repeatedly runs* in X if for all $t \in [0, \infty)$ there exists $t' \in [t, \infty)$ such that process i runs in $\text{Step}_X(t)$.

Definition - A computation X for an I/O-system is *fair* if for each process i in the system, if process i is repeatedly enabled in X, then process i repeatedly runs in X. ▮

5.2 I/O-Behaviors and I/O-Consistency

Each computation of an I/O-system produces an observation over the system interface. We call the set of all observations that are produced in fair computations of an I/O-system the "primitive behavior" of the system. This behavior is called "primitive" because it contains complete detail about the events that occur during a computation of the system.

Definition - The *primitive behavior* $\text{PBeh}(\mathcal{S})$ of a system of I/O-processes $\mathcal{S}$ is the set of all Obs_X where X is a fair computation for $\mathcal{S}$. ▮

By applying abstraction maps to the primitive behaviors, we obtain additional (nonprimitive) behaviors. We call any behavior that is the abstraction of a primitive behavior an "I/O-behavior." An abstraction map can suppress information in a behavior by mapping two distinct events of the same type (either input or output) to the same event, or by mapping an output event to λ. To ensure that an abstraction map faithfully preserves the input/output structure of a behavior, we require that an abstraction map never map an input event to λ, and never map an input event and an output event to the same event. Furthermore, we require that each abstract input event be the image of some concrete input event.

Definition - An *I/O-abstraction map* from the I/O-interface E to the I/O-interface D is a translation $\alpha: E \rightarrow D$ with the following properties:

(1) $\alpha(\text{Out}_E) \subseteq \text{Out}_D \cup \{\lambda_D\}$. (*$\alpha$ preserves outputs*)

(2) $\alpha(\text{In}_E) \subseteq \text{In}_D$. (*$\alpha$ strictly preserves* inputs)

(3) α is onto In_D.

▮

Definition - A behavior $B \in \mathrm{Beh}(D)$ is an *I/O-behavior of interface D* iff there exists a system $\mathcal{S}$ of I/O-processes with system interface E and an I/O-abstraction map $\alpha: E \rightarrow D$ such that $B = \alpha(\mathrm{PBeh}(\mathcal{S}))$. ∎

The following result shows that the class of I/O-behaviors is a kind of completion under I/O-abstraction of the class of primitive behaviors.

Theorem 5.1 - The class of I/O-behaviors contains all primitive behaviors and is closed under I/O-abstraction operators.

Proof - Obvious from the definition of an I/O-behavior and the facts:

(1) Identity translations are I/O-abstraction maps.

(2) If $\alpha: F \rightarrow E$ and $\beta: E \rightarrow D$ are I/O-abstraction maps, then $\beta \circ \alpha$ is an I/O-abstraction map. ∎

By taking the I/O-behaviors as our class of realizable or computable behaviors, we obtain the notion of "I/O-consistency" of specifications.

Definition - A specification S of I/O-interface D is *I/O-consistent* if there exists an I/O-behavior B of interface D such that B satisfies S. ∎

5.3 Machine Characterization of I/O-Behaviors

To obtain techniques for proving the I/O-consistency of state-transition specifications, it is convenient to have a direct characterization, not involving I/O-abstraction maps, of the I/O-behaviors of interface E. Such a characterization is provided by Theorem 5.4 below. Theorem 5.4 states that the I/O-behaviors are exactly the sets of observations produced by "productive step machines," which are I/O-machines plus some scheduling information.

Definition - A *productive step set* for an I/O-machine M of interface E is a set $\mathrm{Prod} \subseteq \mathrm{Trans}_M \cap \mathrm{Steps}(\mathrm{Out}_E \cup \{\lambda_E\}, Q_M)$ that contains no null steps. ∎

Definition - A *productive step machine* (PS-machine) of I/O-interface E is a tuple $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$, where M is an I/O-machine of interface E and $\langle \mathrm{Prod}_i \rangle_{i \in I}$ is a finite, nonempty collection of productive step sets for M, such that $\cup_{i \in I} \mathrm{Prod}_i$ equals the set of all nonnull steps $\langle q, e, r \rangle \in \mathrm{Trans}_M \cap \mathrm{Steps}(\mathrm{Out}_E \cup \{\lambda_E\}, Q_M)$. ∎

Suppose that $\langle M, \langle \text{Prod}_i \rangle_{i \in I} \rangle$ is a PS-machine. The notions of the productive step set Prod_i being enabled in a state of M and running in a step of M are defined in the obvious way. A computation X for M is *fair* if for each $i \in I$, if Prod_i is repeatedly enabled in X then Prod_i repeatedly runs in X. Define the *behavior* $\text{Beh}(M, \langle \text{Prod}_i \rangle_{i \in I})$ of the PS-machine $\langle M, \langle \text{Prod}_i \rangle_{i \in I} \rangle$ to be the set of all Obs_X where X is a fair computation of M.

The following lemma states that every PS-machine has the same behavior as a PS-machine whose productive step sets are pairwise disjoint.

Lemma 5.2 - If $\langle M, \langle \text{Prod}_i \rangle_{i \in I} \rangle$ is a PS-machine of interface E, then there exists a PS-machine $\langle M', \langle \text{Prod}_i' \rangle_{i \in I} \rangle$ of interface E such that the collection $\langle \text{Prod}_i' \rangle_{i \in I}$ is pairwise disjoint and such that $\text{Beh}(M', \langle \text{Prod}_i' \rangle_{i \in I}) = \text{Beh}(M, \langle \text{Prod}_i \rangle_{i \in I})$.

Proof - The idea of the proof is to include a dummy "tag" component in the state of M', so that steps in Prod_i' write i into the tag component. This ensures disjointness, since if $i \neq j$, then steps in Prod_i and Prod_j write different values into the tag component.

Formally, define

$$\begin{aligned}
Q_{M'} &= Q_M \times I \\
\text{Init}_{M'} &= \text{Init}_M \times I \\
\text{Trans}_{M'} &= \{\langle\langle q, k\rangle, e, \langle r, m\rangle\rangle : \text{(1)-(3) below all hold}\} \\
&\qquad \text{(1) } \langle q, e, r\rangle \in \text{Trans}_M \\
&\qquad \text{(2) If } \langle q, e, r\rangle \in \cup_{i \in I} \text{Prod}_i, \text{ then } \langle q, e, r\rangle \in \text{Prod}_m. \\
&\qquad \text{(3) If } \langle q, e, r\rangle \notin \cup_{i \in I} \text{Prod}_i, \text{ then } m = k. \\
\text{Prod}_i' &= \{\langle\langle q, k\rangle, e, \langle r, i\rangle\rangle \in \text{Trans}_{M'} : \langle q, e, r\rangle \in \text{Prod}_i\}
\end{aligned}$$

It is straightforward to check that M' is an I/O-machine of interface E and that the collection $\langle \text{Prod}_i' \rangle_{i \in I}$ is pairwise disjoint.

To show that $\langle M', \langle \text{Prod}_i' \rangle_{i \in I} \rangle$ is a PS-machine, we must show that the Prod_i' cover the nonnull output or λ-steps in $\text{Trans}_{M'}$. If $\langle\langle q, k\rangle, e, \langle r, m\rangle\rangle$ is a nonnull output or λ-step in $\text{Trans}_{M'}$, then either $m \neq k$ or $\langle q, e, r\rangle$ is a nonnull output or λ-step in Trans_M. If $m \neq k$, then $\langle q, e, r\rangle \in \cup_{i \in I} \text{Prod}_i$ by part (3) of the definition of $\text{Trans}_{M'}$ and hence $\langle q, e, r\rangle \in \text{Prod}_m$ by part (2) of the definition of $\text{Trans}_{M'}$. By definition of Prod_m', we have that $\langle\langle q, k\rangle, e, \langle r, m\rangle\rangle \in \text{Prod}_m'$. If $\langle q, e, r\rangle$ is a nonnull output or λ-step in Trans_M, then $\langle q, e, r\rangle \in \cup_{i \in I} \text{Prod}_i$ because the Prod_i cover the nonnull output or λ-steps in Trans_M. By part (2) of the definition of $\text{Trans}_{M'}$, we know that $\langle q, e, r\rangle \in \text{Prod}_m$, and hence $\langle\langle q, k\rangle, e, \langle r, m\rangle\rangle \in \text{Prod}_m{}'$ by definition of $\text{Prod}_m{}'$.

We claim that $\text{Beh}(M', \langle\text{Prod}_i{}'\rangle_{i\in I}) = \text{Beh}(M, \langle\text{Prod}_i\rangle_{i\in I})$.

Case $\text{Beh}(M', \langle\text{Prod}_i{}'\rangle_{i\in I}) \subseteq \text{Beh}(M, \langle\text{Prod}_i\rangle_{i\in I})$:

Each computation X' of $\langle M', \langle\text{Prod}_i{}'\rangle_{i\in I}\rangle$ defines a computation X of $\langle M, \langle\text{Prod}_i\rangle_{i\in I}\rangle$, which we obtain simply by deleting the tag information from X. Suppose X' is fair and that Prod_i is repeatedly enabled in X. It is easy to see from the definition of $\text{Prod}_i{}'$ that if Prod_i is enabled at time t in X, then $\text{Prod}_i{}'$ is enabled at time t in X'. Hence $\text{Prod}_i{}'$ is repeatedly enabled in X', and thus repeatedly runs in X' by the assumption that X' is fair. If $\text{Prod}_i{}'$ runs at time t in X', then by definition of $\text{Prod}_i{}'$ it follows that Prod_i runs at time t in X, so that X is fair.

Case $\text{Beh}(M, \langle\text{Prod}_i\rangle_{i\in I}) \subseteq \text{Beh}(M', \langle\text{Prod}_i{}'\rangle_{i\in I})$:

Given a fair computation X of $\langle M, \langle\text{Prod}_i\rangle_{i\in I}\rangle$, we wish to construct a fair computation X' of $\langle M', \langle\text{Prod}_i{}'\rangle_{i\in I}\rangle$ that generates the same observation. We construct X' from X simply by filling in appropriate tag information to match the occurrence of productive steps in X, however we must do this in such a way that X' is fair.

To construct X', let $f: T \to \text{Steps}(E, Q_M)$ be a history skeleton that spans X, where $T = \langle t_k\rangle_{k\in\mathcal{N}}$. Suppose $\text{Step}_X(t_k) = \langle q_k, e_k, q_{k+1}\rangle$ for each $k \in \mathcal{N}$. By a straightforward inductive construction involving fair scheduling of the elements of I, we can obtain a sequence $\langle m_k\rangle_{k\in\mathcal{N}}$ of elements of I such that $\langle\langle q_k, m_k\rangle, e_k, \langle q_{k+1}, m_{k+1}\rangle\rangle \in \text{Trans}_{M'}$ for all $k \in \mathcal{N}$, and such that if $\langle q_k, e_k, r_k\rangle \in \text{Prod}_i$ for infinitely many $k \in \mathcal{N}$, then $m_k = i$ for infinitely many $k \in \mathcal{N}$. The history skeleton f' that maps t_k to the step $\langle\langle q_k, m_k\rangle, e_k, \langle q_{k+1}, m_{k+1}\rangle\rangle$ then defines the desired fair computation X' of M'. ∎

The lemma below shows that the class of behaviors of PS-machines is closed under I/O-abstraction.

Lemma 5.3 - Given a PS-machine $\langle M, \langle\text{Prod}_i\rangle_{i\in I}\rangle$ of interface E, and an I/O-abstraction map $\alpha: E \to D$, there exists a PS-machine $\langle M', \langle\text{Prod}_i{}'\rangle_{i\in I}\rangle$ of interface D such that $\text{Beh}(M', \langle\text{Prod}_i{}'\rangle_{i\in I}) = \alpha(\text{Beh}(M, \langle\text{Prod}_i\rangle_{i\in I}))$.

Proof - The basic idea of the proof is simple: M' and the $\text{Prod}_i{}'$ are defined by taking the images of M and the Prod_i under α. There is one problem with the straightforward execution of this idea: the Prod_i' might contain null steps. We solve this problem by introducing into the state of M' an "idling counter," which is a boolean component whose only purpose is to change state upon execution of productive steps.

Formally, define $\langle M', \langle \text{Prod}_i' \rangle_{i \in I} \rangle$ as follows:

$Q_{M'} = Q_M \times \{0, 1\}$

$\text{Init}_{M'} = \text{Init}_M \times \{0, 1\}$

$\text{Trans}_{M'} = \{\langle\langle q, b\rangle, \alpha(e), \langle r, c\rangle\rangle$: (1) and (2) below both hold$\}$

(1) $\langle q, e, r\rangle \in \text{Trans}_M$

(2) If $\langle q, e, r\rangle \in \cup_{i\in I} \text{Prod}_i$, then $c = 1 - b$, otherwise $c = b$.

$\text{Prod}_i' = \{\langle\langle q, b\rangle, \alpha(e), \langle r, c\rangle\rangle \in \text{Trans}_{M'} : \langle q, e, r\rangle \in \text{Prod}_i\}$.

We claim that M' is an I/O-machine. It is clear that $\text{Init}_{M'}$ is nonempty. Part (2) of the definition of $\text{Trans}_{M'}$ does not prevent $\text{Trans}_{M'}$ from containing all null steps, since no such step can be in $\cup_{i\in I} \text{Prod}_i$. Thus M' is a machine. To show that M' is input-cooperative, suppose $\langle q, b\rangle \in Q_{M'}$ and $d \in \text{In}_D$. Since α is onto In_D and preserves outputs, there exists $e \in \text{In}_E$ with $\alpha(e) = d$. By the input-cooperative property of M, there exists r with $\langle q, e, r\rangle \in \text{Trans}_M$. Since $\langle q, e, r\rangle \notin \cup_{i\in I} \text{Prod}_i$ by the fact that e is an input event, it follows that $\langle\langle q, b\rangle, d, \langle r, b\rangle\rangle \in \text{Trans}_{M'}$.

We next show that $\langle M', \langle \text{Prod}_i' \rangle_{i\in I}\rangle$ is a PS-machine. By definition $\text{Prod}_i' \subseteq \text{Trans}_{M'}$ for all $i \in I$. Since each step in Prod_i is an output or λ-step and α preserves outputs, it follows that each step in Prod_i' is an output or λ-step. Each Prod_i' contains no null steps because the idling counter is complemented in each step in Prod_i'. To see that every output or λ-step in $\text{Trans}_{M'}$ is in some Prod_i', note that because α strictly preserves inputs, each output or λ-step in $\text{Trans}_{M'}$ cannot be the image of an input step in Trans_M, and therefore must be the image of an output or λ-step in Trans_M. Since the Prod_i cover all output or λ-steps of Trans_M, it follows that the Prod_i' must cover all output or λ-steps of Trans_M'.

We claim that $\text{Beh}(M', \langle \text{Prod}_i' \rangle_{i\in I}) = \alpha(\text{Beh}(M, \langle \text{Prod}_i \rangle_{i\in I}))$.

Case $\alpha(\text{Beh}(M, \langle \text{Prod}_i \rangle_{i\in I})) \subseteq \text{Beh}(M', \langle \text{Prod}_i' \rangle_{i\in I})$:

Each computation X of M maps in an obvious way (by taking the image of the observation part under α, and deleting the idling counter from the state part) to a computation X' of M', such that $\text{Obs}_{X'} = \alpha(\text{Obs}_X)$. It suffices to show that if X is fair, then so is X'. Suppose that X is fair. Fix $i \in I$, and suppose that Prod_i' is repeatedly enabled in X'. We claim that Prod_i' repeatedly runs in X'. By definition, Prod_i' is enabled in state q iff Prod_i is enabled in state q. It follows that Prod_i is repeatedly enabled in X, and hence by fairness of X, that Prod_i repeatedly runs in X. By definition of Prod_i', if $\text{Step}_X(t) \in \text{Prod}_i$, then $\text{Step}_{X'}(t) \in \text{Prod}_i'$. Thus Prod_i' repeatedly runs in X'.

Case $\text{Beh}(M', \langle\text{Prod}_i'\rangle_{i\in I}) \subseteq \alpha(\text{Beh}(M, \langle\text{Prod}_i\rangle_{i\in I}))$:

Suppose that $x' \in \text{Beh}(M', \langle\text{Prod}_i'\rangle_{i\in I})$, and let X' be a fair computation of M' in which the observation x' is generated. We will construct a fair computation X of M, such that $\alpha(\text{Obs}_X) = x'$. The idea is simply to choose inverse images under α of the steps in X', however this must be done carefully to ensure fairness.

Let $T = \langle t_k\rangle_{k\in\mathcal{N}}$ be a skeletal sequence that spans X'. Suppose $\text{Step}_{X'}(t_k) = \langle\langle q_k, b_k\rangle, d_k, \langle r_k, c_k\rangle\rangle$ for each $k \in \mathcal{N}$.

For each k, since $\langle\langle q_k, b_k\rangle, d_k, \langle r_k, c_k\rangle\rangle \in \text{Trans}_{M'}$, we can select e_k such that $d_k = \alpha(e_k)$ and $\langle q_k, e_k, r_k\rangle \in \text{Trans}_M$. Because α might map two different e's to the same d, we can't necessarily select the e_k in such a way that for each $i \in I$, the step $\langle q_k, e_k, r_k\rangle \in \text{Prod}_i$ iff $\langle\langle q_k, b_k\rangle, d_k, \langle r_k, c_k\rangle\rangle \in \text{Prod}_i'$. However, by making sure that we don't persistently neglect some Prod_i', we can select the e_k in such a way that for each $i \in I$, if $\langle\langle q_k, b_k\rangle, d_k, \langle r_k, c_k\rangle\rangle \in \text{Prod}_i'$ for infinitely many k, then $\langle q_k, e_k, r_k\rangle \in \text{Prod}_i$ for infinitely many k.

The function f that takes t_k to the step $\langle q_k, e_k, r_k\rangle$ is a history skeleton over E_M and Q_M. By Lemma 3.5 there is a unique history X such that f spans X. It is easily verified that X is a computation of M, with $\alpha(\text{Obs}_X) = x'$. To show fairness, fix $i \in I$ and suppose that Prod_i is repeatedly enabled in X. We claim that Prod_i repeatedly runs in X. From the definition of Prod_i' we know that Prod_i' is repeatedly enabled in X'. By the fairness of X' we know that Prod_i' repeatedly runs in X'. This implies that $\langle q_k, d_k, r_k\rangle \in \text{Prod}_i'$ for infinitely many k, and hence by construction that $\langle q_k, e_k, r_k\rangle \in \text{Prod}_i$ for infinitely many k. It follows that Prod_i repeatedly runs in X. ∎

The following theorem is our desired characterization of the I/O-behaviors: a behavior is an I/O-behavior iff it is the behavior of a PS-machine.

Theorem 5.4 - Suppose D is an I/O-interface. Then a behavior $B \in \text{Beh}(D)$ is an

I/O-behavior of interface D iff B = Beh(M, $\langle \mathrm{Prod}_i \rangle_{i \in I}$) for some PS-machine $\langle M$, $\langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$ of interface D.

Proof - => Since the class of behaviors of PS-machines is closed under I/O-abstraction by Theorem 5.1, it suffices to show that every primitive behavior B is the behavior of a PS-machine. Suppose B = PBeh($\mathcal{S}$), where $\mathcal{S} = \langle E, \langle M_i \rangle_{i \in I} \rangle$ is an I/O-system. We associate a PS-machine $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$ with $\mathcal{S}$ as follows: The machine M is the system machine for $\mathcal{S}$. The set Prod_i is the subset of Trans_M in which process i runs. Since a step in which process i runs is always an output step, it is clear that Prod_i is a productive step set for M. Since every nonnull output or λ-step in Trans_M is in fact an output step for some process $i \in I$, and hence is in Prod_i, it follows that the Prod_i sets cover the nonnull output or λ-steps in Trans_M.

It is obvious that the set of fair computations of the system $\mathcal{S}$ is exactly the set of fair computations of the PS-machine $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$, and thus PBeh($\mathcal{S}$) = Beh($M$, $\langle \mathrm{Prod}_i \rangle_{i \in I}$).

<= Suppose that $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$ is a PS-machine of interface D. We construct an I/O-system $\mathcal{S} = \langle E, \langle M_i \rangle_{i \in I} \rangle$ and an I/O-abstraction map α: $E \rightarrow D$, such that Beh(M, $\langle \mathrm{Prod}_i \rangle_{i \in I}$) = α(PBeh($\mathcal{S}$)).

Without loss of generality we make the following three assumptions about $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$:

(1) The set Init_M of initial states for M contains exactly one state q_0.

(2) For all $q \in Q_M$ and all $e \in \mathrm{In}_D$, there is a unique $r \in Q_M$ such that $\langle q, e, r \rangle \in \mathrm{Trans}_M$.

(3) $\mathrm{Prod}_i \cap \mathrm{Prod}_j = \emptyset$ for $i \neq j$.

A PS-machine $\langle M', \langle \mathrm{Prod}_i' \rangle_{i \in I} \rangle$ that does not have these three properties can easily be transformed, without changing its behavior, into a PS-machine $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$ that does have these properties. We first obtain properties (1) and (2) by buffering input events in an input queue in the order that they occur so that the change of state associated with an input event is just to append the event to the end of the input queue. All nondeterministic choice, including the choice between multiple initial states, is absorbed into the output steps.

Formally, we transform the PS-machine $\langle M', \langle \text{Prod}_i' \rangle_{i \in I} \rangle$ into a PS-machine $\langle M'', \langle \text{Prod}_i'' \rangle_{i \in I} \rangle$ by defining $\text{State}_{M''}$ to consist of all pairs $\langle q, u \rangle$, where q is either an element of $\text{State}_{M'}$ or the distinguished symbol $\perp$, and $u \in \text{In}_E{}^*$. The single initial state of M'' is the state $\langle \perp, \lambda \rangle$. The transition relation $\text{Trans}_{M''}$ consists of all steps $\langle \langle q, u \rangle, e, \langle r, v \rangle \rangle$ such that

- If $e \in \text{In}_E$ then $r = q$ and $v = ue$.
- If $e \in \text{Out}_E \cup \{\lambda_E\}$, then $v = \lambda$, and either
  (a) $\langle q, ue, r \rangle$ is in $\text{Trans}_{M'}{}^*$, or
  (b) $q = \perp$ and $\langle s, ue, r \rangle \in \text{Trans}_{M'}{}^*$ for some s in $\text{Init}_{M'}$.

The set Prod_i'' consists of all steps $\langle \langle q, u \rangle, e, \langle r, v \rangle \rangle \in \text{Trans}_{M''}$ such that for some state s of M', the step $\langle s, e, r \rangle \in \text{Prod}_i'$, and in addition, either $q \neq \perp$ and the step $\langle q, u, s \rangle$ is in $\text{Trans}_{M'}{}^*$, or $q = \perp$ and for some $q' \in \text{Init}_{M'}$ the step $\langle q', u, s \rangle$ is in $\text{Trans}_{M'}{}^*$.

Once $\langle M'', \langle \text{Prod}_i'' \rangle_{i \in I} \rangle$ with properties (1) and (2) is obtained, it can be transformed into $\langle M, \langle \text{Prod}_i \rangle_{i \in I} \rangle$ with all three properties by an application of Lemma 5.2.

We now proceed to the construction of $\mathcal{S}$. The idea is as follows: The system $\mathcal{S}$ will contain one process for each $i \in I$. The processes in $\mathcal{S}$ perform a lock-step simulation of the machine M. The interface for each of the processes in the system $\mathcal{S}$ consists of the null event, the input events of D, and the set of all productive steps for M. The input events for process i will be the input events of D and the steps in $\bigcup_{j \in I-\{i\}} \text{Prod}_j$. The output events for process i will be the steps in Prod_i. Each process keeps track of the current simulated state of M, and permits an output event to occur only if the event corresponds to a step of M from the current simulated state of M. To ensure that the input-cooperative property holds, process i imposes no requirements on the state from which a step in Prod_j can occur, if $j \neq i$.

Formally, define the I/O-interfaces F_i as follows:

$$F_i = \{\lambda_{F_i}\} + \text{In}_D + \bigcup_{j \in I} \text{Prod}_j$$
$$\text{In}_{F_i} = \text{In}_D + (\bigcup_{j \in I-\{i\}} \text{Prod}_j)$$
$$\text{Out}_{F_i} = \text{Prod}_i$$

Define the system interface $E \subseteq F = \otimes_{i \in I} F_i$ as follows:

$$E = \{f \in F : f_i = f_j \text{ for all } i, j \in I\} \cup \{\lambda_F\}$$
$$\lambda_E = \lambda_F$$
$$\text{In}_E = E \cap \text{In}_F$$

$\text{Out}_E = E \cap \text{Out}_F$

It is easy to see that the inclusion map $\gamma: E \to F$ is an embedding. For each $f \in \text{In}_{F_i}$, the identically f vector $\langle f \rangle_{i \in I}$ is in E. By the assumption that the Prod_i are pairwise disjoint, it follows that the identically f vector $\langle f \rangle_{i \in I}$ is in E for each $f \in \text{Out}_{F_i}$ as well. This shows that $\pi_i \circ \gamma$ is onto F_i, where the π_i are the canonical projections associated with F.

Define $\alpha: E \to D$ to be the translation that behaves in the following way on the identically f vector $\langle f \rangle_{i \in I} \in E$:

- If $f \in \text{In}_D$, then $\alpha(\langle f \rangle_{i \in I}) = f$.
- If $f = \langle q, d, r \rangle \in \text{Prod}_i$ for some $i \in I$, then $\alpha(\langle f \rangle_{i \in I}) = d$.

We claim that α is an I/O-abstraction map. It is clear that α is onto In_D. The map α preserves outputs because if the identically f vector is an output event of E, then $f \in \text{Prod}_i$ for some i and hence f is an output or λ-step. To show that α strictly preserves inputs, suppose the identically f vector $\langle f \rangle_{i \in I}$ is an input event of E. Then $f \in \text{In}_D$, so $\alpha(\langle f \rangle_{i \in I}) = f \in \text{In}_D$.

The machines M_i are defined as follows:

$E_{M_i} = F_i$

$Q_{M_i} = Q_M$

$\text{Init}_{M_i} = \text{Init}_M = \{q_0\}$

$\text{Trans}_{M_i} = \{\langle q, f, r \rangle\text{: one of (1)-(4) below holds}\}$

(1) $f = \lambda_{F_i}$ and $r = q$,

(2) $f \in \text{In}_D$ and $\langle q, f, r \rangle \in \text{Trans}_M$.

(3) $f = \langle q', d, r \rangle \in \text{Prod}_j$ for some $j \neq i$.

(4) $f = \langle q, d, r \rangle \in \text{Prod}_i$.

Obviously M_i is a machine and every step $\langle q, \lambda_{F_i}, r \rangle \in \text{Trans}_{M_i}$ has $r = q$. To see that M_i is input-cooperative, suppose $q \in Q_{M_i}$ and $f \in \text{In}_{F_i}$. Then either $f \in \text{In}_D$ or $f \in \text{Prod}_j$ for some $j \neq i$. If $f \in \text{In}_D$ then f is enabled in state q by part (2) of the definition of Trans_{M_i} because M is input-cooperative. If $f = \langle q', d, r \rangle \in \text{Prod}_j$ for some $j \neq i$, then f is enabled in state q by part (3) of the definition of Trans_{M_i}.

A straightforward induction establishes that if $\underline{q}$ is a reachable state of the system $\mathcal{Y}$, then $q_i = q_j$ for all $i, j \in I$. This argument uses the assumed uniqueness of the initial state of M, plus the assumption that a state q and an event $e \in \text{In}_E$ uniquely determine a state r such that $\langle q, e, r \rangle \in \text{Trans}_M$. Intuitively, since the processes in $\mathcal{Y}$ do not interact with each other during input steps, the uniqueness assumptions are needed to ensure that all processes reach the same new state in each such step.

There is an obvious correspondence between the steps of the machine M and the steps of the system $\mathcal{S}$. Specifically, each step $s = \langle q, d, r\rangle$ of M determines a step $s' = \langle \underline{q}, e, \underline{r}\rangle$ of $\mathcal{S}$ under the definitions:

- $\underline{q}$ is the identically q vector
- $\underline{r}$ is the identically r vector
- $e = \lambda_E$, if s is null
  $= \langle d\rangle_{i\in I}$, if $d \in \mathrm{In}_D$
  $= \langle s\rangle_{i\in I}$, if s is a nonnull output or λ-step.

It easy to see that a step s of M is enabled in state q of M iff the corresponding step s' is enabled for the system $\mathcal{S}$ in state $\langle q\rangle_{i\in I}$. The correspondence between the steps of M and the steps of $\mathcal{S}$ therefore defines a bijection between the set of computations of M and the set of computations of $\mathcal{S}$, such that if X' is a computation of $\mathcal{S}$ and X is the corresponding computation of M, then $\mathrm{Obs}_X = \alpha(\mathrm{Obs}_{X'})$. Furthermore, a step s of M is in Prod_i iff process i runs in the corresponding step s' of $\mathcal{S}$, so that fairness is preserved in both directions of this correspondence. It follows that $\alpha(\mathrm{PBeh}(\mathcal{S})) = \mathrm{Beh}(M, \langle \mathrm{Prod}_i\rangle_{i\in I})$. ∎

The following two properties of I/O-behaviors are easily derived from the PS-machine characterization.

Corollary 5.5 If B is an I/O-behavior of interface E, then $B \neq \emptyset$.

Proof - Suppose $B = \mathrm{Beh}(M, \langle \mathrm{Prod}_i\rangle_{i\in I})$. It suffices to show that there is a fair computation of M. We construct a sequence $q_0, q_1, \ldots$ of states of M, and a sequence $e_0, e_1, \ldots$ of events of E, such that the following properties hold:

(1) $\langle q_k, e_k, q_{k+1}\rangle \in \mathrm{Trans}_M$ for all $k \in \mathcal{N}$,

(2) For each $i \in I$, either Prod_i is enabled in only finitely many of the q_k, or else the step $\langle q_k, e_k, q_{k+1}\rangle$ is in Prod_i for infinitely many k.

Letting $t_k = k$ for each natural number k and applying Lemma 3.5 yields a fair computation of M.

To construct the q_k and e_k, first let $q_0 \in \mathrm{Init}_M$ be chosen arbitrarily. We maintain a running assignment of priorities to the elements of A so that at each stage of the construction i is more urgent than j iff a step in Prod_i has been chosen less recently than a step in Prod_j. At stage k, where $k \geq 0$, we choose e_k and q_{k+1} so that $\langle q_k, e_k, q_{k+1}\rangle \in \mathrm{Prod}_i$, where i is the most urgent element of I such that Prod_i is enabled in state q_k. If no Prod_i is enabled in state q_k, then we let $e_k = \lambda$ and q_{k+1}

A behavior B is *asynchronous* if whenever $x \in B$ and f: $[0, \infty) \rightarrow [0, \infty)$ is an order-isomorphism, then $x \circ f \in B$.

Corollary 5.6 - I/O-behaviors are asynchronous.

Proof - Straightforward from the observation that if X is a fair computation of a PS-machine $\langle M, \langle \mathrm{Prod}_i\rangle_{i \in I}\rangle$ and f: $[0, \infty) \rightarrow [0, \infty)$ is an order-isomorphism, then $X \circ f$ is also a fair computation of $\langle M, \langle \mathrm{Prod}_i\rangle_{i \in I}\rangle$. ■ $= q_k$. ■

5.4 Examples of I/O-Behaviors

In this section we give two examples of I/O-behaviors and an example of a behavior that is not an I/O-behavior.

Example 1: An I/O-Behavior:

As an example of how I/O-behaviors can be used to model a system capable of satisfying eventuality requirements, imagine that we wish to model the behavior of a "black box" to which input stimuli can be applied by pressing a single button, and from which output can be observed by flashes of a single light bulb. The black box has the property that every press of the button is later followed by a flash of the light bulb, and no flashes of the bulb occur unless the button has been pressed at least once since the time of the most recent previous flash.

The interface of such a black box is the I/O-interface E with $E = \{\lambda, \text{button}, \text{flash}\}$, $\mathrm{In}_E = \{\text{button}\}$ and $\mathrm{Out}_E = \{\text{flash}\}$. The behavior of the black box is defined by a PS-machine M of interface E. Intuitively, a push of the button sets a flag in the state of M to true. A flash of the light can occur only when the flag is true, and causes the flag to be reset to false. There is one productive step set *Prod*, which contains exactly those steps in which flashes occur.

Formally,

$$\begin{aligned}
E_M &= E \\
Q_M &= \{\text{true}, \text{false}\} \\
\text{Init}_M &= \{\text{false}\} \\
\text{Trans}_M &= \{\langle q, \text{button}, r\rangle : r = \text{true}\} \cup \{\langle \text{true}, \text{flash}, \text{false}\rangle\} \cup \\
&\quad \{\langle q, \lambda, q\rangle : q \in Q_M\}. \\
\text{Prod} &= \{\langle q, \text{flash}, r\rangle \in \text{Trans}_M\}
\end{aligned}$$

That $\langle M, \text{Prod}\rangle$ is a PS-machine of interface E is easily checked.

Let $B = \text{Beh}(M, \text{Prod})$, so that B is an I/O-behavior. Through analysis of the fair computations of M it can be shown that an observation $x \in \text{Obs}(E)$ is in B iff there is a surjective total function f: $\{t \in [0, \infty): x(t) = \text{button}\} \rightarrow \{t' \in [0, \infty): x(t') = \text{flash}\}$ such that for all $t \in [0, \infty)$, $f(t)$ is the least $t' \in (t, \infty)$ such that $x(t') = \text{flash}$. That is, an observation x is in B provided that in x, every push of the button "causes" a future flash of the light, and every flash of the light is caused by some collection of recent past pushes of the button.

Example 2: Two Productive Step Sets

We can give an example of an I/O-behavior that is not the behavior of a PS-machine with one productive step set. Let the interface E be defined by: $E = \{\lambda, \text{button}, \text{flash1}, \text{flash2}\}$, where $\text{In}_E = \{\text{button}\}$ and $\text{Out}_E = \{\text{flash1}, \text{flash2}\}$. Let B be the set of all $x \in \text{Obs}(E)$ such that the following properties hold:

(1) Occurrences of *flash1* appear only between the 2*k*th and $2k+1$st occurrences of *button*, where $k \in \mathcal{N}$.

(2) Occurrences of *flash2* appear only between the $2k+1$st and $2(k+1)$st occurrence of *flash*.

(3) x contains infinitely many occurrences either of *flash1* or *flash2*

(4) If x contains infinitely many occurrences of *button*, then it contains infinitely many occurrences of both *flash1* and *flash2*.

It is straightforward to show that B is the behavior of a PS-machine of interface E with two productive step sets, one that governs the occurrence of *flash1* events and one that governs the occurrence of *flash2* events.

Suppose B is the behavior of a PS-machine $\langle M, \text{Prod}\rangle$ with one productive step set. Construct a computation of M by repeating the following procedure: Run M until a *flash1* event is produced, then run M for two steps containing a *button* input. It is always possible to obtain the *flash1* events in this construction, since otherwise we could construct a fair computation in which only finitely many *flash1* events and no *flash2* events occur. It is always possible to run the *button* events by the input-cooperative property of M.

The above construction yields a computation X of M that must be fair, since it contains infinitely many steps in which the output event *flash1* occurs, and which must be in the single productive step set *Prod* because *Prod* contains all output steps of M. However, X generates an observation in which infinitely many *button* events occur, but no *flash2* events occur.

Example 3: A Non-I/O-Behavior

We can also give an example of a set that is demonstrably *not* a I/O-behavior. Define the I/O-interface E as follows:

$$E = \{\lambda, \text{button}, \text{flash}\}$$
$$\text{In}_E = \{\text{button}\}$$
$$\text{Out}_E = \{\text{flash}\}.$$

Let the behavior $B \in \text{Beh}(E)$ be the set of all $x \in \text{Obs}(E)$ such that x contains an infinite number of occurrences of *flash*, and such that either the number of occurrences of *button* in x is finite or (# *flash*es in x on the interval $[0, t)$)/ (# *button*s in x on the interval $[0, t)$) $\rightarrow 0$ as $t \rightarrow \infty$.

We argue that B is not an I/O-behavior of interface E. Suppose $\langle M, \langle \text{Prod}_i\rangle_{i\in I}\rangle$ is a PS-machine of interface E, whose behavior is B. Construct a computation X for M by repeating the following procedure: Run M without input until a *flash* event is produced, then run M for one step with a *button* input. We can run M until a *flash* event is produced by always trying to take steps in which *flash* events are produced, if possible, otherwise taking some other productive step. During this construction, we make sure to use a fair scheduling algorithm to determine which of the Prod_i should be executed at each step. We can never reach a state in which no productive steps are enabled, otherwise we could construct a fair computation in which only finitely many *flash* events are produced. We can run M at any time with a *button* input by the input-cooperative property of M.

The above construction yields a computation X of M that must be fair, since the fair scheduling of the Prod_i ensures that every repeatedly enabled Prod_i will be repeatedly run. However, computation X generates an observation x that contains infinitely many occurrences of *button* events, and in which the ratio of the density of *flash* events to *button* events approaches one in the limit, rather than zero. This contradicts the assumption that $\mathrm{Beh}(M, \langle \mathrm{Prod}_i \rangle_{i \in I}) = B$.

5.4.1 Proving I/O-Consistency

From the PS-machine characterization of the I/O-behaviors we obtain the following test for I/O-consistency of subset specifications.

Theorem 5.7 - Suppose that S is a subset specification of I/O-interface E. Then S is I/O-consistent iff there exists a PS-machine $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$ of interface E such that $\mathrm{Beh}(M, \langle \mathrm{Prod}_i \rangle_{i \in I}) \subseteq \mathcal{O}(S)$.

Proof - Obvious. ∎

If $S = \langle M, V \rangle$ is a state-transition specification, then to show the I/O-consistency of S, it suffices to define a collection of productive step sets for M, such that every fair computation of M is in the set V of valid computations.

Corollary 5.8 - Suppose that $S = \langle M, V \rangle$ is a state-transition specification of I/O-interface E. Suppose that $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$ is a PS-machine of interface E. If every fair computation of M is in V, then S is I/O-consistent.

Proof - Since $\langle M, \langle \mathrm{Prod}_i \rangle_{i \in I} \rangle$ is a PS-machine of interface E, it follows that $\mathrm{Beh}(M, \langle \mathrm{Prod}_i \rangle_{i \in I})$ is a I/O-behavior of interface E. Since every fair computation of M is in V, we know that $\mathrm{Beh}(M, \langle \mathrm{Prod}_i \rangle_{i \in I}) \subseteq \mathcal{O}(S)$. By Theorem 5.7, S is I/O-consistent. ∎

To illustrate the use of this result, we apply it to a simple example specification: A *neuron* is a module with a single input event *in*, and a single output event *out*. The state set for the neuron is the set {ff, tt}. At any instant of time, if the state of the neuron is *tt*, then the neuron is said to be *excited*, otherwise the neuron is said to be *inhibited*. Initially the neuron is excited. An *in* event can occur at any time, and causes the neuron to become inhibited. If the neuron is excited, then it can *fire*, producing an *out* event, and then becoming inhibited. The neuron should satisfy the condition, "If the neuron becomes excited and remains that way, then eventually it will fire."

The neuron module description can be formalized as a state-transition specification.

$$
\begin{aligned}
E^{\mathrm{NEU}} &= \{\lambda, \mathrm{in}, \mathrm{out}\}\\
\mathrm{In}^{\mathrm{NEU}} &= \{\mathrm{in}\}\\
\mathrm{Out}^{\mathrm{NEU}} &= \{\mathrm{out}\}\\
Q^{\mathrm{NEU}} &= \{\mathrm{ff}, \mathrm{tt}\}\\
\mathrm{Init}^{\mathrm{NEU}} &= \{\mathrm{tt}\}
\end{aligned}
$$

A step $\langle q, e, r\rangle \in \mathrm{Trans}^{\mathrm{NEU}}$ iff either $e = \lambda$ and $r = q$ or one of the conditions (in), (out) below holds:

(in) $e = \mathrm{in}$ and $r = \mathrm{ff}$

(out) $e = \mathrm{out}$ and $q = \mathrm{tt}$

The neuron module validity condition is defined by:

$$\mathrm{Valid}^{\mathrm{NEU}} \equiv \Box(\Box(\mathbf{Now} = \mathrm{tt}) \rightarrow \Diamond(\mathbf{Occurs} = \mathrm{out})),$$

To show the I/O-consistency of the neuron specification, we define a single productive step set $\mathrm{Prod}^{\mathrm{NEU}}$ as follows:

$$\langle q, e, r\rangle \in \mathrm{Prod}^{\mathrm{NEU}} \text{ iff } e = \mathrm{out},\ q = \mathrm{tt}, \text{ and } r = \mathrm{ff}.$$

It is clear by inspection that M^{NEU} is input-cooperative, and that $\mathrm{Prod}^{\mathrm{NEU}}$ is a productive step set for M^{NEU}. To show the I/O-consistency of the neuron specification, we must show that every fair computation of M^{NEU} is valid. That is,

$$\mathrm{Comp}^{\mathrm{NEU}} \wedge \mathrm{Fair}^{\mathrm{NEU}} \models \mathrm{Valid}^{\mathrm{NEU}},$$

where

$$\mathrm{Fair}^{\mathrm{NEU}} \equiv \Box\Diamond\mathrm{Enabled}^{\mathrm{NEU}}(\mathbf{Now}) \rightarrow \Box\Diamond\mathrm{Prod}^{\mathrm{NEU}}(\mathbf{Now}, \mathbf{Occurs}, \mathbf{After})$$

$$\mathrm{Enabled}^{\mathrm{NEU}}(q) \equiv (\exists e \in E^{\mathrm{NEU}}, r \in Q^{\mathrm{NEU}})\ \mathrm{Prod}^{\mathrm{NEU}}(q, e, r).$$

We claim the stronger property

$$\mathrm{Fair}^{\mathrm{NEU}} \models \mathrm{Valid}^{\mathrm{NEU}}.$$

To show this, we use the neuron module specification and the definition of $\mathrm{Prod}^{\mathrm{NEU}}$ to expand the term $\mathrm{Fair}^{\mathrm{NEU}}$. From the definition of $\mathrm{Prod}^{\mathrm{NEU}}$ we obtain

$$\text{Enabled}^{\text{NEU}}(q) \equiv q = \text{tt},$$

and hence that

$$\text{Fair}^{\text{NEU}} \equiv \Box\Diamond(\textbf{Now} = \text{tt}) \rightarrow \Box\Diamond(\textbf{Now} = \text{tt} \wedge \textbf{Occurs} = \text{out} \wedge \textbf{After} = \text{ff}).$$

By straightforward temporal and propositional reasoning it is now easy to see that

$$\Box\Diamond(\textbf{Now} = \text{tt}) \rightarrow \Box\Diamond(\textbf{Now} = \text{tt} \wedge \textbf{Occurs} = \text{out} \wedge \textbf{After} = \text{ff})$$
$$\models \Box(\Box(\textbf{Now} = \text{tt}) \rightarrow \Diamond(\textbf{Occurs} = \text{out})),$$

That is, if we suppose that

(1) whenever the state repeatedly takes on the value one then it is also repeatedly the case that an *out* event occurs (which takes the state from one to zero), then we are entitled to conclude that

(2) whenever the state is persistently one after some instant, then there is a later instant at which an *out* event occurs.

We can use the PS-machine characterization of the I/O-behaviors to show the I/O-inconsistency of a slightly stronger version of the neuron specification, obtained by using the stronger validity condition

$$\text{Valid}_{\text{st}}^{\text{NEU}} \equiv \Box(\textbf{Now} = \text{tt} \rightarrow \Diamond(\textbf{Occurs} = \text{out})).$$

This condition states that if the neuron is ever excited for a single instant, then it must eventually fire. Suppose there is a PS-machine $\langle M, \langle \text{Prod}_i \rangle_{i \in I} \rangle$ of interface E^{NEU} such that $\text{Beh}(M, \langle \text{Prod}_i \rangle_{i \in I})$ satisfies the strong neuron specification. Construct a computation of M as follows: Run M for one step with input *in*, and then repeatedly run productive steps of M if possible, otherwise null steps, being sure to schedule the occurrences of Prod_i fairly. The result is a fair computation X of M.

Since the observation $x = \text{Obs}_X$ satisfies the strong neuron specification, there must exist a valid computation X' of M^{NEU} such that $\text{Obs}_{X'} = x$. In X', the neuron module is excited at time 0, an *in* event occurs at time 0, and no input events occur after time 0. Consequently, the neuron module is inhibited after time 0, and thus no *out* events can appear in x because X' is a computation of M^{NEU}. Thus, in the computation X' of M^{NEU}, the neuron module is excited at time 0 but no *out* events subsequently occur. This means that the computation X' of M^{NEU} fails to satisfy the validity condition $\text{Valid}_{\text{st}}^{\text{NEU}}$, a contradiction. We conclude that the PS-machine $\langle M, \langle \text{Prod}_i \rangle_{i \in I} \rangle$ cannot exist and the strong neuron specification is I/O-inconsistent.

5.4.2 I/O-Consistency of the Specification SC

As an extended example of an I/O-consistency proof, we prove the I/O-consistency of the synchronizer component module specification. For the productive step sets, we use the sets Prod_{run}, $\text{Prod}_{\text{token\_out}}$, and $\text{Prod}_{\text{request\_out}}$, defined as follows:

$$\text{Prod}_{\text{run}}(q, e, r) \quad \equiv e = \text{run} \wedge \text{Trans}^{\text{SC}}(q, e, r)$$
$$\text{Prod}_{\text{token\_out}}(q, e, r) \quad \equiv e = \text{token\_out} \wedge \text{Trans}^{\text{SC}}(q, e, r)$$
$$\text{Prod}_{\text{request\_out}}(q, e, r) \quad \equiv e = \text{request\_out} \wedge \text{Trans}^{\text{SC}}(q, e, r).$$

It is easily checked that $\langle M^{\text{SC}}, \langle \text{Prod}_{\text{run}}, \text{Prod}_{\text{token\_out}}, \text{Prod}_{\text{request\_out}}\rangle\rangle$ is a PS-machine of interface E^{SC}.

We must show that each fair computation is valid; that is,

$$\text{Comp}^{\text{SC}}k \wedge \text{Fair}^{\text{SC}}_{\text{run}} \wedge \text{Fair}^{\text{SC}}_{\text{token\_out}} \wedge \text{Fair}^{\text{SC}}_{\text{request\_out}} \models \text{Valid}^{\text{SC}},$$

where

$$\text{Fair}^{\text{SC}}_{\text{run}} \equiv \Box\Diamond\text{Enabled}^{\text{SC}}_{\text{run}}(\textbf{Now}) \rightarrow \Box\Diamond\text{Prod}^{\text{SC}}_{\text{run}}(\textbf{Now}, \textbf{Occurs}, \textbf{After})$$
$$\text{Fair}^{\text{SC}}_{\text{token\_out}} \equiv \Box\Diamond\text{Enabled}^{\text{SC}}_{\text{token\_out}}(\textbf{Now}) \rightarrow \Box\Diamond\text{Prod}^{\text{SC}}_{\text{token\_out}}(\textbf{Now}, \textbf{Occurs}, \textbf{After})$$
$$\text{Fair}^{\text{SC}}_{\text{request\_out}} \equiv \Box\Diamond\text{Enabled}^{\text{SC}}_{\text{request\_out}}(\textbf{Now}) \rightarrow \Box\Diamond\text{Prod}^{\text{SC}}_{\text{request\_out}}(\textbf{Now}, \textbf{Occurs}, \textbf{After})$$

and each $\text{Enabled}^{\text{SC}}_i(q)$, where $i \in \{\text{run}, \text{token\_out}, \text{request\_out}\}$, is a formula that expresses the conditions under which $\text{Prod}^{\text{SC}}_i$ is enabled in state q. Using the definitions of the Prod_i given above, we derive the following expressions for $\text{Enabled}_{\text{run}}$, $\text{Enabled}_{\text{token\_out}}$, and $\text{Enabled}_{\text{request\_out}}$:

$$\text{Enabled}_{\text{run}}(q) \quad \equiv q(\text{ustate}) = \text{trying} \wedge q(\text{token}) \neq 0$$
$$\text{Enabled}_{\text{token\_out}}(q) \quad \equiv q(\text{ustate}) \neq \text{running} \wedge q(\text{token}) \neq 0$$
$$\text{Enabled}_{\text{request\_out}}(q) \quad \equiv q(\text{token}) = 0$$

To show

$$\text{Comp}^{\text{SC}}k \wedge \text{Fair}^{\text{SC}}_{\text{run}} \wedge \text{Fair}^{\text{SC}}_{\text{token\_out}} \wedge \text{Fair}^{\text{SC}}_{\text{request\_out}} \models \text{Valid}^{\text{SC}},$$

we assume $\text{Comp}^{\text{SC}}k$, Rely^{SC}, $\neg\text{Guar}^{\text{SC}}$, Fair_{run}, $\text{Fair}_{\text{token\_out}}$, and $\text{Fair}_{\text{request\_out}}$, and derive a contradiction. That is, we consider a fair computation in which the synchronizer component module rely-conditions are satisfied, but in which the guarantee-conditions are not satisfied. If $\neg\text{Guar}^{\text{SC}}$ holds, then either

(A) $\quad \neg\Box(\textbf{Now}(\text{ustate}) = \text{trying} \rightarrow \Diamond(\textbf{Now}(\text{ustate}) \neq \text{trying}))$

or

(B) $\neg\Box(\mathbf{Occurs} = \text{request\_in}) \rightarrow \Diamond(\mathbf{Occurs} = \text{token\_out}))$.

Thus the proof can be split into two cases, one headed by assumption (A), and the other by assumption (B).

Case (A): Suppose that (A) holds. Then by temporal reasoning, we have

$$\Diamond(\mathbf{Now}(\text{ustate}) = \text{trying} \wedge \Box(\mathbf{Now}(\text{ustate}) = \text{trying}))$$

(*) $\Diamond\Box(\mathbf{Now}(\text{ustate}) = \text{trying})$

That is, it is persistently the case that the user process is trying. By definition of Trans^{SC}, the following is valid:

$$\text{Comp}^{SC}{}_k \models \Box(\mathbf{Occurs} = \text{run} \rightarrow \mathbf{After}(\text{ustate}) \neq \text{trying})$$

and thus, using the temporal tautology $\models \Box(\varphi(\mathbf{After}) \rightarrow \Diamond\varphi(\mathbf{Now}))$, that

$$\text{Comp}^{SC}{}_k \models \Diamond\Box(\mathbf{Now}(\text{ustate}) = \text{trying}) \rightarrow \Diamond\Box(\mathbf{Occurs} \neq \text{run}).$$

Intuitively, since occurrence of *run* results in the user process leaving the *trying* state, if the user process is persistently *trying*, then it must be the case that a *run* event persistently does not occur. Applying this to formula (*) yields

$$\Diamond\Box(\mathbf{Occurs} \neq \text{run} \wedge \mathbf{Now}(\text{ustate}) = \text{trying})$$

That is, it is persistently the case that the user process is trying but a *run* event never occurs. Using the definition of Prod_{run}, we conclude

$$\Diamond\Box(\neg\text{Prod}_{\text{run}}(\mathbf{Now}, \mathbf{Occurs}, \mathbf{After}) \wedge \mathbf{Now}(\text{ustate}) = \text{trying}).$$

By applying of the hypothesis that Fair_{run} holds, we obtain

$$\Diamond\Box(\neg\text{Enabled}_{\text{run}}(\mathbf{Now}) \wedge \mathbf{Now}(\text{ustate}) = \text{trying}).$$

Using the expression for $\text{Enabled}_{\text{run}}$ obtained above, we have

$$\Diamond\Box(\mathbf{Now}(\text{token}) = 0 \wedge \mathbf{Now}(\text{ustate}) = \text{trying}).$$

That is, it is persistently the case that the synchronizer component module possesses no tokens, and the user process is trying. Using the hypothesis that $\text{Fair}_{\text{request\_out}}$ holds, we obtain

$$\Box\Diamond(\mathbf{Occurs} = \text{request\_out}) \wedge \Diamond\Box(\mathbf{Now}(\text{token}) = 0).$$

That is, it is repeatedly the case that *request_out* occurs, but persistently the case that the synchronizer component module possesses no tokens. Applying the hypothesis that Rely^{SC} holds, we conclude

$$\Box\Diamond(\mathbf{Now}(\text{token}) \neq 0) \wedge \Diamond\Box(\mathbf{Now}(\text{token}) = 0).$$

That is, it is repeatedly the case that the synchronizer module possesses a token, but persistently the case that the synchronizer module possesses no tokens. This is a contradiction, and we conclude that case (A) is impossible.

Case (B): Suppose that (B) holds. Then by temporal reasoning, we have

$$\Diamond(\mathbf{Occurs} = \text{request\_in} \wedge \Box(\mathbf{Occurs} \neq \text{token\_out})).$$

That is, eventually there is a point at which a request for the token is received, but no token is ever sent in response. Using the definition of $\text{Prod}_{\text{token\_out}}$, and temporal reasoning, we obtain

$$\Diamond\Box\neg\text{Prod}_{\text{token\_out}}(\mathbf{Now}, \mathbf{Occurs}, \mathbf{After}).$$

Application of the hypothesis that $\text{Fair}_{\text{token\_out}}$ holds, we have

$$\Diamond\Box\neg\text{Enabled}_{\text{token\_out}}(\mathbf{Now}).$$

That is, it is persistently the case that a *token_out* event is not enabled. Using the expression for $\text{Enabled}_{\text{token\_out}}$ obtained above yields

(**) $\Diamond\Box(\mathbf{Now}(\text{ustate}) = \text{running} \vee \mathbf{Now}(\text{token}) = 0).$

Thus, it is persistently the case that either the user process is running or the synchronizer component module possesses no token. We now use the temporal tautology $\models \Diamond\Box(\varphi \vee \psi) \rightarrow (\Box\Diamond\varphi \vee \Diamond\Box\psi)$. Intuitively, this says that if it is persistently the case that $\varphi \vee \psi$ holds, then either φ holds repeatedly, or else ψ holds persistently. Application of this tautology to (**) gives

$$\Box\Diamond(\mathbf{Now}(\text{ustate}) = \text{running}) \vee \Diamond\Box(\mathbf{Now}(\text{token}) = 0).$$

That is, either the user process is repeatedly running, or the synchronizer component module persistently has no token. We now split the proof into two subcases, depending upon whether

(B1) $\Box\Diamond(\mathbf{Now}(\text{ustate}) = \text{running})$

or

(B2) $\Diamond\Box(\mathbf{Now}(\text{token}) = 0)$

holds.

Subcase (B1): Suppose that (B1) holds. Application of the hypothesis that Rely^{SC} holds gives

$$\Box\Diamond(\mathbf{Now}(\text{ustate}) = \text{running}) \wedge \Box\Diamond(\mathbf{Now}(\text{ustate}) \neq \text{running}).$$

Next, we use the temporal tautology

$$\models (\Box\Diamond\varphi(\mathbf{Now}) \wedge \Box\Diamond\neg\varphi(\mathbf{Now})) \rightarrow \Box\Diamond(\varphi(\mathbf{Now}) \wedge \neg\varphi(\mathbf{After})).$$

Intuitively, if it is repeatedly the case that φ holds of the current state, and it is repeatedly the case that $\neg\varphi$ holds of the current state, then it must repeatedly be the case that a point is reached where φ holds of the current state and $\neg\varphi$ holds of the "next" state. Application of this tautology in the present situation gives

$\Box\Diamond(\textbf{Now}(\text{ustate}) = \text{running} \wedge \textbf{After}(\text{ustate}) \neq \text{running})$.

In addition, we need the following invariance property:

$\text{Comp}^{SC}{}_k \models \Box(\textbf{Now}(\text{ustate}) = \text{running} \rightarrow \textbf{Now}(\text{token}) \neq 0)$.

The validity of this sentence can easily be shown by Corollary 3.7, and the details are omitted. Using this, plus the fact that

$$\models (\forall q,r \in \text{State}, e \in \text{Event})((\text{Trans}^{SC}(q, e, r) \wedge q(\text{ustate}) = \text{running} \wedge r(\text{ustate}) \neq \text{running}) \rightarrow r(\text{token}) = q(\text{token})),$$

which is verified by case analysis on e, we obtain

$\Box\Diamond(\textbf{After}(\text{ustate}) \neq \text{running} \wedge \textbf{After}(\text{token}) \neq 0)$.

Let us examine the intuitive content of the preceding steps. If the user process is running in the current state and not running in the "next" state, then the following must be true: Since the synchronizer component module must possess a token whenever the user process is running, and no event that takes the user process out of the running state can affect the number of tokens possessed, it must be the case that the synchronizer component module possesses a token in the next state as well.

Another use of the temporal tautology $\models \Box(\varphi(\textbf{After}) \rightarrow \Diamond\varphi(\textbf{Now}))$, we obtain

$\Box\Diamond(\textbf{Now}(\text{ustate}) \neq \text{running} \wedge \textbf{Now}(\text{token}) \neq 0)$,

which is a contradiction with formula (**). We conclude that subcase (B1) is impossible.

Subcase (B2): Suppose that (B2) holds, that is

$\Diamond\Box(\textbf{Now}(\text{token}) = 0)$.

Then by definition of $\text{Enabled}_{\text{request\_out}}$ we have

$\Diamond\Box\text{Enabled}_{\text{request\_out}}(\textbf{Now})$,

and thus by the hypothesis that $\text{Fair}_{\text{request\_out}}$ holds, we infer

$\Box\Diamond(\textbf{Occurs} = \text{request\_out})$.

That is, it is repeatedly the case that *request_out* events occur. By the hypothesis that Rely^{SC} holds, we conclude

$\Box\Diamond(\textbf{Now}(\text{token}) \neq 0)$

a contradiction with (B2). We conclude that subcase (B2) is impossible, and hence that case (B) is impossible.

Since both cases (A) and (B) have been shown to be impossible, we conclude that the original hypotheses are contradictory, and thus the synchronizer component module specification is I/O-consistent.

5.5 Composition of I/O-Behaviors

We have previously shown that the class of I/O-behaviors is closed under the abstraction operators associated with the I/O-abstraction maps. In this section, we define the class of "I/O-decomposition maps," and show that the class of I/O-behaviors is also closed under the composition operators associated with these maps.

5.5.1 I/O-Decomposition Maps

When we defined the notion of a system interface above, we noted that there is a canonical decomposition map (and hence a composition operator) associated with each system interface. We would now like to extend the notion of composition associated with system interfaces so that we can view behaviors of non-system interfaces as a composition of component behaviors. The most natural way to do this is to require that the the domain of a decomposition map be a system interface only up to isomorphism.

Definition - An *isomorphism* from the I/O-interface E to the I/O-interface D is a bijective translation $\gamma: E \rightarrow D$ such that γ and γ^{-1} are embeddings. ▮

Definition - An *I/O-decomposition map* from the I/O-interface E to the collection of I/O-interfaces $\langle F_i \rangle_{i \in I}$ is a vector $\langle \delta_i \rangle_{i \in I}$ of translations, where $\delta_i: E \rightarrow F_i$, with the following property: There exists a system interface $E' \subseteq \otimes_{i \in I} F_i$ and an isomorphism $\gamma: E \rightarrow E'$, such that $\delta_i = \delta_i' \circ \gamma$ for all $i \in I$, where $\langle \delta_i' \rangle_{i \in I}$ is the canonical decomposition map associated with E'. ▮

From this definition, we can immediately derive a number of properties of the I/O-decomposition maps.

Lemma 5.9 - If $\langle \delta_i \rangle_{i \in I}$ is an I/O-decomposition map from E to $\langle F_i \rangle_{i \in I}$, then

(1) $e \neq e'$ implies $\delta_i(e) \neq \delta_i(e')$ for some $i \in I$. ($\underline{\delta}$ is injective)

(2) $\delta_i(\mathrm{In}_E) \subseteq \mathrm{In}_{F_i} \cup \{\lambda_{F_i}\}$ for all $i \in I$. ($\underline{\delta}$ preserves inputs)

(3) If $e \in \mathrm{Out}_E$ then $\delta_i(e) \in \mathrm{Out}_{F_i}$ for some $i \in I$. ($\underline{\delta}$ strictly preserves outputs)

(4) $\delta_i^{-1}(\mathrm{Out}_{F_i}) \cap \delta_j^{-1}(\mathrm{Out}_{F_j}) = \emptyset$ whenever $i \neq j$. (*Compatible Coupling Property*)

(5) δ_i is onto F_i for all $i \in I$.

Proof - Straightforward. ∎

5.5.2 Closure Proof

We can now prove that the class of I/O-behaviors is closed under the composition operators associated with I/O-decomposition maps.

Theorem 5.10 - Suppose $\langle\delta_i\rangle_{i\in I}$ is an I/O-decomposition map, where $\delta_i: E \to F_i$. If B_i is an I/O-behavior of interface F_i for each $i \in I$, then $\underline{\delta}^{-1}(\underline{B})$ is an I/O-behavior of interface E.

Proof - Suppose that for each $i \in I$, $\langle M_i, \langle \mathrm{Prod}_{i,a}\rangle_{a\in A_i}\rangle$ is a PS-machine of interface F_i, such that $\mathrm{Beh}(M_i, \langle \mathrm{Prod}_{i,a}\rangle_{a\in A_i}) = B_i$. We construct a PS-machine $\langle M, \langle \mathrm{Prod}_{i,a}{}'\rangle_{i\in I, a\in A_i}\rangle$ of interface E such that $\mathrm{Beh}(M, \langle \mathrm{Prod}_{i,a}{}'\rangle_{i\in I, a\in A_i}) = \underline{\delta}^{-1}(\underline{B})$.

Let M be defined as follows:

$$Q_M = \Pi_{i\in I} Q_{M_i}$$
$$\mathrm{Init}_M = \Pi_{i\in I} \mathrm{Init}_{M_i}$$
$$\mathrm{Trans}_M = \{\langle \underline{q}, e, \underline{r}\rangle\text{: such that } \langle q_i, \delta_i(e), r_i\rangle \in \mathrm{Trans}_{M_i} \text{ for all } i \in I\}.$$

It is easy to check that Init_M is nonempty and that $\langle \underline{q}, \lambda, \underline{q}\rangle \in \mathrm{Trans}_M$ for all $\underline{q} \in Q_M$. Thus M is a machine. To show that M is an I/O-machine, we must show that it is input-cooperative. Suppose $\underline{q} \in Q_M$ and $e \in \mathrm{In}_E$. Since $\underline{\delta}$ preserves input, it follows that $\delta_i(e) \in \mathrm{In}_{F_i} \cup \{\lambda_{F_i}\}$ for each $i \in I$. Since each M_i is input-cooperative, for each $i \in I$ we can get r_i such that $\langle q_i, \delta_i(e), r_i\rangle \in \mathrm{Trans}_{M_i}$. It follows that $\langle \underline{q}, e, \underline{r}\rangle \in \mathrm{Trans}_M$.

For each $i \in I$, and $a \in A_i$, define

$$\mathrm{Prod}_{i,a}{}' = \{\langle \underline{q}, e, \underline{r}\rangle \in \mathrm{Trans}_M\text{: } \langle q_i, \delta_i(e), r_i\rangle \in \mathrm{Prod}_{i,a} \text{ and } e \notin \mathrm{In}_E\}.$$

It is clear that each $\mathrm{Prod}_{i,a}{}'$ is a productive step set for M. To show that $\langle M, \langle \mathrm{Prod}_{i,a}{}'\rangle_{i\in I, a\in A_i}\rangle$ is a PS-machine, we must show that the sets $\mathrm{Prod}_{i,a}{}'$ cover the set of nonnull output or λ-steps in Trans_M.. Suppose $\langle \underline{q}, e, \underline{r}\rangle$ is such a step. Then $\langle q_i, \delta_i(e), r_i\rangle$ is a nonnull output or λ-step for M_i for some $i \in I$, by the fact that $\underline{\delta}$ strictly preserves outputs. Since the collection $\langle \mathrm{Prod}_{i,a}\rangle_{a\in A_i}$ covers the nonnull output or

λ-steps for M_i, we know that $\langle q_i, \delta_i(e), r_i\rangle \in \text{Prod}_{i,a}$ for some $a \in A_i$. Hence $\langle \underline{q}, e, \underline{r}\rangle \in \text{Prod}_{i,a}'$.

We claim that $\text{Beh}(M, \langle \text{Prod}_{i,a}'\rangle_{i\in I, a\in A_i}) = \underline{\delta}^{-1}(\langle \text{Beh}(M_i, \langle \text{Prod}_{i,a}\rangle_{a\in A_i})\rangle_{i\in I})$.

Case $\text{Beh}(M, \langle \text{Prod}_{i,a}'\rangle_{i\in I, a\in A_i}) \subseteq \underline{\delta}^{-1}(\underline{B})$:

Each computation X of M maps in an obvious way (by taking the image of the observation part under δ_i, and the canonical projection of the state part) to a computation X_i of M_i, for each $i \in I$. Suppose that X is fair. Let $i \in I$ and $a \in A_i$ be fixed. We show that if $\text{Prod}_{i,a}$ is repeatedly enabled in X_i, then $\text{Prod}_{i,a}$ repeatedly runs in X_i. Suppose $\text{Prod}_{i,a}$ is repeatedly enabled in X_i.

We first show that, given $\underline{q} \in \text{State}_M$, if $\text{Prod}_{i,a}$ is enabled in state q_i, then $\text{Prod}_{i,a}'$ is enabled in state $\underline{q}$. If $\text{Prod}_{i,a}$ is enabled in state q_i, then there exists $f_i \in \text{Out}_{F_i} \cup \{\lambda_{F_i}\}$ and $r_i \in Q_{M_i}$ such that $\langle q_i, e_i, r_i\rangle \in \text{Prod}_{i,a}$. Since δ_i is onto Out_{F_i}, we we can get $e \in \text{Out}_E \cup \{\lambda_E\}$ with $\delta_i(e) = f_i$. By the compatible coupling property of $\underline{\delta}$, we know that $\delta_j(e) \in \text{In}_{F_j} \cup \{\lambda_{F_j}\}$ for all $j \in I - \{i\}$. For each $j \in I - \{i\}$, by the input-cooperative property of M_j, we can get r_j' such that $\langle q_j, \delta_j(e), r_j'\rangle \in \text{Trans}_{M_j}$. It follows that $\langle \underline{q}, e, \underline{r}'\rangle \in \text{Prod}_{i,a}'$, and thus $\text{Prod}_{i,a}'$ is enabled in state $\underline{q}$.

Since $\text{Prod}_{i,a}$ is repeatedly enabled in X_i by hypothesis, and $\text{Prod}_{i,a}$ enabled in state q_i implies $\text{Prod}_{i,a}'$ enabled in state $\underline{q}$, we know that $\text{Prod}_{i,a}'$ is repeatedly enabled in X. By the fairness of X, it follows that $\text{Prod}_{i,a}'$ repeatedly runs in X. By definition of $\text{Prod}_{i,a}'$, if $\text{Step}_X(t) \in \text{Prod}_{i,a}'$, then $\text{Step}_{X_i}(t) \in \text{Prod}_{i,a}$. This implies that $\text{Prod}_{i,a}$ repeatedly runs in X_i.

Case $\underline{\delta}^{-1}(\underline{B}) \subseteq \text{Beh}(M, \langle \text{Prod}_{i,a}'\rangle_{i\in I, a\in A_i})$:

Suppose that $\delta_i(x) \in \text{Beh}(M_i, \langle \text{Prod}_{i,a}\rangle_{a\in A_i})$, for all $i \in I$. For each $i \in I$, let X_i be a fair computation of M_i in which the observation $\delta_i(x)$ is generated. We construct a fair computation X for M, such that $\text{Obs}_X = x$.

Without loss of generality we assume that the X_i have the following property: For all $t \in [0, \infty)$, if a productive λ-step runs at time t in X_i for some $i \in I$, then the step that runs at time t in X_j is null for all $j \in I - \{i\}$. If we are given a collection $\langle X_i'\rangle_{i\in I}$ for which this property does not hold, then it is a simple matter to construct order-isomorphisms f_i:

$[0, \infty) \rightarrow [0, \infty)$ such that if $X_i = X_i' \circ f_i$, then $\text{Obs}_{X_i} = \text{Obs}_{X_i'}$ for all i and the desired property holds for the collection $\langle X_i \rangle_{i \in I}$. Since the property of being a fair computation is preserved under stretching of $[0, \infty)$ by an order-isomorphism, it follows that each X_i is a fair computation for M_i.

We now define X by letting $\text{Obs}_X = x$ and $\text{State}_X(t) = \langle \text{State}_{X_i}(t) \rangle_{i \in I}$. It is easy to see that X is a computation for M.

To show that X is fair, suppose that $\text{Prod}_{i,a}'$ is repeatedly enabled in X. Since $\text{Prod}_{i,a}'$ enabled in state $\underline{q}$ implies $\text{Prod}_{i,a}$ enabled in state q_i, it follows that $\text{Prod}_{i,a}$ is repeatedly enabled in X_i. Since X_i is fair, we know that $\text{Prod}_{i,a}$ repeatedly runs in X_i. We claim that if $\text{Step}_{X_i}(t) \in \text{Prod}_{i,a}$ then $\text{Step}_X(t) \in \text{Prod}_{i,a}'$ as well, and hence $\text{Prod}_{i,a}'$ repeatedly runs in X.

By definition of $\text{Prod}_{i,a}'$, if $\text{Step}_{X_i}(t) \in \text{Prod}_{i,a}$ then $\text{Step}_X(t) \in \text{Prod}_{i,a}'$, except in case $\text{Obs}_X(t) \in \text{In}_E$. But if $\text{Obs}_X(t) \in \text{In}(E)$, then the fact that δ_i preserves inputs and $\text{Prod}_{i,a}$ contains only output and λ-steps implies that $\text{Step}_{X_i}(t) = \langle q_i, \lambda, r_i \rangle \in \text{Prod}_{i,a}$. Since $\underline{\delta}$ is injective and preserves inputs, it must be the case that $\text{Obs}_{X_j}(t) \in \text{In}_{F_j}$ for some $j \in I - \{i\}$, and hence $\text{Step}_{X_j}(t)$ is nonnull. This contradicts our assumption that if a productive λ-step runs at time t in X_i, then the step occurring at time j in X_j is null for all $j \in I - \{i\}$. ∎

5.6 Alternative Classes of Computable Behaviors

The class of I/O-behaviors is by no means the only class of "computable" behaviors that it is interesting to consider. By replacing the fairness requirement for computations of I/O-systems with that of "weak fairness," in which a process is required to repeatedly run only if it is *persistently enabled*, rather than repeatedly enabled, we obtain the class of *weak I/O-behaviors* (WI/O-behaviors). It can be shown that every WI/O-behavior is an I/O-behavior, but not every I/O-behavior is a WI/O-behavior. The notion of WI/O-consistency is therefore strictly more stringent than I/O-consistency.

Besides the fairness assumption, the definition of the class of I/O-behaviors embodies several other choices that might have been made differently:

(1) (*Asynchrony*) - The I/O-systems model is an asynchronous model of computation. We might have chosen a timing-dependent model of computation instead.

(2) (*Input/Output Structure*) - Instead of focusing on interfaces with input/output structure, we might have chosen additional or alternative structure, such as interfaces in which events include information about the physical location at which they occur.

(3) (*Simultaneity*) - The definition of an I/O-system permits at most one process to perform an output at any instant of time. We might imagine a more general model in which any number of processes can perform an output at once.

An interesting avenue for future research is to try to discover additional classes of behaviors and associated notions of consistency by modifying one or more of the above assumptions.

6. A Completeness Result

A reasonable question to ask about the sufficient correctness conditions required by the Correctness Theorem is whether these conditions are also necessary. That is, is it the case that the maximality and validity conditions hold for every correct implementation involving state-transition specifications? In this chapter we show that in general the maximality and validity conditions need not hold for every correct implementation. However, it is possible to impose some well-formedness conditions on the state-transition specifications involved in the implementation, which are sufficient to ensure that correctness implies maximality and validity. The Completeness Theorem (Theorem 6.4) is the formal statement of this result. Although Theorem 6.4 is probably not the strongest result of this kind it is possible to prove, it nevertheless sheds some light on the limitations of the Correctness Theorem, and serves to motivate some well-formedness properties of state-transition specifications.

6.1 Specification Domains

The statement and proof of Theorem 6.4 depends crucially on the existence of a collection of interfaces, behaviors, abstraction maps, and decomposition maps with closure properties like those of the I/O-interfaces, I/O-behaviors, I/O-abstraction maps, and I/O-decomposition maps defined in Chapter 5. The definition of a "specification domain" below summarizes these properties, which seem like fundamental properties that are likely to be shared by other interesting models.

Informally, a specification domain $\mathfrak{D}$ contains four pieces of data: the "$\mathfrak{D}$-interfaces," the "$\mathfrak{D}$-behaviors," the "$\mathfrak{D}$-abstraction maps," and the "$\mathfrak{D}$-decomposition maps." The $\mathfrak{D}$-interfaces are interfaces with structure particular to the domain $\mathfrak{D}$. For example, the I/O-interfaces are those whose non-λ events are partitioned into input and output events. For each $\mathfrak{D}$-interface E, the $\mathfrak{D}$-behaviors of interface E represent a class of "realizable" or "computable" behaviors of interface E. Just as the definition of I/O-behavior depends upon the input/output structure of an I/O-interface, whether or not a behavior of $\mathfrak{D}$-interface E is a $\mathfrak{D}$-behavior of interface E will depend, in general, on the particular structure of the interface E. The $\mathfrak{D}$-abstraction and $\mathfrak{D}$-decomposition maps represent meaningful ways to abstract and decompose systems modeled by $\mathfrak{D}$-behaviors. In general, these maps will have certain preservation properties with respect to the particular structure of the interfaces, just as the I/O-abstraction and I/O-decomposition maps preserve input/output structure in various ways.

The definition of a specification domain requires that the class of $\mathfrak{S}$-behaviors be closed under the abstraction and composition operators associated with the $\mathfrak{S}$-abstraction and $\mathfrak{S}$-decomposition maps. In addition to the properties of closure under abstraction and composition discussed above, we require a third regularity property of the $\mathfrak{S}$-behaviors. This property, called "nondegeneracy," rules out the empty behavior as a $\mathfrak{S}$-behavior of any interface. Intutively, the empty behavior does not model any real system, since it is always possible to obtain an observation of a real system, even if that observation is only the null observation Λ.

Definition - A *specification domain* $\mathfrak{S}$ consists of the following:

- A class $\mathrm{Interfaces}_{\mathfrak{S}}$ of interfaces, called the *$\mathfrak{S}$-interfaces.*

- For each pair E, $D \in \mathrm{Interfaces}_{\mathfrak{S}}$, a set $\mathrm{AbsMaps}_{\mathfrak{S}}(E, D)$ of translations from E to D, called the set of *$\mathfrak{S}$-abstraction maps from E to D.*

- For each pair E, $\langle F_i \rangle_{i \in I}$, where I is a finite index set and E and each F_i are elements of $\mathrm{Interfaces}_{\mathfrak{S}}$, a set $\mathrm{DecMaps}_{\mathfrak{S}}(E, \underline{F})$ called the set of *$\mathfrak{S}$-decomposition maps from E to $\underline{F}$*. Each element of $\mathrm{DecMaps}_{\mathfrak{S}}(E, \underline{F})$ is a vector $\langle \delta_i \rangle_{i \in I}$, where δ_i is a translation from E to F_i.

- For each interface $E \in \mathrm{Interfaces}_{\mathfrak{S}}$, a set $\mathrm{Behaviors}_{\mathfrak{S}}(E)$ of behaviors of interface E, called the set of *$\mathfrak{S}$-behaviors of interface E.*

In addition, $\mathfrak{S}$ is required to have the following properties:

(1) (Nondegeneracy) - For all $\mathfrak{S}$-interfaces E, the empty behavior $\emptyset$ is not in $\mathrm{Behaviors}_{\mathfrak{S}}(E)$.

(2) (Abstraction Closure) - For all $\mathfrak{S}$-interfaces E, D, if $\alpha \in \mathrm{AbsMaps}_{\mathfrak{S}}(E, D)$ and $B \in \mathrm{Behaviors}_{\mathfrak{S}}(E)$, then $\alpha(B) \in \mathrm{Behaviors}_{\mathfrak{S}}(D)$.

(3) (Composition Closure) - For all $\mathfrak{S}$-interfaces E, $\langle F_i \rangle_{i \in I}$, if $\underline{\delta} \in \mathrm{DecMaps}_{\mathfrak{S}}(E, \underline{F})$ and $\underline{B} = \langle B_i \rangle_{i \in I}$ is such that $B_i \in \mathrm{Behaviors}_{\mathfrak{S}}(F_i)$ for each $i \in I$, then $\underline{\delta}^{-1}(\underline{B}) \in \mathrm{Behaviors}_{\mathfrak{S}}(E)$. ∎

A rather simple example of a specification domain is the domain "CSP," where we define every interface to be a CSP-interface, every translation α to be a CSP-abstraction map, every finite vector $\underline{\delta}$ of translations with a common domain to be a CSP-decomposition map, and define the CSP-behaviors of interface E to be exactly those behaviors of interface E that are *nonempty*, *asynchronous*, and *truncation-closed*.[1] We call this specification domain CSP because it is closely related to the "trace model" for CSP defined in [Hoare81b]. In that paper, process behaviors are modeled by nonempty, prefix-closed subsets of E^*, where E is an alphabet of process events. To each nonempty, prefix-closed subset of E^*, there naturally corresponds a nonempty, asynchronous, and truncation-closed behavior of interface E. Thus, for each of Hoare's processes, there is a CSP-behavior that contains the same information. Hoare defines operations of *parallel composition*, *concealment*, and *alphabet transformation* on processes. Under the natural correspondence described above, Hoare's concealment and alphabet transformation operations are special cases of the CSP-abstraction operators defined here, and Hoare's parallel composition operation is a special case of the CSP-composition operators defined here. Since no truncation-closed behavior can satisfy a specification with nontrivial eventuality properties, the specification domain CSP is not particularly useful for the analysis of such specifications.

As a consequence of Theorem 5.1, Corollary 5.5, and Theorem 5.10, the I/O-interfaces, I/O-behaviors, I/O-abstraction maps, and I/O-decomposition maps also define a specification domain, which we call "I/O."

We can generalize the definition of I/O-consistency to an arbitrary specification domain $\mathfrak{I}$.

Definition - A specification S of $\mathfrak{I}$-interface E is *$\mathfrak{I}$-consistent* if $\mathfrak{B}(S) \cap \text{Behaviors}_{\mathfrak{I}}(E) \neq \emptyset$. ∎

We define relativized notions of interconnection, implementation, and correctness with respect to a specification domain $\mathfrak{I}$ as follows: An interconnection $\mathfrak{J}$ is a *$\mathfrak{I}$-interconnection* if the interfaces $D^{\mathfrak{J}}$, $E^{\mathfrak{J}}$, and $F_i^{\mathfrak{J}}$ for each $i \in I$ are $\mathfrak{I}$-interfaces, the abstraction map $\alpha^{\mathfrak{J}}$ is a $\mathfrak{I}$-abstraction map from $E^{\mathfrak{J}}$ to $D^{\mathfrak{J}}$, and the decomposition map $\underline{\delta}^{\mathfrak{J}}$ is a $\mathfrak{I}$-decomposition map from $E^{\mathfrak{J}}$ to $\underline{F}^{\mathfrak{J}}$. An implementation $\langle \mathfrak{J}, S_{\text{abs}}, \underline{S} \rangle$ is a *$\mathfrak{I}$-implementation* if $\mathfrak{J}$ is a $\mathfrak{I}$-interconnection. We say that the $\mathfrak{I}$-implementation

1. If x is an observation and $t \in [0, \infty)$, then the *t-truncation* of x is the observation x' such that $x'(t') = x(t')$ for all $t' \in [0, t)$, and $x'(t') = \lambda$ for all $t' \in [t, \infty)$. A behavior B is *truncation-closed* if whenever $x \in B$ and $t \in [0, \infty)$, then the t-truncation of x is also in B.

$\langle \mathcal{J}, S_{\text{abs}}, \underline{S} \rangle$ is $\mathcal{D}$-*correct* if $\alpha^{\mathcal{J}} \circ (\underline{\delta}^{\mathcal{J}})^{-1}(\underline{B}) \in \mathfrak{B}(S_{\text{abs}})$ $\cap B_i \in \mathfrak{B}(S_i) \cap \text{Behaviors}_{\mathcal{D}}(F_i^{\mathcal{J}})$ for each $i \in I$.

Every $\mathcal{D}$-implementation that is correct in the sense of Chapter 2 is also $\mathcal{D}$-correct, and thus the Correctness Theorem can be used to prove $\mathcal{D}$-correctness. However, in general there will be $\mathcal{D}$-correct implementations that are not correct in the sense of Chapter 2.

Lemma 6.1 - If a $\mathcal{D}$-implementation is correct, then it is $\mathcal{D}$-correct.

Proof - Suppose $\langle \mathcal{J}, S_{\text{abs}}, \langle S_i \rangle_{i \in I} \rangle$ is a correct $\mathcal{D}$-implementation. For each $i \in I$, let B_i be an arbitrary $\mathcal{D}$-behavior of interface $F_i^{\mathcal{J}}$ that satisfies S_i. Let $B_{\text{abs}} = \alpha^{\mathcal{J}} \circ (\underline{\delta}^{\mathcal{J}})^{-1}(\underline{B})$. Then since $\mathcal{D}$ is closed under abstraction and composition, it follows that B_{abs} is a $\mathcal{D}$-behavior of interface $D^{\mathcal{J}}$. By the assumption of correctness, B_{abs} satisfies S_{abs}. Since the B_i were arbitrary, it follows that $\langle \mathcal{J}, S_{\text{abs}}, \underline{S} \rangle$ is $\mathcal{D}$-correct. ∎

We next define the notion of an "evolutionary" specification domain. Intuitively, if an evolutionary specification domain $\mathcal{D}$ contains a behavior B that models what a system S can do starting from time 0, and if we observe S produce a certain prefix of an observation over the interval $[0, t)$, then $\mathcal{D}$ will also contain a "future" behavior B', which models what S is capable of doing, starting from time t. Probably any reasonable specification domain will be evolutionary (as is the specification domain I/O) although this property does not seem quite fundamental enough to be included as part of the definition of a specification domain.

To define the evolutionary property precisely, we require some additional notation. If x and y are observations and $a \in [0, \infty)$, then we write $x =_a y$ if $x(t) = y(t)$ for all $t \in [0, a)$. If B is a behavior of interface E, $x \in \text{Obs}(E)$ is an observation, and $t \in [0, \infty)$, then define the *future* of B with respect to x and t as follows:

$$\text{future}_{x,t}(B) = \{\text{suffix}_t(y) : y \in B, y =_t x\}.$$

Intuitively, if a behavior B models what a system can do if we begin watching at time $t = 0$, then $\text{future}_{x,t}(B)$ models what the system can do after we have already observed the initial segment of x on the interval $[0, t)$.

Definition - A specification domain $\mathfrak{D}$ is *evolutionary* if, whenever B is a $\mathfrak{D}$-behavior of $\mathfrak{D}$-interface E, $x \in B$, and $t \in [0, \infty)$, then $\mathrm{future}_{x,t}(B)$ is also a $\mathfrak{D}$-behavior of $\mathfrak{D}$-interface E. ∎

For the remainder of this chapter, we assume that an evolutionary specification domain $\mathfrak{D}$ (such as the domain CSP or I/O) has been fixed.

6.2 Locally $\mathfrak{D}$-Consistent Subset Specifications

This section introduces the notion of a "locally $\mathfrak{D}$-consistent" subset specification, and obtains some properties of such specifications that will be used in the proof of Theorem 6.4. Intuitively, local $\mathfrak{D}$-consistency of a subset specification S means that $\mathcal{O}(S)$ contains no isolated observations that cannot be realized in some $\mathfrak{D}$-behavior satisfying S.

Definition - A subset specification S of $\mathfrak{D}$-interface E is *locally $\mathfrak{D}$-consistent* if for all $x \in \mathcal{O}(S)$ there exists a $\mathfrak{D}$-behavior B of interface E such that $x \in B \subseteq \mathcal{O}(S)$. ∎

Note that if S is locally $\mathfrak{D}$-consistent, and in addition $\mathcal{O}(S) \neq \emptyset$, then S is $\mathfrak{D}$-consistent.

Lemma 6.2 below states that if the component module specifications in a $\mathfrak{D}$-implementation are locally $\mathfrak{D}$-consistent, then the necessary and sufficient conditions for correctness provided by Lemma 3.1 for implementations involving subset specifications, are also necessary and sufficient for $\mathfrak{D}$-correctness.

Lemma 6.2 - Suppose $\langle \mathcal{I}, S_{\mathrm{abs}}, \langle S_i \rangle_{i \in I} \rangle$ is a $\mathfrak{D}$-implementation, where S_{abs} and each S_i are subset specifications. Suppose that each S_i is locally $\mathfrak{D}$-consistent. Then $\langle \mathcal{I}, S_{\mathrm{abs}}, \langle S_i \rangle_{i \in I} \rangle$ is $\mathfrak{D}$-correct iff $\alpha^{\mathcal{I}} \circ (\underline{\delta}^{\mathcal{I}})^{-1}(\langle \mathcal{O}(S_i) \rangle_{i \in I}) \subseteq \mathcal{O}(S_{\mathrm{abs}})$.

Proof - $=>$ follows directly from Lemma 3.1 and Lemma 6.1, and actually does not require the assumption of local $\mathfrak{D}$-consistency. To show $<=$, suppose $\langle \mathcal{I}, S_{\mathrm{abs}}, \langle S_i \rangle_{i \in I} \rangle$ is $\mathfrak{D}$-correct. It suffices to show that if $x \in \mathrm{Obs}(E)$ is such that $\delta_i^{\mathcal{I}}(x) \in \mathcal{O}(S_i)$ for each $i \in I$, then $\alpha^{\mathcal{I}}(x) \in \mathcal{O}(S_{\mathrm{abs}})$. Because each $\mathcal{O}(S_i)$ is assumed locally $\mathfrak{D}$-consistent, given $x \in \mathrm{Obs}(E)$ such that $\delta_i^{\mathcal{I}}(x) \in \mathcal{O}(S_i)$ for each $i \in I$, then for each $i \in I$ there exists a $\mathfrak{D}$-behavior B_i of interface $F_i^{\mathcal{I}}$ such that $\delta_i^{\mathcal{I}}(x) \in B_i \subseteq \mathcal{O}(S_i)$. Thus $\alpha^{\mathcal{I}}(x) \in B_{\mathrm{abs}} = \alpha^{\mathcal{I}} \circ (\underline{\delta}^{\mathcal{I}})^{-1}(\underline{B})$, which is a $\mathfrak{D}$-behavior because $\mathfrak{D}$ is closed under abstraction and composition. By the assumption of $\mathfrak{D}$-correctness, it follows that $B_{\mathrm{abs}} \subseteq \mathcal{O}(S_{\mathrm{abs}})$, and hence $\alpha^{\mathcal{I}}(x) \in \mathcal{O}(S_{\mathrm{abs}})$. ∎

The proof of Theorem 6.4 requires Lemma 6.3 below, which expresses a special property of locally $\mathfrak{I}$-consistent subset specifications in an evolutionary specification domain $\mathfrak{I}$.

Lemma 6.3 - Suppose that $\mathfrak{I}$ is an evolutionary specification domain, and that S is a locally $\mathfrak{I}$-consistent subset specification of $\mathfrak{I}$-interface E. Then $\text{future}_{x,t}(\mathcal{O}(S))$ contains a $\mathfrak{I}$-behavior of $\mathfrak{I}$-interface E whenever $x \in \mathcal{O}(S)$ and $t \in [0, \infty)$.

Proof - The local $\mathfrak{I}$-consistency of S means that, given $x \in \mathcal{O}(S)$ there exists a $\mathfrak{I}$-behavior B of interface E such that $x \in B \subseteq \mathcal{O}(S)$. Since $\mathfrak{I}$ is evolutionary, it follows that $\text{future}_{x,t}(B)$ is a $\mathfrak{I}$-behavior contained in $\text{future}_{x,t}(\mathcal{O}(S))$. ▮

6.3 Well-Formedness Properties of Specifications

This section defines three properties of state-transition specifications, which are used in the statement of Theorem 6.4. These properties are: *regularity*, *quasi-determinacy*, and *orthogonality*. The original motivation for these definitions was technical, in the sense that they were sufficient to permit the proof of Theorem 6.4 to go through. However, it was surprising to find that these properties could be thought of as well-formedness properties that should be satisfied by "good" state-transition specifications. In a regular state-transition specification, whether or not a computation is valid depends only upon the observation that is produced, and not upon the particular choice of states. In a quasi-determinate specification, the fact that the state-transition relation permits choices between states is inessential, since a choice of state made at time t can have no effect on the portion of the observation produced subsequent to time t. Orthogonality is related to the correct partitioning of "local" and "global" properties between the state-transition relation and the validity conditions of a specification.

We first consider regularity. Intuitively, the requirement of regularity amounts to the assumption that whether a computation is valid does not depend upon the states appearing in the computation, but rather only the observation produced.

Definition - A state-transition specification $S = \langle M, V \rangle$ is *regular* if, for all computations X and Y of M, if $\text{Obs}_X = \text{Obs}_Y$, then $X \in V$ iff $Y \in V$. ▮

To motivate the somewhat technical definition of quasi-determinacy, it is convenient to first examine the stronger, but more simply defined notion of "determinacy."

Definition - A machine M is *determinate* if Init_M is a singleton set, and for all $q \in Q_M$ and all $e \in E_M$, there is at most one $r \in Q_M$ such that $\langle q, e, r\rangle \in \mathrm{Trans}_M$. A state-transition specification $S = \langle M, V\rangle$ is *determinate* if M is determinate. ▮

A determinate specification is automatically regular, since a determinate specification can have at most one computation that produces a given observation. The importance of the determinacy property is that each observation generated by a determinate machine is produced in exactly one computation of that machine. Thus, if $S = \langle M, V\rangle$ is a determinate specification, $x \in \mathcal{O}(S)$, and X is a computation of M with $\mathrm{Obs}_X = x$, then it is automatically the case that $X \in V$, since no other computation of M can produce the observation x.

To show that the maximality condition holds for a correct implementation, it appears to be necessary to assume that some property similar to determinacy holds for the abstract module specification. To see why, consider the following example: We are attempting to implement an abstract module whose function is to produce a finite number of occurrences of a single event e. (Think of a "black box" with a single light bulb on top, and let e be an event corresponding to a flash of the light bulb.) This module can be specified in two different ways:

(*Determinate*): The state set of the specification consists of a single state $*$. The event e is enabled in state $*$, and obviously cannot produce any state change. The constraint that e should appear only finitely many times is captured by the validity condition.

(*Indeterminate*): The state set for the specification is the set of natural numbers. Every state is an initial state. The event e is enabled in state k iff $k \neq 0$, and the occurrence of e causes the state to be decremented. Every computation is valid. In this specification, the requirement that e occurs only finitely often is captured by the indeterminate choice of initial state.

Let S_{det} be the determinate specification and let S_{ind} be the indeterminate specification. Clearly $\mathcal{O}(S_{\mathrm{det}}) = \mathcal{O}(S_{\mathrm{ind}})$.

Now consider an interconnection $\mathfrak{I} = \langle\alpha^{\mathfrak{I}}, \delta^{\mathfrak{I}}\rangle$, where the abstract interface $D^{\mathfrak{I}}$, the single component interface $F^{\mathfrak{I}}$, and the composite interface $E^{\mathfrak{I}}$ are all the same interface $\{\lambda, e\}$, and the abstraction map $\alpha^{\mathfrak{I}}$ and the decomposition map $\delta^{\mathfrak{I}}$ are the identity translation. Clearly both of the implementations $\langle\mathfrak{I}, S_{\text{det}}, S_{\text{ind}}\rangle$ and $\langle\mathfrak{I}, S_{\text{ind}}, S_{\text{det}}\rangle$ are correct. However, the maximality condition does not hold for the implementation $\langle\mathfrak{I}, S_{\text{ind}}, S_{\text{det}}\rangle$. To see this, note that any pair $\langle k, *\rangle$ is an initial state for the composite machine, and is hence reachable for that machine. Furthermore, the event e is always enabled for the component machine. For maximality to hold, it would have to be the case that e is enabled for the abstract machine no matter what the value of k is. But e is not enabled for the abstract machine if $k = 0$.

In certain situations, for example the transmission line module specification in Appendix II, the use of indeterminate specifications is quite natural. However, the preceding example shows that unless we are careful, it may not be possible to use the Correctness Theorem to prove the correctness of implementations when such a specification is used as the specification for the abstract module.

The proof of the Completeness Theorem actually does not require that the abstract module specification S_{abs} be determinate, but rather the somewhat weaker assumption that S_{abs} be regular and "quasi-determinate." Intuitively, the set of future observations that can be produced by a quasi-determinate machine is independent of the choice of states made on the initial segment $[0, t]$.

To define quasi-determinacy precisely, we extend to histories the $=_a$ notation defined above for observations. If X and Y are histories, then we write $X =_a Y$ if $\text{Obs}_X(t) =_a \text{Obs}_Y(t)$ and $\text{State}_X(t) = \text{State}_Y(t)$ for all $t \in [0, a)$ (and hence for all $t \in [0, a]$ by the properties of state functions).

Definition - A machine M is *quasi-determinate* if for all computations X and Y for M, and all $t \in [0, \infty)$, if $\text{Obs}_X =_t \text{Obs}_Y$, then there exists a computation Z of M such that $Z =_t X$ and $\text{Obs}_Z = \text{Obs}_Y$. A state-transition specification $S = \langle M, V\rangle$ is *quasi-determinate* if M is quasi-determinate. ∎

If a state-transition specification is determinate, then it can also be seen to be quasi-determinate by choosing $Z = Y$ in the above definition. Determinacy implies that State_Z can be defined in exactly one way on the interval $[0, t]$, thus showing that $X =_t Z$.

We next consider orthogonality. Intuitively, in an orthogonal specification, every computation agrees for an arbitrarily long time with a valid computation. Orthogonality is related to the correct partitioning of "local" and "global" properties between the state-transition relation and the validity conditions of a specification. Roughly, orthogonality means that the validity conditions contain no information that could have been expressed by strengthening the machine part of the specification.

Definition - A state-transition specification $S = \langle M, V \rangle$ is *orthogonal* if for all computations X of M and all $t \in [0, \infty)$, there exists $Y \in V$ such that $X =_t Y$.

6.4 The Completeness Theorem

We can now state and prove the Completeness Theorem.

Theorem 6.4 (Completeness Theorem) - Let $\mathfrak{I}$ be an evolutionary specification domain. Suppose that $\langle \mathfrak{I}, S_{abs}, \langle S_i \rangle_{i \in I} \rangle$ is a $\mathfrak{I}$-implementation, where S_{abs} and S_i for each $i \in I$ are state-transition specifications. Suppose that S_{abs} is regular and quasi-determinate, and that S_i is orthogonal and locally $\mathfrak{I}$-consistent for each $i \in I$. If $\langle \mathfrak{I}, S_{abs}, \langle S_i \rangle_{i \in I} \rangle$ is $\mathfrak{I}$-correct then the maximality and validity conditions hold.

Proof - Suppose that $S_{abs} = \langle M_{abs}, V_{abs} \rangle$, and that $S_i = \langle M_i, V_i \rangle$, for each $i \in I$. Let M be the composite machine. Note that the assumption that each S_i is locally $\mathfrak{I}$-consistent together with the assumption of $\mathfrak{I}$-correctness implies, by Lemma 6.2, that $\alpha^{\mathfrak{I}} \circ (\underline{\delta}^{\mathfrak{I}})^{-1}(\langle \mathcal{O}(S_i) \rangle_{i \in I}) \subseteq \mathcal{O}(S_{abs})$.

(*Validity*): To see that the validity condition holds, suppose that X is a computation for M, such that $X^{(i)} \in V_i$ for each $i \in I$. Then $\delta_i^{\mathfrak{I}}(\mathrm{Obs}_X) \in \mathcal{O}(S_i)$ for each $i \in I$. It therefore follows, by the previous paragraph, that $\alpha^{\mathfrak{I}}(\mathrm{Obs}_X) \in \mathcal{O}(S_{abs})$. This means that there exists a computation X_{abs} of M_{abs}, such that $X_{abs} \in V_{abs}$ and $\mathrm{Obs}_{X_{abs}} = \alpha^{\mathfrak{I}}(\mathrm{Obs}_X)$. Since S_{abs} is assumed regular, and $\mathrm{Obs}_{X^{(abs)}} = \mathrm{Obs}_{X_{abs}}$, it follows that the computation $X^{(abs)}$ is also in V_{abs}, as required.

(*Maximality*): To prove maximality, suppose $q \in Q_M$ is reachable, and that $e \in E_M$ is such that $\delta_i^{\mathfrak{I}}(e)$ is enabled for M_i in state $\pi_i(q)$, for each $i \in I$. We wish to show that $\alpha^{\mathfrak{I}}(e)$ is enabled for M_{abs} in state $\pi_{abs}(q)$.

The proof is of necessity somewhat roundabout, since the assumption of $\mathfrak{I}$-correctness is the only information we have at our disposal concerning the relationship between the computations of M_{abs} and those of the M_i. The idea is as follows: We first obtain a computation X of M that arrives at state q at time n, and such that no non-λ events occur on the interval $[n, \infty)$. Let $x = \text{Obs}_X$. For each $i \in I$, we can modify $X^{(i)}$ to obtain a computation X_i for M_i, by letting the event $\delta_i^{\mathfrak{I}}(e)$ occur at time n. Of course, we do not yet know that we can modify $X^{(\text{abs})}$ in a similar way -- this is what we are trying to show.

We next use the orthogonality assumption on the S_i to obtain, for each $i \in I$, a valid computation Y_i that "looks like" X_i on the initial segment $[0, n+1)$. Each Y_i produces an observation $y_i \in \mathcal{O}(S_i)$ that looks like $\delta_i^{\mathfrak{I}}(x)$ on the interval $[0, n+1)$. We do not know that there is a single observation y such that $y_i = \delta_i^{\mathfrak{I}}(y)$ for all $i \in I$. However, we can use Lemma 6.3, plus the composition closure property of the specification domain $\mathfrak{I}$ to obtain an observation z such that, for all $i \in I$, $\delta_i^{\mathfrak{I}}(z) \in \mathcal{O}(S_i)$ and $\delta_i(z)$ looks like y_i on the interval $[0, n+1)$. $\mathfrak{I}$-correctness implies that $\alpha^{\mathfrak{I}}(z) \in \mathcal{O}(S_{\text{abs}})$.

Since $\alpha^{\mathfrak{I}}(z) \in \mathcal{O}(S_{\text{abs}})$, we can obtain a computation Z_{abs} for M_{abs}, such that $\text{Obs}_{Z_{\text{abs}}} = \alpha^{\mathfrak{I}}(z)$. Now, event $\alpha^{\mathfrak{I}}(e)$ occurs at time n in Z_{abs}. If we knew that $\text{State}_{Z_{\text{abs}}}(n) = \pi_{\text{abs}}(q)$, then this would show that $\alpha^{\mathfrak{I}}(e)$ is enabled for M_{abs} in state $\pi_{\text{abs}}(q)$. Although it need not be the case that $\text{State}_{Z_{\text{abs}}}(n) = \pi_{\text{abs}}(q)$, the quasi-determinacy of S_{abs} lets us replace the $[0, n]$ segment of $\text{State}_{Z_{\text{abs}}}$ with the corresponding segment of $\text{State}_{X^{(\text{abs})}}$, with the result still a computation of M_{abs}. Since $\text{State}_{X^{(\text{abs})}}(n) = \pi_{\text{abs}}(q)$, this will complete the proof.

Formally, since q is reachable for M, by definition of reachability there exist $q_0, q_1, \ldots, q_n \in Q_M$ and $e_0, e_1, \ldots, e_{n-1} \in E_M$, such that $q_0 \in \text{Init}_M$, $q_n = q$, and $\langle q_k, e_k, q_{k+1}\rangle \in \text{Trans}_M$ for all $k \in \{0, 1, \ldots, n-1\}$.

Let $f\colon \mathcal{N} \to \text{Steps}(E_M, Q_M)$ be defined by:

$$\begin{aligned} f(k) &= \langle q_k, e_k, q_{k+1}\rangle, \text{ if } 0 \le k < n \\ &= \langle q_n, \lambda, q_n\rangle, \quad \text{otherwise.} \end{aligned}$$

Then f is a history skeleton and by Lemma 3.5 there is a unique history X such that f spans X. Let $x = \text{Obs}_X$. It is easy to see that X is a computation of M.

Since $\delta_i^{\mathfrak{I}}(e)$ is enabled for M_i in state $\pi_i(q)$, for each $i \in I$ we can choose $r_i \in Q_{M_i}$ such that $\langle \pi_i(q), \delta_i^{\mathfrak{I}}(e), r_i \rangle \in \mathrm{Trans}_{M_i}$. It follows that for each $i \in I$, the history X_i, where $X_i =_n X^{(i)}$ and

$$\begin{aligned} \mathrm{Obs}_{X_i}(t) &= \delta_i^{\mathfrak{I}}(e), \text{ for } t = n \\ &= \lambda, \quad \text{for } t \in (n, \infty) \\ \mathrm{State}_{X_i}(t) &= r_i, \text{ for } t \in [n, \infty) \end{aligned}$$

is a computation for M_i. Let $x_i = \mathrm{Obs}_{X_i}$ for each $i \in I$.

By the assumption that each S_i is orthogonal, we can obtain computations $Y_i \in V_i$ such that $Y_i =_{n+1} X_i$. Let $y_i = \mathrm{Obs}_{Y_i}$. Then $y_i \in \mathcal{O}(S_i)$ and $y_i =_{n+1} x_i$.

Since each S_i is locally $\mathfrak{I}$-consistent, and the the specification domain $\mathfrak{I}$ is evolutionary, we can apply Lemma 6.3 to show that $\mathrm{future}_{y_i, n+1}(\mathcal{O}(S_i))$ contains a $\mathfrak{I}$-behavior B_i of interface $F_i^{\mathfrak{I}}$ for each $i \in I$. Let $B = (\underline{\delta}^{\mathfrak{I}})^{-1}(\underline{B})$. Then since the specification domain $\mathfrak{I}$ is closed under composition, it follows that B is a $\mathfrak{I}$-behavior of interface $E^{\mathfrak{I}}$. Since $\mathfrak{I}$-behaviors are nonempty by the nondegeneracy property of $\mathfrak{I}$, it follows that we can choose an element z' of B. Let z be the observation defined by the properties:

$$\begin{aligned} &z =_n x \\ &z(n) = e \\ &z(t) = \lambda, \text{ for } t \in (n, n+1) \\ &\mathrm{suffix}_{n+1}(z) = z'. \end{aligned}$$

Then by construction, $\delta_i^{\mathfrak{I}}(z) \in \mathcal{O}(S_i)$ for each $i \in I$. As shown in the first paragraph of the proof, it follows by $\mathfrak{I}$-correctness that $\alpha^{\mathfrak{I}}(z) \in \mathcal{O}(S_{\mathrm{abs}})$.

Since $\alpha^{\mathfrak{I}}(z) \in \mathcal{O}(S_{\mathrm{abs}})$, there exists a computation Z_{abs} for M_{abs} with $\mathrm{Obs}_{Z_{\mathrm{abs}}} = \alpha^{\mathfrak{I}}(z)$. By construction, $\mathrm{Obs}_{Z_{\mathrm{abs}}} = \alpha^{\mathfrak{I}}(z) =_n \alpha^{\mathfrak{I}}(x) = \mathrm{Obs}_{X^{(\mathrm{abs})}}$. Since S_{abs} is quasi-determinate, there exists a computation Z_{abs}' such that $Z_{\mathrm{abs}}' =_n X^{(\mathrm{abs})}$ and $\mathrm{Obs}_{Z_{\mathrm{abs}}'} = \mathrm{Obs}_{Z_{\mathrm{abs}}}$. Since $Z_{\mathrm{abs}}' =_n X^{(\mathrm{abs})}$, we know that $\mathrm{State}_{Z_{\mathrm{abs}}'}(n) = \pi_{\mathrm{abs}}(q)$. Since $\alpha^{\mathfrak{I}}(e)$ occurs at time n in Z_{abs}', it follows that $\alpha^{\mathfrak{I}}(e)$ is enabled for M_{abs} in state $\pi_{\mathrm{abs}}(q)$, as desired. ∎

7. Conclusion

7.1 Summary

The important accomplishments of this thesis are the following:

1. **Formal Framework** - A major accomplishment of this thesis is that it sets up a formal framework within which it is possible to formulate precisely a large number of interesting and important questions about specifications and correctness proofs, and to obtain rigorous answers to these questions. The framework includes the notions of *interface*, *observation behavior*, *composition*, and *abstraction*, as primitive. These primitive notions are used to give precise, language-independent definitions of the notions of *implementation*, *correctness*, and *consistency*.

2. **State-Transition Specifications** - The thesis shows how module behaviors can be conveniently and naturally described in terms of a machine that generates an observation as it executes, plus some validity conditions on the computations of that machine. Specifications stated in such a form lend themselves to a systematic method for performing correctness proofs.

3. **Rely- and Guarantee-Conditions** - The concept of rely- and guarantee-conditions is shown to be useful for organizing eventuality specifications and proofs of correctness involving such specifications. The use of rely- and guarantee-conditions seems to result in simple proofs based on the communication structure of a system, rather than in proofs based on the structure of computations.

4. **Consistency of Specifications** - The I/O-behavior model provides an interesting and useful notion of consistency for eventuality specifications. The thesis obtains a technique for proving the I/O-consistency of state-transition specifications.

The investigation of state-transition specification performed in this thesis has resulted in some practical insights that can be tentatively expressed in the form of the following procedure for refinement of an abstract module into a system of component modules:

(1) Determine the interconnection of component modules that will be used to implement the abstract module.

(2) Identify the implementation invariant and the rely-/guarantee-conditions required for the proof of correctness.

(3) "Localize" the rely-/guarantee-conditions to each component module.

introduce sufficient information into the component module states to permit the localized conditions to be conveniently expressed.

(4) Define the state-transition relations for each component module.

(5) Check the completed component module specifications by proving their consistency.

(6) Use the component module specifications to perform a complete proof of correctness for the implementation.

The resource manager example in Appendix II illustrates the use of this procedure.

It has unfortunately been possible in this thesis to investigate only a tiny fraction of the questions that could conceivably be formulated using the framework developed here. The remainder of this chapter lists a number of questions that have not been addressed, but should be. Hopefully the answers to these questions can provide further practical insights into the problem of design, and ultimately contribute to more useful and reliable distributed/concurrent systems.

7.2 Ideas for Future Work

The basic framework set up in this thesis can serve as a starting point for a number of interesting extensions. The discussion below is concerned with the following broad possibilities for investigation:

(1) Specification Domains

(2) Semantic Properties of State-Transition Specifications

(3) Organizing Principles for Specifications and Proofs

(4) Formal Specification and Proof

(5) Non-State-Transition Specifications

7.2.1 Specification Domains

The concept of a specification domain appears to offer considerable possibilities for theoretical investigation. There are two broad directions for future investigation of specification domains. The first direction is concerned with developing the general theory of specification domains, and relating this theory to domain theory as used in programming language semantics. The second direction is to construct additional example specification domains that model systems with interesting properties.

Plausible steps toward a general theory of specification domains might include the following:

(1) The definition of a specification domain should be generalized so that the particular structure of an observation is not specified. The assumption of the particular structure of translations between interfaces would also have to be removed. A reasonable approach might be to assume that the interfaces and translations comprise the objects and morphisms of a category. The relationship between interfaces and observations would take the form of a functor defined on the category of interfaces, which maps each interface E to the set $\mathrm{Obs}(E)$ of observations over E, and which maps each translation $\alpha: E \rightarrow F$ to a function on observations.

(2) The notion of a behavior should be generalized so that a behavior determines, but is not identified with, a set of observations. This would permit behaviors such as the "futures processes" of [Rounds81] to be used, as well as correspondingly more general abstraction and composition operations. There still should be some constraints on the effect of abstraction and composition operations with respect to the set of observations determined by a behavior. It is not immediately obvious what those constraints should be.

(3) An attempt should be made to try to identify the correct set of regularity assumptions for abstraction and composition operations. The results of Chapter 6 required no such assumptions, however it seems reasonable that the classes of abstraction and decomposition maps ought to be closed under function composition and include the identity translations. The I/O-abstraction maps and I/O-decomposition maps certainly have these properties.

(4) The specification domain I/O seems to provide motivation for a kind of duality between abstraction and decomposition, in the sense that abstraction and decomposition maps seem to have complementary preservation properties with respect to input and output. It would be very interesting if abstraction and decomposition maps could be unified, so that they are just dual instances of a single underlying notion of translation, or "interface morphism." One way this might be accomplished is by assuming the existence of a kind of conjugation operation on interfaces. Intuitively, the conjugate of a module interface would be the interface of the module's environment. The duality between abstraction and decomposition might then be captured by stating that a decomposition map is an abstraction map defined on conjugate interfaces.

To help motivate the correct general definitions, further specific examples of specification domains should be constructed and studied. Ideas for constructing further examples of specification domains might be as follows:

(1) Different notions of observation might be used to construct a number of interesting specification domains. One example is to replace the assumption that observations contain only finitely many events in any finite interval with some less restrictive topological assumption, and to attempt to construct corresponding classes of behaviors. If the machine approach to defining behaviors is to be used, then there is the problem of how to define a machine that permits infinitely many events to occur in a finite interval. Examples of such "machines" already appear in the theory of dynamical systems. For example, if one is willing to assume that an observation is a continuous, differentiable function on $[0, \infty)$, then the correct notion of machine is that of a differential equation.

(2) Different special assumptions on behaviors can be made to model systems with particular properties. For example, it would be interesting to find a class of behaviors that includes non-asynchronous behaviors, corresponding to sets of observations that are not necessarily closed under stretching of the time axis. These behaviors would model timing-dependent systems. If observations contain space coordinates, in addition to time coordinates, then it might be possible to construct a class of behaviors with the property that information doesn't travel "too quickly" from one place to another. This specification domain could be used to investigate the problem of what can be observed by one module about the operation of another in a distributed system. Another idea might be to try to characterize a class of "atomic" behaviors, like the *atomic data types* of [Weihl84]. The observations in these behaviors would have certain serializability properties.

(3) An attempt should be made to deal correctly with simultaneity. It should be possible to do this within the specification domain framework as follows: Introduce additional structure on interfaces to model the intuitive idea that some events represent the simultaneous occurrence of more primitive events. For example, it might be assumed that the events in an interface form a complete lower semilattice with λ at the bottom, and with the semilattice operation $\sqcup$ representing the operation of "simultaneous occurrence." The main problem with this approach is how to introduce the notions of input and output so that an assignment of behaviors that is nondegenerate and closed under composition can be defined.

7.2.2 Semantic Properties of State-Transition Specifications

In Chapter 6 three semantic properties of state-transition specifications were identified (determinacy, regularity, orthogonality) and it was suggested that these might be properties characteristic of "well-formed" specifications. The idea of finding semantic well-formedness properties of specifications also appears in [Jones81], where the notion of an "unbiased" specification is discussed. It is interesting and useful to try to identify such properties, since they can possibly serve as guidelines in the design process. An important extension to this thesis would be to try to examine more closely the properties identified in Chapter 6, to develop techniques for proving that specifications have these properties, and to try to develop additional well-formedness properties.

7.2.3 Organizing Principles for Specifications and Proofs

The development of organizing principles for specifications and proofs appears to be a promising area of investigation. The rely- and guarantee-condition approach to writing specifications and performing correctness proofs is an example of the kind of results one might try to obtain. The way to proceed in this area is to perform example specifications and correctness proofs, and then try to abstract from these examples something in the way of general methods that would be applicable to other examples. This is difficult, because the examples take a long time to do, and it is hard to abstract general methods from a few examples.

Rely-/Guarantee-Conditions:

Rely- and guarantee-conditions were used in this thesis in the statement of the validity condition portion of a specification only. This is in contrast to the work of other researchers, for example [Jones81], in which rely- and guarantee-conditions can be used for state-transition properties only. For the examples in this thesis it did not seem particularly helpful to use rely- and guarantee-conditions for the state-transition portion of a specification. One possible exception might be the synchronizer and synchronizer component module specifications, in which the use of rely- and guarantee-conditions in the state-transition part of the specification might obviate the need for an error state.

Determinate vs. Indeterminate Specifications:

Both determinate and indeterminate specifications seem to be useful. From a strictly theoretical standpoint, determinate specifications are more convenient to work with than indeterminate specifications. From a practical point of view, though, there are cases (such as the transmission line specification of Appendix II) in which the use of indeterminate specifications is quite natural, and in which an equivalent determinate specification would have to be stated in a much more convoluted fashion. Perhaps a result could be proved which shows that determinate and indeterminate specifications are equivalent in expressive power, in the sense that every indeterminate specification could be stated equivalently as a determinate specification. Such a result would permit the theory of specification to deal only with the more convenient determinate specifications, while permitting indeterminate specifications to be used in examples where they seem natural.

Parallel Specifications:

In certain examples, though not in any of the ones considered in this thesis, it is convenient to describe the desired functioning of a module in terms of a collection of loosely interacting concurrent processes. This process structure is a logical one used for descriptive purposes only, and may or may not bear any relation to the structure of an implementation of the module. It would be nice to be able to write specifications that reflect such a logical decomposition. State-transition specifications as described in this thesis are an inherently sequential form of description, since they include only a single machine. Perhaps the state-transition technique could be extended by permitting specifications to include a collection of machines that execute in parallel, and whose state sets are mostly independent of each other. To perform correctness proofs with this kind of specification would require a modified version of the Correctness Theorem.

Differential vs. Integral Form:

There is a certain amount of flexibility in whether state-transition properties are expressed in "differential," state-transition form, or in "integral," invariant form. In general, given a statement of the invariant form, "for all reachable states *q*, property *P*(*q*) holds," an equivalent expression in state-transition form can be obtained by a simple syntactic transformation analogous to differentiation, e.g. "Property *P* holds of all initial states, and a state transition from q to r can occur only if $P(q)$ implies $P(r)$." There is apparently no general method for "integration," that is, for obtaining equivalent statements in invariant form, given a statement in state-transition form.

In this thesis, the policy was adopted that all local properties would be expressed in state-transition form, rather than in invariant form. One reason for this is that, in general, invariants for the composite machine for an implementation cannot be proved directly from invariants for the component machines. Rather, it is necessary to first "differentiate" the invariants for the components, to obtain corresponding preconditions for event occurrences, and then use these preconditions in an inductive proof of the desired invariant for the composite machine. In certain circumstances, though, it seems natural to express specifications in invariant, rather than state-transition form. For example, in the synchronizer module specification it is perhaps more natural to state explicitly that "at most one user process can be running at any instant," rather than the more indirect approach taken here, where we use the precondition "a run event can occur only if there are no users currently running." Further investigation into the relationship between state-transition and invariant specifications seems needed.

7.2.4 Formal Specification and Proof

For the specification and proof techniques developed in this thesis to be useful for practical examples, the development of mechanical aids for manipulating specifications and assisting in correctness proofs is essential. Appendix I takes the first steps toward this goal by showing how all of the proof techniques developed in this thesis can be formalized within an appropriate temporal language. Further steps should be taken along the following lines:

(1) A practical method should be devised for describing heterogeneous algebras and for associating with each description a reasonably powerful, sound deductive system for deducing properties of the described algebra. In spite of the large amount of work that has been done in this area (specification of abstract data types), a completely satisfactory method is still lacking.

(2) Tools are needed for enumeration and checking of cases in inductive proofs of invariance. In the correctness proofs performed in this thesis, once the implementation invariant is devised, the proof that it is inductive is a tedious case

analysis that ought to be easily mechanizable.

(3) Mechanical aids for checking proofs in temporal logic are needed. Such a proof checker would probably not be capable of performing complete proofs by itself, but rather would serve to fill in intermediate steps in a proof generated by a human verfier.

7.2.5 Non-State-Transition Specifications

It would be interesting to use the framework of definitions set up in Chapter 2 to investigate specification languages not based on the state-transition approach. One obvious example is to investigate specification languages based on some kind of generalized regular expression. Preliminary experience with this kind of specification seems to indicate that the regular expression approach seems to produce shorter specifications for trivial examples, but for more complex examples it is much more difficult to express the desired properties. Interesting questions are what sort of deductive system, if any, could be used to derive consequences from specifications stated in regular expression form, and what form the Correctness Theorem would take for such specifications.

References

[Abrial80] Abrial, J.R., "The Specification Language Z: Syntax and Semantics," Programming Research Group, Oxford University, 1980.

[Apt81] Apt, K., "Ten Years of Hoare's Logic: A Survey - Part I," TOPLAS 3, 4(1981), pp. 431-483.

[Barringer83] Barringer, H., Kuiper, R., "A Temporal Logic Specification Method Supporting Hierarchical Development," Manuscript, University of Manchester Department of Computer Science, November, 1983.

[Bartlett69] Bartlett, K.A., et al. "Note on Reliable Full Duplex Transmission on Half Duplex Links," CACM 12, 5(May 1969), pp. 260-261.

[Berzins79] Berzins, V.A., "Abstract Model Specifications for Data Abstractions," MIT/LCS/TR-221, 1979.

[Bochmann78] Bochmann, G.V., "Finite State Description of Communication Protocols," Computer Networks 2(1978), pp. 361-372.

[Brock81] Brock, J.D., Ackermann, W.B., "Scenarios: A Model of Non-determinate Computation," Proc. Peniscola Colloquim, Springer LNCS 107, 1981.

[Brock83] Brock, J.D., "A Formal Model of Non-Determinate Dataflow Computation," MIT/LCS/TR-309.

[Chen81] Chen, B., Yeh, R.T., "Event Based Behavior Specification of Distributed Systems," IEEE Symposium on Reliability in Distributed Software and Database Systems, July, 1981.

[Chen82] Chen, B., "Event-Based Specification and Verification of Distributed Systems," PhD Dissertation, University of Maryland, 1982.

[Clinger81] Clinger, W. "Foundations of Actor Semantics," MIT/AI/TR-633, May, 1981.

[Dijkstra76] Dijkstra, E.W., A Discipline of Programming, Prentice Hall, 1976.

[DiVito82] DiVito, B.L., "Verification of Communications Protocols and Abstract Process Models," Institute for Computing Science TR-25, University of Texas at Austin, 1982.

[Fischer83] Fischer, M.J., Griffeth, N.D., Guibas, L.J., Lynch, N.A., "Probabilistic Analysis of a Network Resource Allocation Algorithm," submitted for publication.

[Floyd67] Floyd, R.W., "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science, American Math. Soc., 1967.

[Francez79] Francez, N., et. al., "Semantics of Nondeterminism, Concurrency, and Communication JCSS 19(1979), pp. 290-308.

[Goguen78] Goguen, J.A., Thatcher, J.W., Wagner, E.G., "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in Current Trends in Programming Methodology, Vol. IV, Data Structuring, R.T. Yeh, ed., Prentice-Hall, 1978.

[Good79] Good, D.I., Cohen, R.M., Keeton-Williams, J., "Principles of Proving Concurrent Programs in GYPSY," 6th POPL, 1979.

[Good82] Good, D.I., "The Proof of a Distributed System in GYPSY," Technical Report 30, University of Texas at Austin, September 1982.

[Gordon79] Gordon, M.J.C., "The Denotational Description of Programming Languages," Springer-Verlag, 1979.

[Goree81] Goree, J.A., "Internal Consistency of a Distributed Transaction System with Orphan Detection," MIT/LCS/TR-286, Mass. Institute of Technology, 1981.

[Greif75] Greif, I. "Semantics of Communicating Parallel Processes," MIT/LCS/TR-154, September, 1975.

[Guttag78] Guttag, J.V., Horowitz, E., Musser, D.R., "Abstract Data Types and Software Validation," CACM 21, 12(Dec. 1978), pp. 1048-1064.

[Guttag80] Guttag, J., Horning, J., "Formal Specification as a Design Tool," 7th POPL, 1980, pp. 251-261.

[Hailpern80] Hailpern, B.T., Owicki, S.S., "Verifying Network Protocols Using Temporal Logic," Technical Report No. 192, Computer Systems Laboratory, Stanford University, June, 1980.

[Hailpern81] Hailpern, B.T., Owicki, S.S., "Modular Verification of Computer Communication Protocols," IBM Research Report RC 8726, March, 1981.

[Harel78] Harel, D., "Logics of Programs: Axiomatics and Descriptive Power," MIT LCS TR-200, May, 1978.

[Hewitt77] Hewitt, C., Baker, H., "Laws for Communicating Parallel Processes," IFIP 77, Toronto, August, 1977.

[Hoare69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," CACM, Vol. 21, October, 1969.

[Hoare72] Hoare, C:A.R., Proof of Correctness of Data Representations, Acta Informatica 1, 4(1972) pp. 271-281.

[Hoare78] Hoare, C.A.R., "Communicating Sequential Processes," CACM, Vol. 21, August, 1978.

[Hoare81a] Hoare, C.A.R., Brookes, S.D., Roscoe, A.W., "A Theory of Communicating Sequential Processes," Technical Monograph PRG-22, Oxford University Computing Laboratory, May, 1981.

[Hoare81b] Hoare, C.A.R., "A Model for Communicating Sequential Processes," Technical Monograph PRG-22, Oxford University Computing Laboratory, June, 1981.

[Jones81] Jones, C.B., "Development Methods for Computer Programs Including a Notion of Interference," Wolfson College, June, 1981.

[Jones83] Jones, C.B., "Specification and Design of (Parallel) Programs," IFIP 83.

[Kahn74] Kahn, G., "The Semantics of a Simple Language for Parallel Processing," IFIP 74, pp. 471-475.

[Kahn77] Kahn, G., MacQueen, D.B., "Coroutines and Networks of Parallel Processes," IFIP 77, pp. 993-998.

[Kapur80] Kapur, D., "Towards a Theory for Abstract Data Types," MIT/LCS/TR-237, May, 1980.

[Keller76] Keller, R.M., "Formal Verification of Parallel Programs," CACM 19,7(July 1976), pp. 371-384.

[Lamport80] Lamport, L., "'Sometime' is Sometimes 'Not Never': On the Temporal Logic of Programs," ACM POPL 1980.

[Lamport83] Lamport, L., "Specifying Concurrent Program Modules," TOPLAS, 1983.

[Lansky83] Lansky, A.L., Owicki, S., "GEM: A Tool for the Description of Concurrency Primitives and Verification of Concurrent Programs," PODC 83.

[Liskov79] Liskov, B.H., "Modular Program Construction Using Abstractions," MIT Computation Structures Group Memo 184, September, 1979.

[Lynch81] Lynch, N.A., Fischer, M.J., "On Describing the Behavior and Implementation of Distributed Systems," Theoretical Computer Science 13(1981), pp. 17-43.

[Lynch83] Lynch, N.A., "Concurrency Control for Resilient Nested Transactions," ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, March, 1983.

[Milner80] Milner, R., A Calclulus of Communicating Systems, Springer Lecture Notes in Computer Science 92, 1980.

[Misra81] Misra, J., Chandy, K.M., "Proofs of Networks of Processes," IEEE TOSE, Vol. SE-7, No. 4, July 1981.

[Misra82] Misra, J., Chandy, K.M., Smith, T., "Proving Safety and Liveness of Communicating Processes with Examples," ACM PODC 1982.

[Owicki76] Owicki, S., Gries, D., "Verifying Properties of Parallel Programs: An Axiomatic Approach," CACM 15, 5(1976).

[Parnas72] Parnas, D.L., "A Technique for Software Module Specification with Examples," CACM 15, 5(May, 1972), pp. 330-336.

[Pneuli77] Pnueli, A., "The Temporal Logic of Programs," FOCS 1977.

[Pratt82] Pratt, V.R., "On the Composition of Processes," ACM POPL 1982.

[Rounds81] Rounds, W.C., Brookes, S.D., "Possible Futures, Acceptances, Refusals, and Communicating Processes," FOCS 1981.

[Schwabe81a] Schwabe, D., "Formal Techniques for the Specification and Verification of Protocols," Report No. CSD-810401, UCLA Computer Science Department, April, 1981.

[Schwabe81b] Schwabe, D., "Formal Specification and Verification of a Connection-Establishment Protocol," USC-ISI Tech. Rpt. ISI/RR-81-91, April, 1981.

[Schwartz81] Schwartz, R.L., Melliar-Smith P.M., "Temporal Logic Specification of Distributed Systems," Second International Conference on Distributed Systems, INRIA, April, 1981.

[Sunshine78] Sunshine, C.A., "Survey of Protocol Definition and Verification Techniques," Computer Networks 2(1978), pp. 346-350.

[Weihl84] Weihl, W.E., "Specification and Implementation of Atomic Data Types," PhD Thesis, MIT, March, 1984.

[Wing83] Wing, J.M., "A Two-Tiered Approach to Specifying Programs," MIT/LCS/TR-299, 1983.

[Wirth71] Wirth, N., "Program Development by Stepwise Refinement," CACM 14, 4(April 1971), pp. 221-227.

[Wolper82] Wolper, P., "Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic," ACM POPL 1982.

[Yonezawa77] Yonezawa, A., "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics," MIT/LCS/TR-191, December, 1977.

Appendix I - Formal Specification and Proof

The purpose of this appendix is to outline the way in which the informal state-transition specification and proof techniques used in this thesis can be formalized, perhaps to permit mechanically-assisted specification and verification. The major new concepts introduced to permit this formalization are those of an "event/state algebra" and an "implementation algebra." An event/state algebra is a heterogeneous algebra that embeds the machine part of a state-transition specification. An "implementation algebra" is a special kind of event/state algebra, which embeds the composite machine for an implementation, and which contains among its operations the abstraction and decomposition map for the implementation.

The utility of event/state algebras and implementation algebras derives from the fact that associated with each event/state algebra A (and hence each implementation algebra as well) is a temporal logic language $\mathfrak{T}(A)$, within which can be expressed properties of the computations of the embedded machine. Each of the proof techniques presented in this thesis has the property that its hypotheses can be formalized in terms of the validity of *verification conditions*, which are sentences expressed in the temporal language of an appropriate event/state algebra. The problem of formalizing proofs that use the techniques of this thesis is thereby reduced to the following two problems:

(1) Find a convenient method for describing event/state algebras.

(2) Find a general method whereby the description of an event/state algebra A can be used to obtain a formal deductive system for deriving a large number of true statements about A, where these statements are expressed in the temporal language $\mathfrak{T}(A)$.

In this appendix, the following tasks are accomplished:

(1) The notions of event/state algebra and implementation algebra are defined.

(2) Precise semantics are given for the temporal language $\mathfrak{T}(A)$ associated with an event/state algebra A.

(3) An approach, based on set theory, for describing event/state algebras is sketched. It is indicated how, from the description of an event/state algebra A, an A-sound deductive system for the language $\mathfrak{T}(A)$ might be obtained.

(4) It is show how the various proof techniques presented in this thesis can be

formalized in the language $\mathcal{T}(A)$ for an appropriate A.

I.3 Event/State Algebras

Definition - An *event/state algebra A* is a heterogeneous algebra whose signature is of the form: <Events$_A$, States$_A$, Init$_A$, Trans$_A$, λ_A, ... >, where λ_A is a distinguished constant of sort *Events*$_A$ so that <Events$_A$, λ_A> is an interface, and <Events$_A$, States$_A$, Init$_A$, Trans$_A$> is a machine, which we call the *embedded machine* and which we denote by Mach$_A$. ∎

When there is only one event/state algebra under consideration, we will omit the identifying subscripts. The ellipsis in the signature of A indicates that A is permitted to contain additional sorts, relations, and functions besides those explicitly listed. The reason for permitting A to contain these additional sorts, relations, and functions, is to provide a mechanism by which the temporal language $\mathcal{T}(A)$ can be made as expressive as desired.

We now define precisely the syntax and semantics of the temporal language $\mathcal{T}(A)$ of an event/state algebra A. Let Σ_A be the signature of A. The signature Σ_A is required to contain distinguished sorts *Events* and *States*. In addition, we assume that corresponding to each sort of Σ_A is a countably infinite collection of variables which we use to range over values of that sort. The language $\mathcal{T}(A)$ contains syntactic categories of "terms," "atomic formulas," and "formulas," which are defined by induction as follows:

Terms:

(1) The distinguished symbols **Now** and **After** are terms of sort *States*.

(2) The distinguished symbol **Occurs** is a term of sort *Events*.

(3) If v is a variable of sort S, then v is a term of sort S.

(4) If $t_1, \ldots, t_n$ are terms of sorts $S_1, \ldots, S_n$, respectively, and **f** is an n-ary function symbol of type $S_1 \times \ldots \times S_n \rightarrow S$, then $\mathbf{f}(t_1 \ldots t_n)$ is a term of sort S.

Atomic Formulas: If $t_1, \ldots, t_n$ are terms of sorts $S_1, \ldots, S_n$, respectively, and **R** is an n-ary relation symbol of type $S_1 \times \ldots \times S_n$, then $\mathbf{R}(t_1 \ldots t_n)$ is an atomic formula.

Formulas:

(1) An atomic formula is a formula.

(2) If φ and ψ are formulas, and v is a variable of sort S, then $\neg\varphi$, $\varphi \vee \psi$, and $(\exists v \in S)\varphi$ are formulas.

(3) If φ is a formula, then $\Box\varphi$ is a formula.

The sets of terms, atomic formulas, and formulas of $\mathcal{T}(A)$ are the least sets with the properties listed.

The first-order language $\mathcal{L}(A)$ is the sublanguage of $\mathcal{T}(A)$ obtained by omitting formation rules (1) and (2) under "Terms," and formation rule (3) under "Formulas." We treat the additional logical connectives $\wedge$, $\rightarrow$, $\leftrightarrow$, $\forall$ as abbreviations in the usual way. In addition, the temporal operator $\Diamond$ is regarded as an abbreviation for $\neg\Box\neg$.

We use the notation $t(v_1 \dots v_n)$ to denote a term t whose variables are a subset of the set $\{v_1, \dots, v_n\}$, and the notation $\varphi(v_1 \dots v_n)$ to denote a formula whose free variables are a subset of $\{v_1, \dots, v_n\}$. The notations $t(t_1/v_1 \dots t_n/v_n)$ and $\varphi(t_1/v_1 \dots t_n/v_n)$ denote the result of substituting the terms $t_1, \dots, t_n$ for free occurrences of the variables $v_1, \dots, v_n$ in t and φ, respectively.

Next, we define the semantics of $\mathcal{T}(A)$. If S is a symbol (sort name, function symbol, or relation symbol) in the signature of A, then we use S_A to represent the denotation (set, function, or relation) assigned by A to the symbol S. Define an *interpretation* for a sequence $v_1, \dots, v_n$ of variables of sorts $S_1, \dots, S_n$, respectively, to be a sequence $a_1, \dots, a_n$ of elements of A, where each a_k is of sort S_k. The semantics of $\mathcal{T}(A)$ are defined in two parts. First, given an intepretation $a_1, \dots, a_n$ for the free variables $v_1, \dots, v_n$, a term $t(v_1 \dots v_n)$ of sort S denotes a function $t[a_1/v_1 \dots a_n/v_n]$ from Steps(Events$_A$, States$_A$) to S_A, whose value on the step $s = \langle q, e, r\rangle$ is defined as follows:

(1) If t is **Now**, then $t[a_1/v_1 \dots a_n/v_n](s) = q$.
If t is **After**, then $t[a_1/v_1 \dots a_n/v_n](s) = r$.

(2) If t is **Occurs**, then $t[a_1/v_1 \dots a_n/v_n](s) = e$.

(3) If t is the variable v_k, then $t[a_1/v_1 \dots a_n/v_n](s) = a_k$.

(4) If t is $\mathbf{f}(t_1 \dots t_n)$, then $t[a_1/v_1 \dots a_n/v_n](s) = \mathbf{f}_A(b_1 \dots b_n)$,
where $b_k = t_k[a_1/v_1 \dots a_n/v_n](s)$ for each k.

The second part of the definition of the semantics of $\mathcal{T}(A)$ is concerned with when a formula $\varphi(v_1 \dots v_n)$ is *satisfied* by a history $X \in$ Hist(Events$_A$, States$_A$), and an intepretation $a_1, \dots, a_n$ for $v_1, \dots, v_n$. We abbreviate this as $X \models_A \varphi[a_1/v_1 \dots a_n/v_n]$, or, when the algebra A is clear from the context, as simply $X \models \varphi[a_1/v_1 \dots a_n/v_n]$.

Atomic Formulas: If φ is the atomic formula $\mathbf{R}(t_1 \dots t_m)$, where the free variables of each t_k are in the set $\{v_1, \dots, v_n\}$, then $X \models \varphi[a_1/v_1 \dots a_n/v_n]$ iff $\langle b_1, \dots, b_n \rangle \in \mathbf{R}_A$, where $b_k = t_k[a_1/v_1 \dots a_n/v_n](\mathrm{Step}_X(0))$ for each k.

Formulas: If φ is a formula, but not an atomic formula, then

(1) If φ is $\neg\psi$, $\psi \vee \chi$, or $(\exists v \in S)\psi$, then satisfaction for φ is defined by induction in the usual way.

(2) If φ is $\Box\psi$, then $X \models \varphi[a_1/v_1 \dots a_n/v_n]$ iff $\mathrm{suffix}_t(X) \models \varphi[a_1/v_1 \dots a_n/v_n]$ for all $t \in [0, \infty)$.

Suppose that $\varphi(v_1 \dots v_n)$ is a formula of $\mathcal{T}(A)$ and that Ψ is a set of formulas of $\mathcal{T}(A)$, the free variables of which are a subset of $\{v_1, \dots v_n\}$. We say that φ is a *consequence of* Ψ *in* A, written $\Psi \models_A \varphi$, if whenever $a_1, \dots, a_n$ is an interpretation for the variables $v_1, \dots, v_n$, and $X \in \mathrm{Hist}(\mathrm{Events}_A, \mathrm{States}_A)$ is such that $X \models \psi[a_1/v_1 \dots a_n/v_n]$ for all ψ in Ψ, then $X \models \varphi[a_1/v_1 \dots a_n/v_n]$ as well. The formula φ is said to be *valid in* A, abbreviated $\models_A \varphi$, if φ is a consequence in A of the null set of formulas. A *sentence* of $\mathcal{T}(A)$ is a formula of $\mathcal{T}(A)$ that has no free variables. If φ is a sentence and ψ is a formula, then it is easily verified that $\varphi \models_A \psi$ iff $\models_A \varphi \rightarrow \psi$.

The following result makes explicit the relationship between the preceding definitions and the usual semantics of first/order logic.

Lemma I.1 - Suppose that $\varphi(v_0 \dots v_n)$ is a formula of $\mathcal{T}(A)$, containing no occurrences of $\Box$. Suppose that A is an event/state algebra, that $X \in \mathrm{Hist}(\mathrm{Events}_A, \mathrm{States}_A)$, and that $a_1, \dots, a_n$ is an interpretation for the variables $v_1, \dots, v_n$. Suppose $X(0) = \langle q, e, r \rangle$. Then

$$X \models_A \varphi[a_1/v_1 \dots a_n/v_n]$$

iff

$$\models_A \varphi[a_1/v_1 \dots a_n/v_n, q/\mathbf{Now}, e/\mathbf{Occurs}, r/\mathbf{After}],$$

where the latter is defined in the usual sense of first-order logic.

Proof - Straightforward. ∎

We recall here the definitions, given in Chapter 4, of the sentence Comp of $\mathcal{T}(A)$.

$$\mathrm{Comp} \equiv \mathrm{Init}(\mathbf{Now}) \wedge \Box\mathrm{Trans}(\mathbf{Now}, \mathbf{Occurs}, \mathbf{After})$$

Intuitively, $X \models \mathrm{Comp}$ iff X is a computation for the embedded machine Mach_A.

We conclude this section with the following definition: Suppose that A is an event/state algebra, and Valid is a sentence of $\mathfrak{T}(A)$. Then the *state-transition specification defined by the pair* $\langle A$, Valid$\rangle$ is the state transition specification $S = \langle M, V\rangle$, where $M = \mathrm{Mach}_A$, and $V = \{X \in \mathrm{Hist}(\mathrm{Events}_A, \mathrm{States}_A): X \models \mathrm{Comp} \wedge \mathrm{Valid}\}$.

I.4 Description of Event/State Algebras

In this section, we consider the problem of describing event/state algebras in such a way that a sound deductive system for $\mathfrak{T}(A)$ can be obtained from a description of the event/state algebra A. It should be noted that this problem has already received a good deal of attention in the research literature under the heading of "Specification of Abstract Data Types." In spite of the effort that has been expended on this problem, there still does not seem to be an available description method that is convenient for the purposes of this thesis. Hopefully this situation will be rectified in the near future.

The description technique we use here can be summarized as follows: We assume fixed in advance a standard "primitive" or "core" algebra with a sufficiently expressive first-order theory. Let C be the core algebra, and let T be its complete first-order theory, expressed in the language $\mathcal{L}(C)$. An event/state algebra is described by writing a collection of first-order axioms U in an extension $\mathcal{L}$ of $\mathcal{L}(C)$, that define an extension by definition of T. Such a collection of axioms defines a unique extension of the core algebra C to a model A of $T \cup U$.

We wish to obtain an A-sound deductive system for the language $\mathcal{L}(A)$ (= $\mathcal{L}$). Since we wish our description method to be powerful enough to describe algebras such as $\langle \mathcal{N}, 0, 1, +, \cdot\rangle$, which cannot be completely axiomatized, it seems unreasonable to expect the core theory T to be axiomatizable. If we fix in advance a deductive system that axiomatizes a usefully large fragment of T, though, then by augmenting this deductive system with the defining axioms U, we can hopefully obtain an axiomatization of a usefully large fragment of the complete first-order theory of A. In this thesis, we assume as the core theory some suitable variant of the theory of sets. Set theory is highly expressive, and this makes it easy to describe desired event/state algebras. However, if machine-assisted verification is a goal, then set theory might not be the most appropriate: it seems quite possible that some less expressive core theory would be more amenable to mechanization.

We next consider the problem of deduction in $\mathcal{T}(A)$. Given an event/state algebra description, which, as discussed above, we regard as denoting an extension by definition of an underlying set theory, we wish to be able to deduce a large class of A-valid formulas of $\mathcal{T}(A)$. Suppose we could somehow transform an arbitrary sentence φ of $\mathcal{T}(A)$ into a sentence φ' of $\mathcal{L}(A)$ such that $\models_A \varphi \leftrightarrow \varphi'$. In other words, suppose that we could axiomatize the temporal operator $\Box$ and special symbols **Now**, **Occurs**, and **After**, in terms of the set theoretic notions of $\mathcal{L}$. Then the problem of showing $\models_A \varphi$, where $\varphi \in \mathcal{T}(A)$, would be reduced to the problem of showing that $T \models \varphi'$, where $\varphi' \in \mathcal{L}(A)$ is the transformed version of φ.

It seems likely that the reduction described the preceding paragraph can actually be carried out, since the idea seems essentially the same as that used in the proofs [Harel78] of the "arithmetic completeness" of deductive systems for dynamic logics. Assuming that this idea works for the temporal logics $\mathcal{T}(A)$, this would give us a way of deducing all valid formulas of $\mathcal{T}(A)$, assuming we have available the complete theory of some model of set theory. Although we can never obtain a complete axiomatization of set theory, it seems likely that any of the usual collections of axioms for set theory would provide us with a deductive system for $\mathcal{T}(A)$ that is powerful enough to be useful in practice.

In practice, to write down explicitly the collection of defining axioms that describe an event/state algebra A is cumbersome. It is convenient to introduce some notation for common constructions. We do this with the understanding that descriptions expressed in this notation stand for collections of first-order defining axioms. In general, the description of an event/state algebra can be divided into two parts: one, the definition of new sorts, and two, the definition of new function and relation symbols.

We define new sorts by a set of defining equations that define the new sorts in terms of more primitive components. These equations take the form:

$$S = \mathcal{E}(S_1, \ldots, S_n),$$

where S is the new sort being defined, $S_1, \ldots, S_n$ are the names of previously-defined sorts, and $\mathcal{E}$ is an expression within which various set-theoretic constructions can appear. These defining equations are analogous to the domain equations used in the denotational definition of the semantics of a programming language [Gordon79]; however, to ensure that a set of equations can be regarded as denoting a colletion of defining axioms, we do not permit here the use of recursive equations. The set-theoretic constructions (cartesian product, disjoint union, etc.) that appear on the right-hand sides of the defining equations introduce implicitly various "built-in" functions and relations (projection, injection, etc.). The constructions we use, and their associated built-in functions and relations are listed below.

Once the equations that define the new sorts have been given, we can use these sorts and their built-in function and relations to define additional functions and relations, in particular the initial state and state-transition relations for the embedded machine. These additional functions and relations are defined by writing defining axioms in the usual way.

I.4.1 Set-Theoretic Constructions Used in Defining Equations

1. **(Enumeration)** - The expression $\{a_1, \ldots, a_n\}$ denotes the n-element set whose elements are the constants $a_1, \ldots, a_n$.

2. **(Disjoint Union)** - If A and B are sets, then the expression $[t_A: A + t_B: B]$ denotes the disjoint union D of the sets A and B. The tags t_A and t_B are used to denote the injection operations associated with the disjoint union. That is, if $a \in A$ and $b \in B$, then $t_A{:}a$ denotes the image of a, and $t_B{:}b$ the image of b, in D.

3. **(Cartesian Product)** - The expression $[t_A: A \times t_B: B]$ denotes the cartesian product C of the sets A and B. Associated with an element c of C are its projections $c(t_A)$ and $c(t_B)$ onto the sets A and B, respectively. Given $a \in A$ and $b \in B$, then the expression $\langle t_A: a, t_B: b\rangle$ denotes the ordered pair with components a and b. If $c \in C$ and $a \in A$, then the notation $c[a/t_A]$ denotes the element c' of C which is identical to c except that its t_A component has the value a. To reduce clutter in expressions, tags will be omitted from both the disjoint union and cartesian product constructions when this is unlikely to cause confusion as to the intended meaning.

4. **(Function Space)** - If A and B are sets, then the notation $[A \rightarrow B]$ denotes the set of all functions with domain A and range B. We use the usual notation $f(a)$ for the application of f to the argument a, and the notation $f[b/a]$ for the function that is identical to f except that it has value b for argument a.

5. **(Finite Powerset)** - The notation Set[A] denotes the set of all finite subsets of the set A. If $s \in$ Set[A] and $a \in A$, then the expression $a \in s$ is true iff a is an element of the set s. The expression $|s|$ denotes the cardinality of the set s. We also use the usual operations $\cup$, $\cap$, and $\sim$ on Set[A]. The notation MSet[A] denotes the set of all finite

multisets of elements of A. We use the same notation for operations on multisets as for sets, however, the meaning appropriate for multisets is assumed in this case.

6. **(Finite Sequences)** - The notation Seq[A] denotes the set of all finite sequences (i.e. strings) of elements of A. If u, $v \in$ Seq[A], then $|u|$ denotes the length of u, uv and $u \cdot v$ denote the concatenation of u and v, and if $n \in$ Nat, then $u(n)$ denotes the $n+1$st element of u.

I.4.2 Definition of the State-Transition Relation

Manipulation of the state-transition relation is sometimes more convenient if its defining axioms are factored into a collection of pairs, each of which consists of a *precondition*, and a *next-state predicate*. The precondition defines the class of events to which the pair applies, and defines conditions on the current state that must be satisfied before an event in that class can occur. The next-state predicate determines the relation that must hold between the current state and the new state that results from an occurrence of such an event.

Although the basic idea of precondition/next-state predicate pairs is fairly simple, some subtleties arise in actual use, especially associated with the interpretation of free variables common to the two predicates. This problem is similar to that which arises in the interpretation of free variables in the pre- and post-conditions used to specify sequential programs. We must therefore be somewhat more careful about the precise form and meaning of the pairs. A pair takes the form: $\langle \text{Pre}(q, e, \underline{x}), \text{Next}(q, r, \underline{x})\rangle$, where e is a variable of sort *Events*, q and r are variables of sort *States*, and $\underline{x}$ is a vector of free variables of sorts $\underline{S}$, where $\underline{S}$ can be chosen arbitrarily for each pair. A finite collection $\langle \text{Pre}_1, \text{Next}_1\rangle, \ldots, \langle \text{Pre}_n, \text{Next}_n\rangle$, where the kth pair contains free variables $\underline{x}_k$ of sorts $\underline{S}_k$, determines the defining axiom for the state transition relation Trans according to the following definition:

$$\text{Trans}(q, e, r) \equiv \vee_{k=0}^{n}(\exists \underline{x}_k \in \underline{S}_k)(\text{Pre}_k(q, e, \underline{x}_k) \wedge \text{Next}_k(q, r, \underline{x}_k)),$$

where $\text{Pre}_0(q, e, \underline{x}_0) \equiv e = \lambda$ and $\text{Next}_0(q, r, \underline{x}_0) \equiv r = q$. What the above definition says is that a step $\langle q, e, r\rangle$ satisfies the state transition relation Trans iff there exists a pair $\langle \text{Pre}_k, \text{Next}_k\rangle$ $(0 \leq k \leq n)$, and an interpretation of the free variables of that pair, such that the precondition and next state predicate hold for that pair.

A useful convention we will follow, which simplifies the maximality part of correctness proofs, is to define the preconditions Pre(q, e, $\underline{x}$) and next state predicate Next(q, r, $\underline{x}$) in each pair so that they satisfy the following relationship:

$$\models (\forall q \in \text{States}, e \in \text{Events}, \underline{x} \in \underline{S})(\text{Pre}(q, e, \underline{x}) \rightarrow (\exists r \in \text{States})\text{Next}(q, r, \underline{x})).$$

That is, whenever the precondition is satisfied by a state q, an event e, and an interpretation $\underline{x}$ for the free variables, then there must be a new state r such that q, r and $\underline{x}$ satisfy the next state predicate.

I.4.3 Parameterized Descriptions

Quite often one wishes to write parameterized descriptions of event/state algebras, where the parameters may be values, as in the case of the synchronizer component module, where the number of initial tokens is given as a parameter, or perhaps sets or some other kind of object. In this thesis, we view a parameterized event/state algebra description as a schema for the construction of a family of related descriptions. This way of treating parameters is satisfactory as long as there is no need to perform reasoning about parameters with infinitary structure. A more general treatement of parameters requires extensions to the event/state algebra formalism, and is outside the scope of this thesis.

I.5 Implementation Algebras

We have previously discussed the notion of an event/state algebra, which is a formal structure that embeds the machine part of a state-transition specification. The purpose of an event/state algebra is to provide semantics for the associated temporal language. The temporal language, in turn, serves as a vehicle for the formal statement of properties of histories, among which are the validity conditions for a specification. Augmented with a sound deductive system, the temporal language can also serve to express derivations of consequences of a specification.

Just as we can use the temporal language associated with a specification to express and derive consequences of that specification, we would like to associate with an implementation a language suitable for the expression and derivation of the conditions required for the correctness of that implementation. However, taken separately, none of the temporal languages associated with any of the modules involved in an implementation suffices for this purpose. To solve this problem, we define below the notion of an "implementation algebra," which is a kind of "composite" event/state algebra whose associated temporal language is powerful enough to permit the expression of correctness conditions.

Let us say that an algebra A *embeds* an algebra B if there exists a signature morphism[1] ι from the signature of B to the signature of A such that, for each sort S (resp. function symbol f, relation symbol R) of B, the interpretation of S (resp. f, R) in B is the same as the interpretation of $\iota(S)$ (resp. $\iota(f)$, $\iota(R)$) in A. If A embeds B, then since we might as well think of B as a subalgebra of A, we will omit mention of the signature morphism ι when no confusion can arise.

Suppose A_{abs} is an event/state algebra (corresponding to an abstract module to be implemented), and let $\underline{A} = \langle A_1, \ldots, A_n \rangle$ be a finite-length vector of event/state algebras (corresponding to the component modules).

Definition - An *implementation algebra* for A_{abs} and $\underline{A}$ is an event state algebra A with the following properties:

(1) A embeds A_{abs} and each A_i, with $1 \leq i \leq n$. For each sort or operation S of A_{abs} (resp. A_i), we write S^{abs} (resp. S^i) for the corresponding sort or operation of A.

(2) A contains distinguished functions

α: Events $\rightarrow$ Events$^{\text{abs}}$

δ_i: Events $\rightarrow$ Events^i, for $1 \leq i \leq n$

π_{abs}: States $\rightarrow$ States$^{\text{abs}}$

π_i: States $\rightarrow$ States^i, for $1 \leq i \leq n$,

such that: $\mathfrak{I}_A = \langle \alpha, \underline{\delta} \rangle$ is an interconnection, called the *embedded interconnection*, Mach$_A$ is the composite machine for $\mathfrak{I}_A$, Mach$_{A_{\text{abs}}}$ and $\langle \text{Mach}_{A_i} \rangle_{i=1}^{n}$; and π_{abs} and the π_i are the canonical projections from the cartesian product *States* to the factors *States*$^{\text{abs}}$ and *States*^{i}, respectively. ∎

1. A function, mapping each sort, function symbol, and relation symbol of the signature of B to a corresponding sort, function, symbol, or relation symbol of the signature of A, that preserves relevant structure such as the -arity of the symbols.

Since an implementation algebra is a particular kind of event/state algebra, it has an associated temporal language. Furthermore, the temporal language $\mathcal{T}(A)$ associated with an implementation algebra A contains the temporal languages $\mathcal{T}(A_{\mathrm{abs}})$ and each $\mathcal{T}(A_i)$ as sublanguages. This property is what makes an implementation algebra useful for expressing correctness conditions.

The description of an implementation algebra is performed in the same way as for ordinary event/state algebras. The meanings of many the symbols are fixed by the definition of an implementation algebra, and in practice it is convenient to omit their defining axioms. For example, the definition of the sort *States* is fixed by the requirement that it be the cartesian product of the sorts *States*^{k}:

$$\mathrm{States} = [\pi_{\mathrm{abs}}: \mathrm{States}^{\mathrm{abs}} \times \pi_1: \mathrm{States}^1 \times \ldots \times \pi_n: \mathrm{States}^n].$$

Other examples of symbols whose meanings are fixed by the definition of an implementation algebra are the initial state relation Init, and state-transition relation Trans for the composite machine. Definitions must always be explicitly given for the sort *Events*, the abstraction map α and the components δ_i of the decomposition map.

I.6 Proof Techniques

I.6.1 Formal Correctness Theorem

In this section we reduce the problem of proving the correctness of an implementation to the problem of showing the validity of a set of *verification conditions*, which are expressed in the temporal language associated with the implementation algebra. There are three verification conditions in the technique introduced here. The "invariance" verification condition expresses that the predicate *Inv* is an implementation invariant. The "maximality" verification is a straightforward formalization of the the maximality condition required by the Correctness Theorem, except that the phrase "q is reachable for the composite machine" is replaced by "*Inv*(q) holds." The "validity" verification condition is the formalization of the validity condition required by the Correctness Theorem.

Recall that the validity condition required by the Correctness Theorem states that, if X is a computation for the composite machine that projects, under the canonical projections associated with the composite machine, to a valid computation for each component machine, then X projects to a valid computation for the abstract machine as well. This condition cannot be formalized directly as a sentence in the temporal language of the implementation algebra, since that language has no constructs for dealing directly with histories and functions on histories. However, the language does contain the function symbols α, $\langle\delta_i\rangle_{i \in I}$, π_{abs}, and $\langle\pi_i\rangle_{i \in I}$, which denote the abstraction map, components of the decomposition map, and canonical projections on the state set, respectively.

To formalize the validity verification condition, we need some way of taking the sentences that express the conditions required for a computation of the abstract machine or a component machine to be valid, and "lifting" these sentences to sentences that express the corresponding properties on computations of the composite machine. In Chapter 4 we defined a syntactic translation that accomplished this lifting in the case of the synchronizer implementation. We now define this translation in general, and state a lemma that summarizes its useful properties.

Suppose that A is an implementation algebra for A_{abs} and $\langle A_i\rangle_{i \in I}$. Given a formula φ of $\mathcal{T}(A_{\mathrm{abs}})$, define $[\![\varphi]\!]_{\mathrm{abs}}$ to be the formula of $\mathcal{T}(A)$ obtained by replacing each occurrence of the symbol **Now** by the term π_{abs}(**Now**), each occurrence of **After** by the term π_{abs}(**After**), and each occurrence of **Occurs** by the term α(**Occurs**). Similarly, for each $i \in I$, given a formula φ of $\mathcal{T}(A_i)$, define $[\![\varphi]\!]_i$ be the formula of $\mathcal{T}(A)$ obtained by replacing each occurrence of **Now** by π_i(**Now**), each occurrence of **After** by π_i(**After**), and each occurrence of **Occurs** by δ_i(**Occurs**).

The precise relationship between a formula and its translation is captured by Lemma I.2 below. An analogous result is stated in [Wolper82], where process of "lifting" specifications of processes to obtain specifications of a system of processes is called "relativization."

Lemma I.2 (Translation Lemma) - Suppose that A is an implementation algebra for A_{abs} and $\langle A_i\rangle_{i \in I}$. Suppose that $\varphi(v_0 \dots v_m)$ is a formula of $\mathcal{T}(A_{\mathrm{abs}})$ (resp. $\mathcal{T}(A_i)$, for some $i \in I$), that $a_0, \dots, a_m$ is an interpretation of the variables $v_0, \dots, v_m$, and that X is a history over *Events*$_A$ and *States*$_A$. Then

$$X \models_A [\![\varphi]\!]_{\mathrm{abs}}[a_0/v_0 \dots a_m/v_m] \text{ iff } X^{(\mathrm{abs})} \models_{A_{\mathrm{abs}}} \varphi[a_0/v_0 \dots a_m/v_m].$$

(resp. $X \models_A [\![\varphi]\!]_i[a_0/v_0 \dots a_m/v_m]$ iff $X^{(i)} \models_{A_i} \varphi[a_0/v_0 \dots a_m/v_m]$.)

Proof - Straightforward induction on formulas, based on the precise syntax and semantics of $\mathcal{T}(A)$ given above. ∎

In the sequel, to make formulas in the language $\mathcal{T}(A)$ of an implementation algebra A easier to read, we will often abbreviate the application of the functions π_{abs}, and π_k to a variable or constant by simply affixing an appropriate subscript to that variable or constant. Thus, if q is a variable of sort *States*, then q_{abs} and q_k abbreviate $\pi_{abs}(q)$ and $\pi_k(q)$, respectively.

We can now give a formalized version of the Correctness Theorem. Roughly speaking, this result says that to prove the correctness of an implementation defined by an implementation algebra A, it suffices to perform the following three steps:

(1) Determine the *implementation invariant* $Inv(q)$ expressed in the first-order language $\mathcal{L}(A)$ and containing the single free variable q of sort *States*. Show the validity of two sentences of $\mathcal{L}(A)$, which assert that *Inv* is inductive.

(2) Show the validity of a sentence of $\mathcal{L}(A)$ which implies that the maximality condition holds. This sentence is obtained by formalizing the maximality condition of the Correctness Theorem in the obvious way.

(3) Show the validity of a sentence of $\mathcal{T}(A)$ that asserts that the validity condition holds. This sentence is formed from the sentences that describe the sets of valid computations for the abstract and component machines, through the use of the translation operation discussed above.

Lemma I.2 (Formal Correctness Theorem) - Suppose that A is an implementation algebra for A_{abs} and $\langle A_i \rangle_{i \in I}$. Suppose that Valid_{abs} is a sentence of $\mathcal{T}(A_{abs})$, and for each i, Valid_i is a sentence of $\mathcal{T}(A_i)$. Let S_{abs} be the state-transition specification defined by the pair $\langle A_{abs}, \text{Valid}_{abs} \rangle$, and for each i, let S_i be the state-transition specification defined by the pair $\langle A_i, \text{Valid}_i \rangle$. Suppose that $\text{Inv}(q)$ is a formula of $\mathcal{L}(A)$, with one free variable q of sort *States*, such that the *verification conditions* below hold. Then $\langle \mathcal{I}_A, S_{abs}, \underline{S} \rangle$ is correct.

(Invariance):

(Basis) $\models (\forall q \in \text{States})(\text{Init}(q) \rightarrow \text{Inv}(q))$

(Induction) $\models (\forall q, r \in \text{States}, e \in \text{Events})(\text{Trans}(q, e, r) \rightarrow (\text{Inv}(q) \rightarrow \text{Inv}(r)))$

(Maximality):

$\models (\forall q \in \text{States}, e \in \text{Events})((\text{Inv}(q) \wedge \bigwedge_{i \in I} \text{Enabled}_i(q, e)) \rightarrow \text{Enabled}_{abs}(q, e))$.

(Validity):

$$\text{Comp} \models (\wedge_{i \in I} [\![\text{Valid}_i]\!]_i) \rightarrow [\![\text{Valid}_{\text{abs}}]\!]_{\text{abs}}.$$

where

$$\text{Enabled}_{\text{abs}}(q, e) \equiv (\exists r \in \text{States})\text{Trans}^{\text{abs}}(q_{\text{abs}}, \alpha(e), r_{\text{abs}})$$
$$\text{Enabled}_i(q, e) \equiv (\exists r \in \text{States})\text{Trans}^i(q_i, \delta_i(e), r_i).$$

Proof - The basis part of the invariance verification condition states that Inv is true for all initial states, and the induction part of the invariance verification condition states that Inv is preserved under state transitions, and hence the truth of these two conditions implies that *Inv* is inductive.

From the definition of the predicates $\text{Enabled}_{\text{abs}}$ and Enabled_i, we know that $\text{Enabled}_{\text{abs}}(q, e)$ is true of a state q and event e iff $\alpha(e)$ is enabled for $\text{Mach}_{A_{\text{abs}}}$ in state q_{abs}, and similarly, $\text{Enabled}_i(q, e)$ is true iff $\delta_i(e)$ is enabled for Mach_{A_i} in state q_i. The maximality verification condition therefore says that whenever q is a state such that $\text{Inv}(q)$ holds, and $\delta_i(e)$ is enabled for Mach_{A_i} in state q_i for each i with $1 \leq i \leq n$, then $\alpha(e)$ is enabled for $\text{Mach}_{A_{\text{abs}}}$ in state q_{abs}. This implies the maximality condition required by the Correctness Theorem.

By the Translation Lemma, we know that $[\![\text{Valid}_{\text{abs}}]\!]_{\text{abs}}$ is satisfied by a computation X of Mach_A iff $\text{Valid}_{\text{abs}}$ is satisfied by the computation $X^{(\text{abs})}$ of $\text{Mach}_{A_{\text{abs}}}$. Similarly, for each i we know that $[\![\text{Valid}_i]\!]_i$ is satisfied by a computation X of Mach_A iff Valid_i is satisfied by the computation $X^{(i)}$ of Mach_{A_i}. Since a history X satisfies Comp iff X is a computation of Mach_A, we see that the validity verification condition is the formal statement of the validity condition required by the Correctness Theorem.

Since the truth of the verification conditions above implies that the hypotheses of the Correctness Theorem are satisfied, an application of the Correctness Theorem shows the correctness of the implementation $\langle \mathcal{I}_A, S_{\text{abs}}, \underline{S} \rangle$. ∎

I.7 Rely-/Guarantee-Condition Proof Techniques

In this section we give the formalized versions of the rely-/guarantee-condition proof techniques stated in Chapter 3. The first result formalizes Lemma 3.11.

Corollary I.4 (Formal Rely/Guarantee Technique I) - Suppose that A is an implementation algebra for A_{abs} and $\langle A_i \rangle_{i \in I}$. Suppose that $\text{Valid}_{\text{abs}} \equiv \text{Rely}_{\text{abs}} \rightarrow \text{Guar}_{\text{abs}}$

is a sentence of $\mathfrak{T}(A_{abs})$ and that $\mathrm{Valid}_i = \mathrm{Rely}_i \rightarrow \mathrm{Guar}_i$ for each $i \in I$ is a sentence of $\mathfrak{T}(A_i)$. Suppose that

(1) $\mathrm{Comp} \models (\wedge_{i \in I} [\![\mathrm{Guar}_i]\!]_i) \rightarrow [\![\mathrm{Guar}_{abs}]\!]_{abs}$, and

(2) There exists a well-founded partial order $<$ on I such that for all $i \in I$,

$$\mathrm{Comp} \models [\![\mathrm{Rely}_{abs}]\!]_{abs} \wedge (\wedge_{j<i} [\![\mathrm{Guar}_j]\!]_j) \rightarrow [\![\mathrm{Rely}_i]\!]_i.$$

Then $\mathrm{Comp} \models (\wedge_{i \in I} [\![\mathrm{Valid}_i]\!]_i) \rightarrow [\![\mathrm{Valid}_{abs}]\!]_{abs}$.

Proof - Straightforward from Lemma 3.11. ∎

The next result formalizes Lemma 3.12.

Corollary I.5 (Formal Rely/Guarantee Technique II) - Suppose that A is an implementation algebra for A_{abs} and $\langle A_i \rangle_{i \in I}$. Suppose that $\mathrm{Valid}_{abs} \equiv \mathrm{Rely}_{abs} \rightarrow \mathrm{Guar}_{abs}$ is a sentence of $\mathfrak{T}(A_{abs})$ and that $\mathrm{Valid}_i = \mathrm{Rely}_i \rightarrow \mathrm{Guar}_i$ for each $i \in I$ is a sentence of $\mathfrak{T}(A_i)$. Suppose that for each $i, j \in I \cup \{abs\}$, we have determined a sentence $\mathrm{RG}_{i,j}$ of $\mathfrak{T}(A)$, such that properties (1)-(3) below hold.

(1)(a) $\mathrm{Comp} \models [\![\mathrm{Rely}_{abs}]\!]_{abs} \rightarrow \wedge_{j \in I} \mathrm{RG}_{abs,j}$

(b) $\mathrm{Comp} \models \wedge_{i \in I} \mathrm{RG}_{i,abs} \rightarrow [\![\mathrm{Guar}_{abs}]\!]_{abs}$

(2)(a) $\mathrm{Comp} \models \mathrm{RG}_{abs,j} \wedge \wedge_{i \in I+\{abs\}} \mathrm{RG}_{i,j} \rightarrow [\![\mathrm{Rely}_j]\!]_j$, for all $j \in I$

(b) $\mathrm{Comp} \models [\![\mathrm{Guar}_i]\!]_i \rightarrow \mathrm{RG}_{i,abs} \wedge \wedge_{j \in I+\{abs\}} \mathrm{RG}_{i,j}$, for all $i \in I$

(3) (Acyclicity) - Whenever $\{\langle i_1, i_2 \rangle, \langle i_2, i_2 \rangle, \ldots, \langle i_n, i_{n+1} \rangle\}$ is a cycle of I, then

$$\mathrm{Comp} \models \vee_{k=1}^{n-1} \mathrm{RG}_{i_k, i_{k+1}}.$$

Then $\mathrm{Comp} \models (\wedge_{i \in I} [\![\mathrm{Valid}_i]\!]_i) \rightarrow [\![\mathrm{Valid}_{abs}]\!]_{abs}$.

Proof - Straightforward from Lemma 3.12. ∎

I.8 I/O-Consistency Proof Technique

The result below formalizes the technique for proving I/O-consistency expressed by Corollary 5.8.

Corollary I.6 - Suppose that S is the state-transition specification of I/O-interface E defined by the pair $\langle A, \mathrm{Valid} \rangle$, where the sets of inputs and outputs of E are defined by the unary relations *In* and *Out* of type *Events* in A. Suppose that the event/state algebra A includes among its operations the finite collection of relations $\langle \mathrm{Prod}_i \rangle_{i \in I}$, where Prod_i is of type *States* × *Events* × *States*. If the following sentences of $\mathfrak{T}(A)$ are valid, then S is

$\mathfrak{I}_2$-consistent.

(1) $\models \bigwedge_{i \in I} (\forall q, r \in \text{States}, e \in \text{Events})(\text{Prod}_i(q, e, r) \rightarrow \text{Trans}(q, e, r))$

(2) $\models (\forall q \in \text{States}, e \in \text{Events})(\text{In}(e) \rightarrow (\exists r \in \text{States})\text{Trans}(q, e, r))$

(3) $\models (\forall q, r \in \text{States}, e \in \text{Events})(\text{Trans}(q, e, r) \wedge (\text{Out}(e) \vee e = \lambda) \rightarrow$

$\bigvee_{i \in I} \text{Prod}_i(q, e, r))$

(4) $\text{Comp} \models (\bigwedge_{i \in I} \text{Fair}_i) \rightarrow \text{Valid}$,

where

$\text{Fair}_i \equiv \Box\Diamond\text{Enabled}_i(\textbf{Now}) \rightarrow \Box\Diamond\text{Prod}_i(\textbf{Now}, \textbf{Occurs}, \textbf{After})$.

$\text{Enabled}_i(q) \equiv (\exists r \in \text{States}, e \in \text{Events})\ \text{Prod}_i(q, e, r)$

Proof - Straightforward from Corollary 5.8. Hypothesis (1) says that the Prod_i are subsets of Trans. Hypotheses (2) states that Mach_A is input-cooperative. Hypothesis (3) states that the Prod_k cover the set of nonnull output or λ-steps in Trans. Hypothesis (4) formalizes the requirement that every fair computation of Mach_A is valid. ∎

Appendix II - Additional Examples

In this appendix the specification and verification techniques introduced in the thesis will be further illustrated through two additional examples. The first example concerns the specification and implementation of a resource manager module whose function is to allocate resources in response to requests from user processes. The resource manager is implemented in a highly distributed fashion by a tree-structured system of local resource manager modules that communicate with each other to determine where resources should be sent. In the second example, a reliable message transmission service is specified, and an implementation by an unreliable message transmission substrate is given. Reliability is achieved through the use of a fault-tolerant protocol: the *alternating bit protocol* [Bartlett69]. The alternating bit protocol example has been examined by several other researchers [Chen82, Hailpern80, Lamport83, Schwartz81], and has become somewhat of a standard for evaluating specification and verification techniques for concurrent systems.

The major purpose of the additional examples given here is to lend support to the following assertion: Essentially the same techniques as were used to obtain specifications and a correctness proof for the synchronizer implementation, can be applied in a reasonably systematic way to achieve similar results on other nontrivial examples. Thus, the ideas of state-transition specification, rely- and guarantee-conditions, and the proof technique embodied in the Correctness Theorem, are not *ad hoc* concepts useful for a single example, but serve as generally applicable guiding principles.

A second point illustrated by the examples of this chapter is that more elegant specifications can result if one first imagines the structure of a proof of correctness in which the specifications will be used, and then derives the module specifications in an attempt to satisfy the requirements imposed by the proof structure. The difference between specifications obtained via this approach and those resulting from the "specify first, prove later" approach can be seen by comparing the validity conditions given here for the send and receive protocol modules with the liveness properties given by Lamport [Lamport83] for these modules. The specifications and proof given below are to a large extent independent of the precise assumptions on the behavior of the unreliable transmission medium. Lamport's presentation does not make this independence quite so explicit.

The observation that a proof of correctness can be used to derive component module specifications suggests the following general method for designing a correct implementation of a given abstract module:

(1) Decide on the communication structure of the system of component modules (e.g. tree or ring structure).

(2) For each pair of component modules that can possibly communicate, express informally the properties that each relies on/guarantees to the other to provide. These rely- and guarantee-conditions will serve to "cut" the interdependence of the component modules in a fashion similar to the way in which a loop invariant cuts the dependence of one iteration on preceding and succeeding iterations.

(3) Select event and state sets for the component modules in such a way that the temporal language of the resulting implementation algebra is powerful enough to formally express the informally stated rely- and guarantee-conditions.

(4) "Localize" the rely- and guarantee-conditions so that they are expressed in the temporal language of each component module event/state algebra. The rely- and guarantee-conditions of a resulting component module specification will be the conjunction of the localized rely- and guarantee-conditions, respectively.

The examples in this appendix will be presented using the notation of Appendix I.

II.9 A Distributed Resource Management Algorithm

In this section, we consider the specification and implementation of a *resource manager* module RM, whose function is to allocate *resources* to a set of *clients* in response to requests from those clients. We will see how the resource manager can be implemented by a tree-structured network of *local resource manager* (LRM) modules, each of which communicates with a single client. Initially each local resource manager starts out with some subset of the resources. As client requests arrive and are filled at a particular site, though, the locally available set of resources might be exhausted. An LRM that is deficient in resources must then attempt to obtain additional resources from other sites. The interesting part of the implementation is concerned with how the local resource managers communicate with each other to determine where the resources should be sent. The strategy by which this is accomplished is essentially the "DYNAMIC-MATCH" strategy of [Fischer83], although this stategy is explained here in a slightly different and hopefully simpler way than in that paper.

The resource manager example is presented here as a nontrivial exercise in the use of rely-/guarantee-conditions and an associated correctness argument as a basis for the derivation of specifications for the local resource manager modules. The use of rely-/guarantee-conditions as a guiding principle permits us to derive, in a reasonably systematic fashion, essentially the same specification for the local resource manager module as the node algorithm presented in [Fischer83]. The primary difference between the *specification* derived here and the *algorithm* of [Fischer83] is that we are not concerned here with the way in which an LRM resolves choices as to the pattern in which excess requests are forwarded to its neighbors. In [Fischer83], it is assumed that choices are resolved according to a specific probability distribution, and a large portion of the paper is concerned with probabilistic analysis of the consequences of this assumption. Here we concern ourselves only with showing that every request from a client is eventually satisfied, if possible. The argument provided in [Fischer83] of this basic correctness property is more of a proof sketch than a proof, and is somewhat unsatisfactory for this reason.

II.9.1 Specification of the Resource Manager Module

The function of the resource manager module RM can be described as follows: Let Clients be a set that contains the names of the clients with which the resource manager communicates, and let Resources be a set that contains the names of the resources to be managed. A client *c* requests a resource from the resource manager by issuing a *request event request:c*. The resource manager allocates a resource *r* to client *c* by issuing a *reply event reply:<c, r>*. In this example, a resource that has been allocated to a client is never returned to the resource manager.

The state of the resource manager can be thought of as consisting of a pair <pending, free>, where *pending* is a multiset of clients that represents the collection of unfilled requests and *free* is the set of available resources. The *pending* component is a multiset since we permit more than one request from a single client to be outstanding at one time. Receipt of a request from client *c* by the resource manager causes an instance of *c* to be added to the pending multiset. The event *reply:<c, r>* can occur only if the client *c* is in the pending multiset and the resource *r* is in the free set. Occurrence of this event causes an instance of *c* to be removed from the pending set and the resource r to be removed from the free set. It is clear from this description that no resource is allocated more than once and no more than one resource is allocated in response to each request. In addition, we would like the resource manager to respond eventually to every request, as long as the set of free resources has not been exhausted.

To derive a more precise specification from the preceding informal description, we begin by defining the resource manager event/state algebra. Our description has the following as parameters:

Clients:	a finite set of clients
Resources:	a finite set of resources

The interface of the resource manager is defined as follows:

$$\mathrm{Events}^{RM} = \{\lambda\} + [\text{request: Clients} + \text{reply: (Clients} \times \text{Resources)}].$$
$$\mathrm{In}^{RM} = \{\lambda\} + [\text{request: Clients}]$$
$$\mathrm{Out}^{RM} = \{\lambda\} + [\text{reply: (Clients} \times \text{Resources)}]$$

The state set for the resource manager is defined by:

$$\mathrm{States}^{RM} = [\text{free: Set[Resources]} \times \text{pending: MSet[Clients]}].$$

In an initial state, the multiset of pending requests is empty, and all resources are free.

$$\mathrm{Init}^{RM}(q) \equiv q(\text{free}) = \text{Resources} \wedge q(\text{pending}) = \emptyset.$$

The state-transition relation Trans^{RM} is defined by precondition/next-state predicate pairs as follows:

A request event for client c can occur at any time, and causes c to be added to the pending set.

(request) $\quad \mathrm{Pre}_{request}(q, e, c) \equiv e = \text{request:}c$

$\qquad\qquad \mathrm{Next}_{request}(q, r, c) \equiv r = q[(q(\text{pending}) \cup \{c\})/\text{pending}]$

A reply event with resource *res* for client c can occur only if *res* is in the free set and c is in the pending multiset. It causes *res* to be removed from the free set and an instance of c to be removed from the pending multiset.

(reply) $\quad \mathrm{Pre}_{reply}(q, e, c, res) \equiv e = \text{reply:}\langle c, res\rangle \wedge c \in q(\text{pending}) \wedge res \in q(\text{free})$

$\qquad\qquad \mathrm{Next}_{reply}(q, r, c, res) \equiv r = q[(q(\text{pending}) - \{c\})/\text{pending}, (q(\text{free}) - \{res\})/\text{free}]$

The validity conditions for the resource manager module can be stated in rely-/guarantee-condition form as follows: $\text{Valid}^{RM} \equiv \text{Rely}^{RM} \rightarrow \text{Guar}^{RM}$, where

$$\text{Rely}^{RM} \equiv \Box(|\mathbf{Now}(\text{free})| \geq |\mathbf{Now}(\text{pending})|)$$

$$\text{Guar}^{RM} \equiv \Box(\forall c \in \text{Clients})(c \in \mathbf{Now}(\text{pending}) \rightarrow \Diamond(\exists r \in \text{Resources})(\mathbf{Occurs} = \text{reply:}\langle c, r\rangle)).$$

Thus, if the number of outstanding requests never exceeds the number of available resources, then the resource manager module guarantees that every request will eventually receive a reply.

II.9.2 Implementation of the Resource Manager

Our plan is to implement the resource manager module by a tree-structured network of local resource manager modules as depicted in Figure 3. Each local resource manager is responsible for filling requests originating from a single client. If the set of resources locally available is exhausted, the the LRM must try to obtain additional resources from elsewhere in the system. If an LRM has a surplus of resources, then it must be willing to give out resources to other LRM's whose resources have already been allocated.

To guide us in our derivation of the components that will be needed as part of an LRM state, let us first obtain a rough statement of the validity conditions that an LRM is to satisfy. We organize these conditions into properties the LRM relies on its environment to provide, and properties that an LRM guarantees to its environment in return. An LRM relies on:

(1) No special properties on the part of the client.

(2) The eventual elimination of resource debts owed to the LRM by its parent.

(3) The eventual elimination of resource debts owed to the LRM by each of its children.

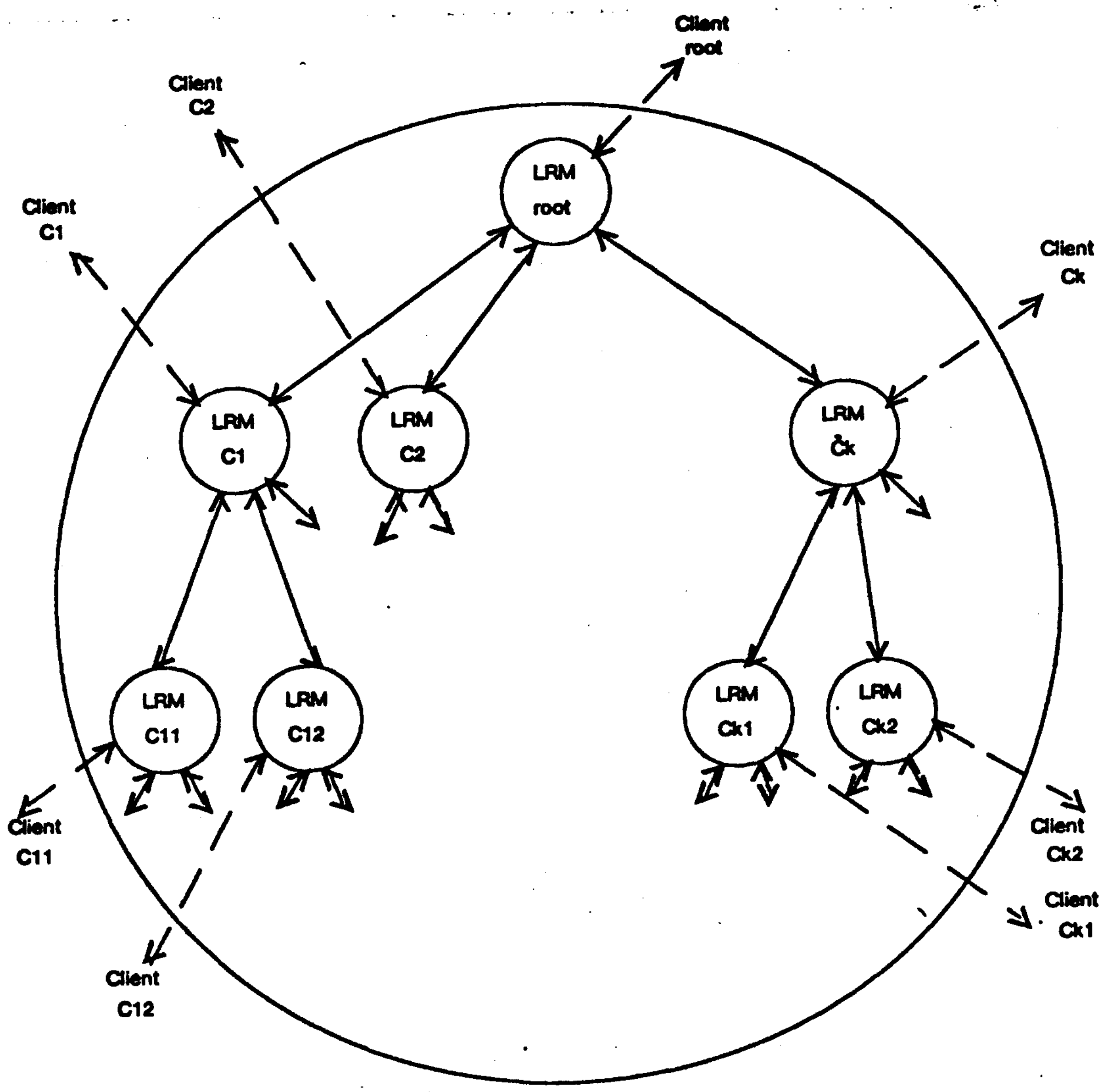
In return for these properties, an LRM guarantees that:

(1) Every client request eventually receives a reply.

(2) Resource debts owed by the LRM to its parent will eventually be eliminated.

(3) Resource debts owed by the LRM to each of its children will eventually be eliminated.

Fig. 3. Resource Manager Implementation



Resource Manager Module

To obtain formal statements of the preceding conditions, we must first obtain a precise definition of the notion of an LRM having a "resource debt" to one of its neighbors, and we must describe the mechanics of how such debts are incurred and eliminated. The introduction of the various components of the LRM state below can be viewed as providing us with enough expressive power in the language $\mathfrak{T}(A^{LRM})$ of the LRM event/state algebra, to permit the formalization of the undefined quantities in the above statement of the LRM validity conditions.

A significant feature of the validity conditions stated above is the complementary form of the rely- and guarantee-conditions. The conditions above have been selected in such a way that ultimately, in the resource manager implementation, the conditions relied upon by an LRM *i* from its neighboring LRM *j* will be precisely the conditions that LRM *j* guarantees to provide to LRM *i*. This symmetric statement of the validity conditions will be seen below to result in a rather simple and pleasant proof of correctness.

With the above validity conditions in mind, we now attempt to identify the various events of the LRM interface and the components of the LRM state. We can identify immediately several kinds of events that must be in the interface of the LRM. Communication with the client requires the existence of a request event *request*, which represents the receipt of a request from the client, and a reply event of the form *reply:r*, in which resource *r* is allocated to the client in response to a prior request. Furthermore, the interface of an LRM must contain events corresponding to the transfer of resources between an LRM and its neighbors in the system. Let Resources be the set of names of all the resources that the LRM might be called upon to handle. For each $r \in$ Resources, the LRM interface includes the event *parent_in:r*, which represents the receipt of resource *r* from an LRM's parent in the tree, and *parent_out:r*, which represents the delivery of resource *r* by an LRM to its parent. Let Children be a set of names used to index the children of the LRM. For each $c \in$ Children and $r \in$ Resources the interface of the LRM includes the event *child_out:*$\langle c, r\rangle$, which represents the transfer of resource *r* from the LRM to child *c*, and the event *child_in:*$\langle c, r\rangle$, which represents the receipt of resource *r* by the LRM from child *c*.

To describe the conditions under which transmission of resources between LRM's and between a client and an LRM is permitted, we include in the state of each LRM a set *free*, which represents the resources locally available at the LRM, and a nonnegative integer *pending*, which counts the number of unfilled requests that originated at the client associated with the LRM. A *request* event causes *pending* to be incremented. A *reply:r* event can occur only if *pending* is nonzero and $r \in$ *free*, and causes *pending* to be decremented and *r* to be removed from *free*. The resource transmission events *parent_in:r* and *child_in:r* cause *r* to be added to the set *free*. The events *parent_out:r* and *child_out:*$\langle c, r\rangle$ can occur only if $r \in$ *free*, and cause *r* to be removed from *free*.

We have thus settled the issue of how and when requests and replies are transmitted betweeen an LRM and its client, and how resources are shuttled between LRM's. However, we have not yet determined how and when an LRM should request resources from one of its neighbors, or when an LRM should issue resources to a neighboring LRM. To describe the conditions governing the transmission of resources between LRM's, we introduce a few more components into the state of an LRM. The state of each LRM contains a component *p_balance*, and a component *c_balance:c* for each child *c*. The component *p_balance* represents the instantaneous "balance of payments" between the LRM and its parent, and *c_balance:c* represents a similar balance of payments between the LRM and child *c*. A positive balance represents a number of resources owed to the LRM by its neighbor, and a negative balance represents a number of resources owed by the LRM to its neighbor. These balances will be maintained so that the following relation is invariant: If *p* is an LRM with child *c*, then the *c_balance:c* component of the state of LRM *p* is always the negative of the *p_balance* component of the state of LRM *c*. This reflects the idea that resources owed by *p* to *c* can be viewed as a debit from the point of view of *p*, or as a credit from the point of view of *c*. These balances will be updated appropriately as requests are forwarded, and as resources travel between LRM's in payment of debts. An LRM will transmit resources to its neighbor in an attempt to reduce its indebtedness.

To represent the forwarding of requests between LRM's we introduce the following additional kinds of events into the LRM interface: A *forward_in* event represents the receipt by the LRM of a forwarded request from its parent. Similarly, a *forward_out:c* event corresponds to the forwarding of a request by the LRM to child *c*. The event *reject_out* represents the forwarding of a request by the LRM to its parent, and the event *reject_in:c* represents the receipt of a forwarded request by the LRM from child *c*. We use the terminology *reject* for the forwarding of requests upward in the tree to emphasize the asymmetry inherent in the parent/child relationship.

In determining the conditions under which forwarding and rejection events should be permitted to occur, we must attempt to avoid the following two bad situations: (1) We must avoid the deadlock situation in which two LRM's are stubbornly requesting resources from each other, while each of their resource requirements could be fulfilled by resources from elsewhere in the system. (2) We must avoid the "livelock" situation in which a request is continually shuttled back and forth in the system without ever reaching an LRM with available resources. Our proposal for resolving these difficulties is to have each LRM keep estimates of the number of surplus resources available in the subtree headed by each of its children. These estimates are to be *optimistic* in the sense that the estimate held by an LRM for child *c* is at all times an upper bound on the number of surplus resources actually available in the subtree headed by *c*. Situation (1) is avoided by having an LRM request resources from its parent only in the case that it has no resources locally available and there are no surplus resources left in any of the subtrees headed by its children. Situation (2) is avoided by requiring that an LRM only send a request to a child *c* if it estimates that there is a surplus of resources in the subtree headed by *c*. The effect of these two requirements is to ensure that the following invariant holds: If an LRM *p* owes resources to its child LRM *c*, then the number of resources owed by *p* to *c* is a lower bound on the instantaneous amount by which pending requests exceed available resources in the subtree headed by *c*. Thus *p* never owes more resources to *c* than are actually required by *c*'s subtree.

The balances of payments between an LRM and each of its neighbors can be combined with the number of pending requests and locally available resources to produce a quantity *PBalance*, which represents the projected net number of resources (positive = surplus, negative = deficit) that would be left at the LRM after all debts are paid. The quantity *PBalance*, defined formally below, is informally the number of free resources, plus the net number of resources owed to the LRM by its neighbors, minus the number of pending requests. The forwarding and rejection of requests by an LRM to its neighbors is done with the goal of "getting in the black;" that is, reducing the projected deficit.

The remaining components we need as part of the LRM state are the following: For each child *c*, the state of an LRM contains a component *c_estim*(*c*) which is an integer that represents the optimistic estimate made by the LRM, of the projected number of resources that would be available in the subtree headed by child *c*, once all debts have been paid. If *c* is an LRM whose parent is *p*, then the state of *c* also contains a component *p_estim*, which is a local copy of the *c_estim*(*c*) component of the state of LRM *p*. Thus, not only does an LRM keep estimates of the projected number of resources remaining in the subtrees headed by each of its children, but it also keeps track of what its parent must currently estimate as the projected number of resources remaining in the subtree headed by the LRM. We permit *p_estim* and *c_estim*(*c*) to take on arbitrary integer values, although it can be shown that if an LRM is used only in a system of other LRM's in the way we envision, then *p_estim* and *c_estim*(*c*) are invariantly nonnegative.

The important points of the preceding discussion of the LRM events and states can be summarized as follows:

(1) The LRM interface contains events corresponding to requests from and replies to the client, transferring of resources from/to its neighbors, and forwarding and rejection of requests.

(2) An LRM state contains a set *free* of locally available resources and a count *pending* of outstanding requests from the client, to ensure that every request receives a response and that no resource is allocated more than once.

(3) An LRM state contains a record of its "balance of payments" with each of its neighbors. Transfer of resources and requests between LRM's is performed to reduce indebtedness. If *p* and *c* are neighboring LRM's, then the balance kept by *p* for *c* is the negative of the balance kept by *c* for *p*.

(4) An LRM state contains an estimate of the projected net number of resources that would remain, once all debts have been paid, in the subtrees headed by each of its children. This information is used to control the forwarding and rejection of requests. If *p* is the parent of *c*, then *c* maintains a local copy of *p*'s estimate of the projected number of resources remaining in the subtree headed by *c*.

II.9.3 Local Resource Manager Specification

From the informal discussion of the preceding section, we can derive a precise local resource manager specification. In the informal discussion above, we made no distinction between the root LRM and the other LRM's in the system. Although similar in many respects, the precise specifications of these two kinds of LRM's will be slightly different since a root LRM has no parent. To avoid redundancy, the specifications of the two kinds of LRM will be presented simultaneously, with differences pointed out along the way.

The parameters of the LRM are the following:

Children:	a finite set of children
Resources:	a finite set of resources
IResources:	the subset of Resources held initially by the LRM

$\{estim_c : c \in \text{Children}\}$:	initial estimates of the number of resources in the subtrees headed by each of the children.

The set Children is a set of names used to identify the children of the LRM. The set Resources is a set of names for all of the resources that the LRM might have to deal with. This set includes the names of all resources initially held by the LRM, as well as all resources that might be transmitted to the LRM at some later instant by its neighbors. The set IResources is a subset of Resources that represents the set of resources initially available at the LRM. For each $c \in$ Children, the parameter $estim_c$ is a nonnegative number which the LRM uses as its initial estimate of the projected number of resources remaining in the subtree headed by child c. Since there will be no debts in an initial state, correct use of an LRM requires that each $estim_c$ equal the actual number of resources initially available in the subtree headed by child c.

The interface of a node LRM is defined as follows:

$$\text{Events}^{NLRM} = \{\lambda\} + [\text{CEvent} + \text{SEvent}]$$
$$\text{In}^{NLHM} = \{\lambda\} + [\text{CIEvent} + \text{SIEvent}]$$
$$\text{Out}^{NRLM} = \{\lambda\} + [\text{COEvent} + \text{SOEvent}],$$

where

$$\text{CEvent} = \text{CIEvent} + \text{COEvent}$$
$$\text{SEvent} = \text{SIEvent} + \text{SOEvent}$$

and

```
CIEvent  = {request}
COEvent  = [reply: Resource]
SIEvent  = [reject_in:          Children +
            forward_in          +
            parent_in:          Resource +
            child_in:           [Children × Resource]]
SOEvent  = [reject_out:   +
            forward_out: Children +
            parent_out:  Resource +
            child_out:   [Children × Resource]]
```

The events listed above have the following intuitive meanings: *Client* events are those in which the LRM communicates with the client, whereas *system* events are those in which the LRM communicates with other LRM's. The client events are classified into *request* events, in which a request is received from the client, and *reply* events, in which a resource is sent to the client in response to a prior request. The system events are classified into: *forwarding* events (forward_out, forward_in), in which a request is forwarded from an LRM to one of its children; *rejection* events (reject_out, reject_in), in which a request is rejected from an LRM to its parent; and *resource transfer* events (parent_out, parent_in, child_out, child_in), in which a resource is transferred from an LRM to one of its neighbors. The "_in" and "_out" suffixes denote the direction in which resources or requests flow; thus, *forward_out:c* is the event in which a request is forwarded from an LRM to child *c*, whereas forward_in is the event in which a forwarded request is received by an LRM from its parent.

The interface $\text{Events}^{\text{RLRM}}$ of a root LRM is obtained by omitting the forward_in, parent_out, reject_out, and parent_in events.

The state set for both a node and a root LRM is defined as follows:

$$
\begin{aligned}
\text{States}^{\text{LRM}} = [&\text{free:} && \text{Set[Resource]},\\
&\text{pending:} && \text{Nat},\\
&\text{p\_balance:} && \text{Int},\\
&\text{c\_balance:} && [\text{Children} \rightarrow \text{Int}],\\
&\text{p\_estim:} && \text{Int},\\
&\text{c\_estim:} && [\text{Children} \rightarrow \text{Int}]].
\end{aligned}
$$

The set *free* is the set of resources currently available at the LRM. The number *pending* is a counter that records the number of outstanding requests. The quantity *p_balance* records the net number of resources that the LRM either is promised by its parent, or promises to send to its parent. If *p_balance* > 0, then the LRM is promised resources by its parent; if *p_balance* < 0, then the LRM promises to send resources to its parent. The mapping *c_balance* records similar information for each of the children. The mapping *c_estim* records the estimate of the projected number of remaining resources in the subtree headed by each child. The quantity *p_estim* is the LRM's local copy of its parent's estimate for the subtree headed by the LRM, as discussed above.

The initial state relation for the LRM is defined below. Recall that we view a finite multiset over a given universe as a function that assigns a finite multiplicity to each element of the universe. Lambda-notation has been used below as a shorthand for denoting particular multisets.

$$\begin{aligned} \mathrm{Init}^{\mathrm{LRM}}(q) \equiv q = \langle \text{free:}\quad & \mathrm{IResources}, \\ \text{pending:}\quad & 0, \\ \text{p\_balance:}\quad & 0, \\ \text{c\_balance:}\quad & (\lambda c \in \mathrm{Children})(0) \\ \text{p\_estim:}\quad & |\mathrm{IResources}| + \Sigma_{c \in \mathrm{Children}}\, \mathrm{estim}_c, \\ \text{c\_estim:}\quad & (\lambda c \in \mathrm{Children})(\mathrm{estim}_c) \rangle \end{aligned}$$

Thus, in the initial state, all resources in IResources are free, no requests are pending, no resources are promised by/promised to any of the neighbors, and the estimated surplus of resources in the subtree headed by the LRM is the sum of the number of free resources initially at the LRM, plus the sum of all the initial estimates for the subtrees headed by each of the children of the LRM.

We can now give the formal definition of the quantity *PBalance* discussed above.

$$\begin{aligned} \mathrm{PBalance}(q) = {} & |q(\text{free})| - q(\text{pending}) + q(\text{p\_balance}) + \\ & \Sigma_{c \in \mathrm{Children}}\, q(\text{c\_balance})(c). \end{aligned}$$

As discussed above, given a state q, PBalance(q) represents the net number of resources (positive = surplus, negative = deficit) that would be left at the LRM after all debts are paid.

The state-transition relation $\mathrm{Trans}^{\mathrm{NLRM}}$ for a node LRM is defined as follows:

An incoming request from a client gets recorded as pending.

(request) $\quad \mathrm{Pre}_{\mathrm{request}}(q, e) \equiv e = \mathrm{request}$

$\qquad\qquad \mathrm{Next}_{\mathrm{request}}(q, r) \equiv r = q[(q(\text{pending}) + 1)/\text{pending}]$

A resource *res* can be sent to the client if there is at least one pending request, and *res* is in the set of free resources. The resource *res* is removed from the set of free resources, and the number of pending requests is decremented.

(reply) $\quad \mathrm{Pre}_{\mathrm{reply}}(q, e, res) \equiv e = \mathrm{reply}{:}res \wedge res \in q(\text{free}) \wedge q(\text{pending}) > 0$

$$\mathrm{Next}_{\mathrm{reply}}(q, r, res) \equiv r = q[(q(\mathrm{free})-\{res\})/\mathrm{free}, (q(\mathrm{pending})-1)/\mathrm{pending}]$$

Receipt of a forwarded request from the parent means that the LRM promises to send one more resource to the parent, and consequently, that the LRM estimates a surplus of one fewer in its own subtree.

(forward_in) $\mathrm{Pre}_{\mathrm{forward\_in}}(q, e) \equiv e = \mathrm{forward\_in}$

$$\mathrm{Next}_{\mathrm{forward\_in}}(q, r) \equiv r = q[(q(\mathrm{p\_balance})-1)/\mathrm{p\_balance}, (q(\mathrm{p\_estim})-1)/\mathrm{p\_estim}]$$

A request can be forwarded to child c only if the LRM currently is "in the red" and estimates a surplus of resources in the subtree headed by child c. As a result of forwarding the request, the number of resources promised by child c is incremented, and the estimated number of resources in the subtree headed by c must be decremented.

(forward_out) $\mathrm{Pre}_{\mathrm{forward\_out}}(q, e, c) \equiv e = \mathrm{forward\_out}{:}c \wedge \mathrm{PBalance}(q) < 0 \wedge q(\mathrm{c\_estim})(c) > 0$

$$\mathrm{Next}_{\mathrm{forward\_out}}(q, r, c) \equiv r = q[(q(\mathrm{c\_balance})(c)+1)/\mathrm{c\_balance}(c), (q(\mathrm{c\_estim})(c)-1)/\mathrm{c\_estim}(c)]$$

Receipt of a rejected request from child c means that child c promises to send one fewer resource (or requires one more resource) than it did before, and thus the quantity *c_balance*(c) must be decremented. In addition, the fact that a request has been rejected by c means that the resources in the subtree headed by c have been exhausted, and thus *c_estim*(c) should be set to zero.

(reject_in) $\mathrm{Pre}_{\mathrm{reject\_in}}(q, e, c) \equiv e = \mathrm{reject\_in}{:}c$

$$\mathrm{Next}_{\mathrm{reject\_in}}(q, r, c) \equiv r = q[(q(\mathrm{c\_balance})(c)-1)/\mathrm{c\_balance}(c), 0/\mathrm{c\_estim}(c)]$$

A request can be rejected to the parent only if the LRM is "in the red" and there is no projected surplus in any of the subtrees headed by children of the LRM. By rejecting a request, the LRM promises one fewer resource to its parent, and hence reduces its projected deficit. In addition, *p_estim* must be zeroed to maintain the invariant equality between *p_estim* and the corresponding *c_estim* component of the parent LRM.

(reject_out) $\text{Pre}_{\text{reject\_out}}(q, e) \equiv e = \text{reject\_out} \wedge \text{PBalance}(q) < 0 \wedge (\forall c \in \text{Children})(q(\text{c\_estim})(c) \leq 0)$

$\text{Next}_{\text{reject\_out}}(q, r) \equiv r = q[(q(\text{p\_balance}) + 1)/\text{p\_balance}, 0/\text{p\_estim}]$

The various resource transfer events occur when an LRM owes a debt and has an available resource. Their effect is to cancel out some of the debt.

(parent_in) $\text{Pre}_{\text{parent\_in}}(q, e, res) \equiv e = \text{parent\_in}{:}res$

$\text{Next}_{\text{parent\_in}}(q, r, res) \equiv r = q[(q(\text{free}) \cup \{res\})/\text{free}, (q(\text{p\_balance}) - 1)/\text{p\_balance}]$

(parent_out) $\text{Pre}_{\text{parent\_out}}(q, e, res) \equiv e = \text{parent\_out}{:}res) \wedge res \in q(\text{free}) \wedge q(\text{p\_balance}) < 0$

$\text{Next}_{\text{parent\_out}}(q, r, res) \equiv r = q[(q(\text{free}) - \{res\})/\text{free}, (q(\text{p\_balance}) + 1)/\text{p\_balance}]$

(child_in) $\text{Pre}_{\text{child\_in}}(q, e, c, res) \equiv e = \text{child\_in}{:}\langle c, res\rangle$

$\text{Next}_{\text{child\_in}}(q, r, c, res) \equiv r = q[(q(\text{free}) \cup \{res\})/\text{free}, (q(\text{c\_balance})(c) - 1)/\text{c\_balance}(c)]$

(child_out) $\text{Pre}_{\text{child\_out}}(q, e, c, res) \equiv e = \text{child\_out}{:}\langle c, res\rangle \wedge res \in q(\text{free}) \wedge q(\text{c\_balance})(c) < 0$

$\text{Next}_{\text{child\_out}}(q, r, c, res) \equiv r = q[(q(\text{free}) - \{res\})/\text{free}, (q(\text{c\_balance})(c) + 1)/\text{c\_balance}(c)]$

The definition of the state-transition relation $\text{Trans}^{\text{RLRM}}$ for a root LRM is obtained by deleting the pairs above for the *forward_in*, *parent_out*, and *parent_in* events, and replacing the pair for *reject_out* events by the following pair for λ-events:

(λ) $\text{Pre}_{\lambda}(q, e) \equiv e = \lambda \wedge \text{PBalance}(q) < 0 \wedge (\forall c \in \text{Children})(q(\text{c\_estim})(c) \leq 0)$

$\text{Next}_{\lambda}(q, r) \equiv r = q[(q(\text{p\_balance}) + 1)/\text{p\_balance}, 0/\text{p\_estim}]$

The λ-transitions permitted by this pair are necessary for the consistency of the root LRM specification: if the *reject_out* pair were simply deleted as were the *forward_in*, *parent_out*, and *parent_in* pairs, then there would be no way for a root LRM to change the value of *p_balance* and the rely-condition $\text{Rely\_external}^{\text{RLRM}}$ defined below would be vacuous.

To complete the specification of the local resource manager, it remains to define the validity conditions. As outlined in the informal discussion above, the validity conditions for the node and root LRM's can be expressed in rely-/guarantee-condition form as follows:

$$\text{Valid}^{\text{NLRM}} \equiv \text{Rely}^{\text{NLRM}} \rightarrow \text{Guar}^{\text{NLRM}}$$
$$\text{Valid}^{\text{RLRM}} \equiv \text{Rely}^{\text{RLRM}} \rightarrow \text{Guar}^{\text{RLRM}}.$$

As was done in the informal discussion, it is convenient to factor the rely- and guarantee-conditions into what the LRM relies on each of its neighbors and the external environment to provide, and what the LRM guarantees in turn to each of its neighbors and the external environment.

The rely- and guarantee-conditions for the node LRM are defined by

$$\text{Rely}^{\text{NLRM}} \equiv \text{Rely\_parent}^{\text{NLRM}} \wedge (\forall c \in \text{Children})\text{Rely\_child}^{\text{LRM}}(c)$$
$$\text{Guar}^{\text{NLRM}} \equiv \text{Guar\_client}^{\text{LRM}} \wedge \text{Guar\_parent}^{\text{NLRM}} \wedge (\forall c \in \text{Children})\text{Guar\_child}^{\text{LRM}}(c).$$

The rely- and guarantee-conditions for the root LRM are defined by

$$\text{Rely}^{\text{RLRM}} \equiv \text{Rely\_external}^{\text{RLRM}} \wedge (\forall c \in \text{Children})\text{Rely\_child}^{\text{LRM}}(c)$$
$$\text{Guar}^{\text{RLRM}} \equiv \text{Guar\_client}^{\text{LRM}} \wedge (\forall c \in \text{Children})\text{Guar\_child}^{\text{LRM}}(c).$$

A node LRM relies on the eventual payment of debts owed to the LRM by its parent.

$$\text{Rely\_parent}^{\text{NLRM}} \equiv \Box(\Box(\mathbf{Now}(\text{p\_balance}) > 0) \rightarrow \Diamond(\exists r \in \text{Resources})(\mathbf{Occurs} = \text{parent\_in}{:}r))$$

Although a root LRM has no parent, the intuitive significance of a positive value for *p_balance* in the case of a root LRM is that the total number of requests in the entire tree exceeds the total number of available resources. Since we cannot expect a system of LRM's to eventually satisfy all requests under such circumstances, a root LRM relies on the external environment to ensure that *p_balance* is invariantly nonpositive.

$$\text{Rely\_external}^{\text{RLRM}} \equiv \Box(\mathbf{Now}(\text{p\_balance}) \leq 0)$$

Both kinds of LRM rely on each of their children to eventually eliminate debts owed to the LRM, either by the transmission of resources, or by the rejection of requests.

$$\text{Rely\_child}^{\text{LRM}}(c) \equiv \Box(\Box(\textbf{Now}(\text{c\_balance})(c) > 0) \rightarrow \Diamond((\exists r \in \text{Resources})(\textbf{Occurs} = \text{child\_in}{:}\langle c, r\rangle) \vee (\textbf{Occurs} = \text{reject\_in}{:}c)))$$

A node or root LRM guarantees to its client that pending requests will eventually receive a reply.

$$\text{Guar\_client}^{\text{LRM}} \equiv \Box(\textbf{Now}(\text{pending}) > 0 \rightarrow \Diamond(\exists r \in \text{Resource})(\textbf{Occurs} = \text{reply}{:}r))$$

A node LRM guarantees eventually to eliminate debts owed to its parent, either by actual transmission of resources, or by rejecting requests.

$$\text{Guar\_parent}^{\text{NLRM}} \equiv \Box(\Box(\textbf{Now}(\text{p\_balance}) < 0) \rightarrow \Diamond((\exists r \in \text{Resources})(\textbf{Occurs} = \text{parent\_out}{:}r) \vee (\textbf{Occurs} = \text{reject\_out})))$$

Both kinds of LRM guarantee eventually to pay debts owed to their children.

$$\text{Guar\_child}^{\text{LRM}}(c) \equiv \Box(\Box(\textbf{Now}(\text{c\_balance})(c) < 0) \rightarrow \Diamond(\exists r \in \text{Resources})(\textbf{Occurs} = \text{child\_out}{:}\langle c, r\rangle))$$

In devising the validity conditions for the local resource manager module, it was necessary to choose between two possible forms in which to state the rely- and guarantee-conditions. Since we are often faced with such choices in practice, it is useful to examine the motivation for the particular choice made here. As an example, consider the definition of Guar_parent$^{\text{NLRM}}$, which was stated above in the form

(1) $$\text{Guar\_parent}^{\text{NLRM}} \equiv \Box(\Box(\textbf{Now}(\text{p\_balance}) < 0) \rightarrow \Diamond((\exists r \in \text{Resources})(\textbf{Occurs} = \text{parent\_out}{:}r) \vee (\textbf{Occurs} = \text{reject\_out})))$$

This guarantee-condition states that either a *parent_out* or a *reject_out* will occur if there is the condition *p_balance* < 0 holds *persistently* (i.e. forever after some point). We might also have chosen the apparently stronger alternative form

(2) $$\text{Guar\_parent}^{\text{NLRM}} \equiv \Box(\textbf{Now}(\text{p\_balance}) < 0 \rightarrow \Diamond((\exists r \in \text{Resources})(\textbf{Occurs} = \text{parent\_out}{:}r) \vee (\textbf{Occurs} = \text{reject\_out}))).$$

which requires the occurence of a *parent_out* or *reject_out* event in the case that the condition *p_balance* < 0 occurs at a single instant. In fact, we claim these two sentences are equivalent in the context of the LRM specification. More precisely, we claim Comp$^{\text{LRM}} \models (1) \leftrightarrow (2)$. Clearly (2) implies (1) by temporal reasoning alone. To see that $\mathrm{Comp}^{LRM} \models \neg(2)$ implies $\neg(1)$, suppose Comp^{LRM} and $\neg(2)$. Then

(*) $\quad \Diamond(\mathbf{Now}(\mathrm{p\_balance}) < 0 \wedge \Box((\forall r \in \mathrm{Resources})(\mathbf{Occurs} \neq \mathrm{parent\_out}{:}r) \wedge (\mathbf{Occurs} \neq \mathrm{reject\_out})))$.

That is, eventually there is a point at which *p_balance* < 0 holds, but after which no *parent_out* or *reject_out* events ever occur. Inspection of the state-transition relation for the LRM shows that the only events that can cause *p_balance* to be increased are *parent_out* and *reject_out* events. This means that, if no *parent_out* or *reject_out* events occur, then *p_balance* < 0, once established, holds forever. Applying this result to (*) shows that

(**) $\quad \Diamond(\Box(\mathbf{Now}(\mathrm{p\_balance}) < 0) \wedge \Box((\forall r \in \mathrm{Resources})(\mathbf{Occurs} \neq \mathrm{parent\_out}{:}r) \wedge (\mathbf{Occurs} \neq \mathrm{reject\_out})))$.

But (**) is precisely the negation of (1) above, and thus (1) and (2) are equivalent.

In this example, where form (1) and form (2) are equivalent, we chose form (1) over form (2) because form (1) is more convenient for the proof of correctness. Once we have decided on form (1) for the guarantee condition $\mathrm{Guar\_parent}^{NLRM}$, we must use the same form for the complementary rely-condition $\mathrm{Rely\_child}^{LRM}(c)$. Similar arguments apply to $\mathrm{Guar\_child}^{LRM}(c)$ and $\mathrm{Rely\_parent}^{NLRM}$.

II.9.4 The Resource Manager Implementation Algebra

In this section we define the resource manager implementation algebra A^{RMI}. Let the following be given as parameters:

Clients:	a finite set of clients.
root:	a distinguished element of Clients
Children: Clients → Set[Clients]	maps each client to a set of children
Resources:	a finite set of resources
$\{\mathrm{Resources}_c : c \in \mathrm{Clients}\}$:	the initial partitioning of Resources.

We require that ⟨Clients, root, Children⟩ be a rooted tree. Let *parent*: (Clients − {root}) → Clients be the function that maps each $c \in$ Clients to its parent. Define the function PDesc: Clients → Set[Clients], which takes an element c of Clients to the set of all *proper descendants* of c, in terms of the function Children in the obvious way. Define $\mathrm{Desc}(c) = \{c\} \cup \mathrm{PDesc}(c)$ for all $c \in$ Clients.

The set Clients will be the index set for the interconnection; that is, there will be one LRM corresponding to each element of Clients. Define the embedded algebras A_{abs} and $\{A_p: p \in \text{Clients}\}$ as follows:

A_{abs}: is the resource manager event/state algebra A^{RM}, with parameters Clients, Resources instantiated as Clients, Resources, respectively.

A_{root}: is the local resource manager event/state algebra A^{LRM}, with parameters Resources, lResources, Children, $\{\text{estim}_c: c \in \text{Children(root)}\}$ instantiated as Resources, Resources_{root}, Children(root), $\{\Sigma_{d \in \text{Desc}(c)} |\text{Resources}_d|: c \in \text{Children(root)}\}$, respectively.

A_p: where $p \in \text{Clients} - \{\text{root}\}$, is the local resource manager event/state algebra A^{LRM}, with parameters Resources, lResources, Children, $\{\text{estim}_c: c \in \text{Children}(p)\}$ instantiated as Resources, Resources_p, Children(p), $\{\Sigma_{d \in \text{Desc}(c)} |\text{Resources}_d|: c \in \text{Children}(p)\}$, respectively.

Let the composite interface for the resource manager interconnection be defined as follows:

$$
\begin{array}{lll}
\text{Events}^{RMI} = \{\lambda\} + [\text{request:} & \text{Clients} + \\
\quad \text{reply:} & (\text{Clients} \times \text{Resources}) + \\
\quad \text{forward:} & (\text{Clients} - \{\text{root}\}) + \\
\quad \text{reject:} & \text{Clients} + \\
\quad \text{down:} & ((\text{Clients} - \{\text{root}\}) \times \text{Resources}) + \\
\quad \text{up:} & ((\text{Clients} - \{\text{root}\}) \times \text{Resources})] \\
\text{In}^{RMI} = \{\lambda\} + [\text{request:} & \text{Clients}] \\
\text{Out}^{RMI} = \{\lambda\} + (\text{Events}^{RMI} - \text{In}^{RMI}). &
\end{array}
$$

Intuitively, the event *request:p* corresponds to the receipt of a request by LRM p from its client, and *reply:*$\langle p, r\rangle$ corresponds to the allocation of resource r by LRM p to its client. The event *forward:p* represents the simultaneous occurrence of a *forward_in* event for LRM p, and a *forward_out:p* event for LRM parent(p). The event *reject:p* represents the simultaneous occurrence of a *reject_out* event for LRM p and a *reject_in:p* event for LRM parent(p). The event *down:*$\langle p, r\rangle$ represents the simultaneous occurrence of a *parent_in:r* event for LRM p and a *child_out:*$\langle p, r\rangle$ event for LRM parent(p). Finally, the event *up:*$\langle p, r\rangle$ represents the simultaneous occurrence of a *parent_out:r* event for LRM p and a *child_in:*$\langle p, r\rangle$ event for LRM parent(p). Formally, these relationships are captured by the following definitions of the abstraction map α^{RMI}, and the decomposition map $\underline{\delta}^{\mathrm{RMI}} = \langle \delta_p^{\mathrm{RMI}} \rangle_{p \in \mathrm{Clients}}$:

$$\begin{array}{lll} \alpha^{\mathrm{RMI}}(e) & = \text{request: } c & \text{if } e = \text{request: } c \\ & = \text{reply:}\langle c, r\rangle & \text{if } e = \text{reply:}\langle c, r\rangle \\ & = \lambda & \text{otherwise.} \end{array}$$

$$\begin{array}{lll} \delta_p^{\mathrm{RMI}}(e) & = \text{request} & \text{if } e = \text{request:}p \\ & = \text{reply:}r & \text{if } e = \text{reply:}\langle p, r\rangle \\ & = \text{forward\_in} & \text{if } e = \text{forward:}p \\ & = \text{forward\_out:}c & \text{if } e = \text{forward:}c \text{ and } p = \text{parent}(c) \\ & = \text{reject\_out} & \text{if } e = \text{reject:}p \\ & = \text{reject\_in:}c & \text{if } e = \text{reject:}c \text{ and } p = \text{parent}(c) \\ & = \text{child\_in:}\langle c, r\rangle & \text{if } e = \text{up:}\langle c, r\rangle \text{ and } p = \text{parent}(c) \\ & = \text{child\_out:}\langle c, r\rangle & \text{if } e = \text{down:}\langle c, r\rangle \text{ and } p = \text{parent}(c) \\ & = \text{parent\_out:}r & \text{if } e = \text{up:}\langle p, r\rangle \\ & = \text{parent\_in: } r & \text{if } e = \text{down:}\langle p, r\rangle \\ & = \lambda & \text{otherwise.} \end{array}$$

II.9.5 Proof of Correctness

In this section we prove the correctness of the implementation $\langle \mathcal{I}_{A^{\mathrm{RMI}}}, S_{\mathrm{abs}}, \langle S_c \rangle_{c \in \mathrm{Clients}} \rangle$, where S_{abs} is defined by $\langle A_{\mathrm{abs}}, \mathrm{Valid}^{\mathrm{RM}} \rangle$, S_{root} is defined by $\langle A_{\mathrm{root}}, \mathrm{Valid}^{\mathrm{RLRM}} \rangle$, and S_c for $c \in$ Clients – {root} is defined by $\langle A_c, \mathrm{Valid}^{\mathrm{NLRM}} \rangle$.

Implementation Invariant:

As usual, we factor the implementation invariant $\mathrm{Inv}^{\mathrm{RMI}}(q)$ for the resource manager implementation into an abstraction relation $\mathrm{Abs}^{\mathrm{RMI}}(q)$ and a representation invariant $\mathrm{Rep}^{\mathrm{RMI}}(q)$. The abstraction relation simply states that the set of free resources for the abstract resource manager module is just the union of the sets of free resources for each of the component LRM's, and that the multiset of pending requests for the abstract RM assigns to each client a multiplicity equal to the value of the state variable *pending* for the corresponding LRM.

$$\begin{array}{l} \mathrm{Abs}^{\mathrm{RMI}}(q) \equiv q_{\mathrm{abs}}(\mathrm{free}) = \cup_{c \in \mathrm{Clients}}\, q_c(\mathrm{free}) \wedge \\ \qquad\qquad q_{\mathrm{abs}}(\mathrm{pending}) = (\lambda c \in \mathrm{Clients})(q_c(\mathrm{pending})) \end{array}$$

It is convenient to factor the representation invariant into several conjuncts:

$$\text{Rep}^{\text{RMI}}(q) \equiv \text{Disjoint}(q) \wedge \text{Neighbor}(q) \wedge \text{Owed}(q) \wedge \text{Optim}(q).$$

The conjunct Disjoint(q) states that the sets of free resources possessed by two distinct LRM's are disjoint.

$$\text{Disjoint}(q) \equiv \wedge_{c,c' \in \text{Clients}}(c \neq c' \rightarrow q_c(\text{free}) \cap q_{c'}(\text{free}) = \emptyset).$$

The conjunct Neighbor(q) expresses the consistency constraints that hold between the values of the state variables for neighboring LRM's.

$$\text{Neighbor}(q) \equiv \wedge_{p \in \text{Clients}, c \in \text{Children}(p)}\{(q_c(\text{p\_balance}) = -q_p(\text{c\_balance})(c)) \wedge$$
$$(q_c(\text{p\_estim}) = q_p(\text{c\_estim})(c))\}$$

The conjunct Owed(q) states that an LRM can be owed resources by its parent only if the LRM estimates no surplus in the subtree of which it is the root.

$$\text{Owed}(q) = \wedge_{p \in \text{Clients}}(q_p(\text{p\_balance}) > 0 \rightarrow$$
$$\text{PBalance}(q_p) + \Sigma_{c \in \text{Children}(p)}\, q_p(\text{c\_estim})(c) \leq 0)$$

The conjunct Optim(q) states that the estimate *p_estim* held by an LRM p is optimistic in the sense that it is an upper bound on the actual projected number of resources remaining in the subtree of which p is the root, assuming that each estimate *c_estim*(c) held by p is an upper bound for the subtree rooted at c.

$$\text{Optim}(q) \equiv \wedge_{p \in \text{Clients}}(q_p(\text{p\_estim}) \geq$$
$$\text{PBalance}(q_p) + \Sigma_{c \in \text{Children}(p)}\, q_p(\text{c\_estim})(c)).$$

To show the inductiveness of $\text{Inv}^{\text{RMI}}(q)$, first note that the basis step, i.e. that $\text{Init}^{\text{RMI}}(q) \rightarrow \text{Inv}^{\text{RMI}}(q)$ holds for all $q \in$ States, is easily checked. A complete formal proof of the induction step, namely, that $\text{Trans}^{\text{RMI}}(q, e, r) \rightarrow (\text{Inv}^{\text{RMI}}(q) \rightarrow \text{Inv}^{\text{RMI}}(r))$, would be performed by case analysis on the event e. Such a complete proof would be quite tedious to read, and will not be included here. Rather, we will remark on the cases that are not quite trivial. Assume that $\text{Trans}^{\text{RMI}}(q, e, r)$ and $\text{Inv}^{\text{RMI}}(q)$ holds, to show $\text{Inv}^{\text{RMI}}(r)$. We consider each of the conjuncts of $\text{Inv}^{\text{RMI}}(r)$ in turn.

$\text{Abs}^{\text{RMI}}(r)$: The truth of this predicate depends upon the values of r_c(free) and r_c(pending), for each $c \in$ Clients, as well as r_{abs}(free) and r_{abs}(pending). The events e that affect the *pending* components of the state are *request*:c and *reply*:c. The effect of *request*:c is to add one instance of c to the multiset q_{abs}(pending), and to increment the value of q_c(pending) by one. Clearly this preserves the desired invariant relationship. The case of e = *reply*:<c, *res*>, is similar. The events e that affect the *free* components of the state are *up*:<c, *res*>, *down*:<c, *res*>, and *reply*:c. Because of the fact that each *up*:<c, res> or *down*:<c, res> is participated in only by LRM c and its parent, and the effect on the states of these two modules is complementary, it is easily verified that $\cup_{c \in \text{Clients}} r_c$(free) = $\cup_{c \in \text{Clients}} q_c$(free), holds for e = *down*:<c, *res*> or *up*:<c, *res*>. The case e = *reply*:<c, *res*> is slightly more troublesome, since to show that $\cup_{c \in \text{Clients}} r_c$(free) = ($\cup_{c \in \text{Clients}} q_c$(free)) – {*res*}, we need to make use of the inductive assumption that Disjoint(q) holds. From this we know that if *res* $\in q_c$(free), for some $c \in$ Clients, then *res* $\notin q_{c'}$(free) for all $c' \neq c$, and hence deleting *res* from q_c(free) in fact deletes it from the union of the free sets for all the component modules.

Disjoint(r): The truth of this predicate depends only upon the values of r_c(free) for each $c \in$ Clients. These components of the state are affected only by events of the form *reply*:<c, *res*>, *up*:<c, *res*>, and *down*:<c, *res*>. In case e = *reply*:<c, *res*>, we have that r_c(free) = q_c(free) – {*res*} and $r_{c'}$(free) = $q_{c'}$(free) for all $c' \in$ Clients with $c' \neq c$. In case e = *up*:<c, *res*>, and $c \in$ Children(p), we have that r_p(free) = q_p(free) $\cup$ {*res*}, r_c(free) = q_c(free) – {*res*}, and $r_{c'}$(free) = $q_{c'}$(free) for all $c' \in$ Clients with $c' \in$ Clients – {c, p}. In case e = *down*:<c, *res*>, and $c \in$ Children(p), we have that r_p(free) = q_p(free) – {*res*}, r_c(free) = q_c(free) + {*res*}, and $r_{c'}$(free) = $q_{c'}$(free) for all $c' \in$ Clients with $c' \in$ Clients – {c, p}. In each of these cases it is easily checked that Disjoint(r) holds.

Neighbor(r): Note that the predicate Neighbor(r) depends upon the values of r_c(p_balance), r_c(c_balance), r_c(p_estim), and r_c(c_estim), for each $c \in$ Clients. Enumeration of cases shows that the only events that affect the values of these components of the state are the events *reject*:c, *forward*:c, *down*:<c, *res*>, and *up*:<c, *res*>. However, examination of the definition of the LRM state-transition relation and the definition of the decomposition map $\underline{\delta}^{\text{RMI}}$ shows that each change in the state of a participant in one of these events is accompanied by a compensating change in the state of the other participant. For example, if $c \in$ Children(p), then occurrence of an event of the form *reject*:c makes r_c(p_balance) = q_c(p_balance) + 1, but also makes r_p(c_balance)(c) = q_p(c_balance)(c) – 1. Thus the predicate Neighbor is preserved.

Owed(r): Assuming that Owed(q) holds, the only way for Owed(r) to be false is for an event e to occur that increments q_p(p_balance) when it is zero, or increments

PBalance(q_p) + $\Sigma_{c \in \text{Children}(p)}$ q_p(c_estim)(c) when it is zero. The only events that might have this property are e = *reject*:p, and *up*:$\langle p$, *res*$\rangle$. In case e = *reject*:p, PBalance(q_p) + $\Sigma_{c \in \text{Children}(p)}$ q_p(c_estim)(c) is incremented, but the precondition for this event requires that this quantity be less than zero, so Owed(r) holds. In case e = *up*:$\langle p$, *res*$\rangle$, the quantity q_p(p_balance) is incremented, but the precondition for e requires that this quantity be strictly negative, and hence Owed(r) holds.

Optim(r): Assuming that Optim(q) holds, the only way for Optim(r) to be false is for the quantity q_c(p_estim) to be decreased below the quantity PBalance(q_c) + $\Sigma_{d \in \text{Children}(c)}$ q_c(c_estim)(d), or for the latter quantity to be increased above the former. The only events that could possibly have this effect are *forward*:c and *reject*:c. If e = *forward*:c, then q_c(p_estim) is decremented, but so is PBalance(q_c) + $\Sigma_{d \in \text{Children}(c)}$ q_c(c_estim)(d). If e = *reject*:c, then PBalance(q_c) + $\Sigma_{d \in \text{Children}(c)}$ q_c(c_estim)(d) is incremented and q_c(p_estim) is set to zero. However, the precondition for e requires that the former quantity be negative. This fact implies that PBalance(r_c) + $\Sigma_{d \in \text{Children}(c)}$ r_c(c_estim)(d) $\leq$ 0 and r_c(p_estim) = 0, and Optim(r) holds.

From the invariance of Owed, Neighbor, and Optim, we can derive the fundamental property of estimates upon which the correctness of the resource management system crucially depends. This property is expressed by Lemma II.1 below, which states that if an LRM i is owed resources by its parent, then the amount it is owed by its parent is a lower bound on the total instantaneous deficit in the subtree of which i is the root. To express this result formally, we introduce the quantity *IBalance*(q) where q is an LRM state, defined as follows:

$$\text{IBalance}(q) = |q(\text{free})| - q(\text{pending}).$$

Whereas the quantity *PBalance*(q) introduced previously represents the total *projected* balance of resources at an LRM, after all debts have been paid, the quantity *IBalance*(q) represents the total *instantaneous* balance of resources at an LRM, where the amount of indebtedness is not taken into account.

Lemma II.1 - The following is invariant for the resource manager implementation:

$$\wedge_{p \in \text{Clients}}(q_p(\text{p\_balance}) > 0 \rightarrow q_p(\text{p\_balance}) \leq -\Sigma_{c \in \text{Desc}(p)} \text{IBalance}(q_c))$$

Proof - From their definitions, it is easily seen that the quantities *PBalance*(q) and *IBalance*(q) are related by the following identity, expressed in the language $\mathcal{L}(A^{\text{LRM}})$ of the LRM event/state algebra:

$$\mathrm{PBalance}(q) = \mathrm{IBalance}(q) + q(\mathrm{p\_balance}) + \Sigma_{c \in \mathrm{Children}}\, q(\mathrm{c\_balance})(c).$$

From this identity, a simple induction on the height of a node $i \in$ Clients in the tree <Clients, root, Children>, shows the truth of the following identity for all $i \in$ Clients:

(1) $$\Sigma_{j \in \mathrm{Desc}(i)}\, \mathrm{PBalance}(q_j) = q_i(\mathrm{p\_balance}) + \Sigma_{j \in \mathrm{Desc}(p)}\, \mathrm{IBalance}(q_j).$$

That is, the total projected balance in the subtree of which i is the root is equal to the total instantaneous balance in that subtree, plus the net number of resources promised to be exchanged with the parent of i.

The invariance of Owed(q) means that the following is invariant:

(2) $$\wedge_{i \in \mathrm{Clients}}(q_i(\mathrm{p\_balance}) > 0 \rightarrow \mathrm{PBalance}(q_i) + \Sigma_{j \in \mathrm{Children}(i)}\, q_i(\mathrm{c\_estim})(j) \leq 0).$$

That is, if an LRM i is owed resources by its parent, then it must estimate no surplus of resources in the subtree of which i is the head, based on the estimates it has for each of its children.

The invariance of Neighbor(q) implies that the following is invariant:

(3) $$\wedge_{i \in \mathrm{Clients}}(\forall j \in \mathrm{Children}(i))(q_i(\mathrm{c\_estim})(j) = q_j(\mathrm{p\_estim})).$$

Substitution of (3) into (2) shows the invariance of

(4) $$\wedge_{i \in \mathrm{Clients}}(q_i(\mathrm{p\_balance}) > 0 \rightarrow \mathrm{PBalance}(q_i) + \Sigma_{j \in \mathrm{Children}(i)}\, q_j(\mathrm{p\_estim}) \leq 0).$$

Using the invariant Optim(q) to substitute for q_j(p_estim) in (4) shows that

(5) $$\wedge_{i \in \mathrm{Clients}}(q_i(\mathrm{p\_balance}) > 0 \rightarrow \mathrm{PBalance}(q_i) + \Sigma_{j \in \mathrm{Children}(i)}\, (\mathrm{PBalance}(q_j) + \Sigma_{k \in \mathrm{Children}(j)}\, q_j(\mathrm{c\_estim})(k)) \leq 0).$$

is invariant. Repeating this argument to eliminate all occurrences of c_estim yields the invariance of

(6) $$\wedge_{i \in \mathrm{Clients}}(q_i(\mathrm{p\_balance}) > 0 \rightarrow \Sigma_{j \in \mathrm{Desc}(i)}\, \mathrm{PBalance}(q_j) \leq 0),$$

which states, intuitively, that if LRM i is owed resources by its parent, then there can be no projected surplus of resources in the subtree of which i is the root.

By using (1) to eliminate PBalance in favor of IBalance in (6), we obtain the invariance of

(7) $\qquad \bigwedge_{i \in \text{Clients}}(q_i(\text{p\_balance}) > 0 \rightarrow$

$$q_i(\text{p\_balance}) + \Sigma_{j \in \text{Desc}(i)} \text{IBalance}(q_j) \leq 0),$$

which is equivalent to the desired result. ▮

Proof of Maximality

The maximality verification condition is:

$$\models (\forall q \in \text{States}, e \in \text{Events})(\text{Inv}^{\text{RMI}}(q) \wedge \bigwedge_{c \in \text{Clients}} \text{Enabled}_c(q, e) \\ \rightarrow \text{Enabled}_{\text{abs}}(q, e)).$$

The proof of this assertion is most easily performed by a case analysis on the event e; making use of the fact that the module specifications define the state-transition relation by precondition/next-state predicate pairs. If e = *forward*:c, *reject*:c, *down*:$\langle c, r\rangle$, or *up*:$\langle c, r\rangle$, then $\alpha^{\text{RMI}}(e) = \lambda$, and hence $\text{Enabled}_{\text{abs}}(q, e) \equiv$ **true**. We therefore need consider only the cases e = *request*:c and e = *reply*:$\langle c, r\rangle$. If e = *request*:c, then $\alpha^{\text{RMI}}(e)$ = *request*:c, and hence $\text{Enabled}_{\text{abs}}(q, e) \equiv$ **true**.

We are left with the case e = *reply*:$\langle c, r\rangle$. In this case, we obtain the following from the module specifications:

$$\text{Enabled}_{\text{abs}}(q, e) \equiv r \in q_{\text{abs}}(\text{free}) \wedge c \in q_{\text{abs}}(\text{pending})$$
$$\text{Enabled}_c(q, e) \quad \equiv r \in q_c(\text{free}) \wedge q_c(\text{pending}) > 0$$
$$\text{Enabled}_p(q, e) \quad \equiv \textbf{true}, \qquad\qquad \text{if } p \in \text{Clients} - \{c\}.$$

Assume $\text{Inv}^{\text{RMI}}(q)$, and hence $\text{Abs}^{\text{RMI}}(q)$, holds. Assume further that $\bigwedge_{p \in \text{Clients}} \text{Enabled}_p(q, e)$ holds. From $\text{Enabled}_c(q, e)$ we know that $r \in q_c(\text{free}) \wedge q_c(\text{pending}) > 0$ holds. From this and $\text{Abs}^{\text{RMI}}(q)$ we infer that $r \in q_{\text{abs}}(\text{free}) \wedge c \in q_{\text{abs}}(\text{pending})$ holds, as desired.

Proof of Validity

To prove that the validity verification condition holds for the resource manager implementation, we use Corollary I.5. To apply Corollary I.5, we must find, for each $i, j, \in$ Clients + {abs}, a sentence $\text{RG}_{i,j}$ of $\mathfrak{T}(A^{\text{RMI}})$ such that the following hold:

(RMI1)(a) $\quad \text{Comp}^{\text{RMI}} \models [\![\text{Rely}^{\text{RM}}]\!]_{\text{abs}} \rightarrow \bigwedge_{j \in \text{Clients}} \text{RG}_{\text{abs},j}$

(RMI1)(b) $\quad \text{Comp}^{\text{RMI}} \models \bigwedge_{i \in \text{Clients}} \text{RG}_{i,\text{abs}} \rightarrow [\![\text{Guar}^{\text{RM}}]\!]_{\text{abs}}$

(RMI2)(a)

(root) $\quad \text{Comp}^{\text{RMI}} \models (\bigwedge_{i \in \text{Clients}+\{\text{abs}\}} \text{RG}_{i,\text{root}} \rightarrow [\![\text{Rely}^{\text{RLRM}}]\!]_{\text{root}})$

(node) $\quad \text{Comp}^{\text{RMI}} \models \bigwedge_{j \in \text{Clients}-\{\text{root}\}} (\bigwedge_{i \in \text{Clients}+\{\text{abs}\}} \text{RG}_{i,j} \rightarrow [\![\text{Rely}^{\text{NLRM}}]\!]_{j})$

(RMI2)(b)

(root) $\quad \text{Comp}^{\text{RMI}} \models ([\![\text{Guar}^{\text{RLRM}}]\!]_{\text{root}} \rightarrow \bigwedge_{j \in \text{Clients}+\{\text{abs}\}} \text{RG}_{\text{root},j})$

(node) $\quad \text{Comp}^{\text{RMI}} \models \bigwedge_{i \in \text{Clients}-\{\text{root}\}} ([\![\text{Guar}^{\text{NLRM}}]\!]_{i} \rightarrow \bigwedge_{j \in \text{Clients}+\{\text{abs}\}} \text{RG}_{i,j})$

(RMI3) Whenever $\{\langle i_0, i_1\rangle, \langle i_1, i_2\rangle, \ldots, \langle i_{n-1}, i_n\rangle\}$ is a cycle of Clients, then

$$\text{Comp}^{\text{RMI}} \models \bigvee_{k=0}^{n-1} \text{RG}_{i_k, i_{k+1}}.$$

The sentences $\text{RG}_{i,j}$ bear a particular relationship, formalized in Lemma II.2 below, to the various conjuncts appearing in the local resource manager validity conditions. Lemma II.2 states in essence, that the local resource manager validity conditions are "localized" versions of the sentences $\text{RG}_{i,j}$. Part (a) of Lemma II.2 states that the sentence $\text{RG}_{i,\text{parent}(i)}$ captures exactly what LRM i guarantees to its parent and exactly what LRM parent(i) relies on i to provide. Part (b) states that the sentence $\text{RG}_{\text{parent}(j),j}$ captures exactly what LRM j relies on its parent to provide, and exactly what LRM parent(j) guarantees to provide to j. Part (c) states that the sentence $\text{RG}_{\text{abs},\text{root}}$ captures exactly what the root LRM relies on the external environment of the system of LRM's to provide. Part (d) states that the sentence $\text{RG}_{i,\text{abs}}$ captures exactly what LRM i guarantees to provide to the external environment of the system of LRM's.

The sentences $\text{RG}_{i,j}$ are defined as follows:

$$\text{RG}_{\text{root},\text{abs}} \equiv \Box(\textbf{Now}_{\text{root}}(\text{pending}) > 0 \rightarrow \Diamond(\exists r \in \text{Resources})(\delta^{\text{RMI}}_{\text{root}}(\textbf{Occurs}) = \text{reply}{:}r))$$

$$\text{RG}_{\text{abs},\text{root}} \equiv \Box(\textbf{Now}_{\text{root}}(\text{p\_balance}) \leq 0)$$

For all $i, j \in$ Clients − {root}:

$$\text{RG}_{i,\text{abs}} \equiv \Box(\textbf{Now}_{i}(\text{pending}) > 0 \rightarrow \Diamond(\exists r \in \text{Resources})(\delta^{\text{RMI}}_{i}(\textbf{Occurs}) = \text{reply}{:}r))$$

$$\text{RG}_{\text{abs},j} \equiv \textbf{true}.$$

For all $i \in$ Clients − {root}:

$$\text{RG}_{i,\text{parent}(i)} \equiv \Box(\Box(\textbf{Now}_{i}(\text{p\_balance}) < 0) \rightarrow \Diamond((\exists r \in \text{Resources})(\delta^{\text{RMI}}_{i}(\textbf{Occurs}) = \text{parent\_out}{:}r) \vee (\delta^{\text{RMI}}_{i}(\textbf{Occurs}) = \text{reject\_out}))$$

$$\mathrm{RG}_{\mathrm{parent}(i),i} \equiv \Box(\Box(\mathbf{Now}_{\mathrm{parent}(i)}(\mathrm{c\_balance})(i) < 0) \rightarrow$$
$$\Diamond(\exists r \in \mathrm{Resources})(\delta^{\mathrm{RMI}}_{\mathrm{parent}(i)}(\mathbf{Occurs}) = \mathrm{child\_out}{:}\langle i, r\rangle))$$

For all $i, j \in$ Clients such that neither i = parent(j) or j = parent(i):

$$\mathrm{RG}_{i,j} \qquad \equiv \mathbf{true}.$$

Lemma II.2 - The following are valid for the resource manager implementation:

(a) For all $i \in$ Clients – {root},

$$\mathrm{Comp}^{\mathrm{RMI}} \models \mathrm{RG}_{i,\mathrm{parent}(i)} \leftrightarrow$$
$$[\![\mathrm{Guar\_parent}^{\mathrm{NLRM}}]\!]_i \leftrightarrow$$
$$[\![\mathrm{Rely\_child}^{\mathrm{LRM}}(i)]\!]_{\mathrm{parent}(i)}$$

(b) For all $i \in$ Clients – {root},

$$\mathrm{Comp}^{\mathrm{RMI}} \models \mathrm{RG}_{\mathrm{parent}(j),j} \leftrightarrow$$
$$[\![\mathrm{Rely\_parent}^{\mathrm{NLRM}}]\!]_j \leftrightarrow$$
$$[\![\mathrm{Guar\_child}^{\mathrm{LRM}}(j)]\!]_{\mathrm{parent}(j)}$$

(c) $\mathrm{Comp}^{\mathrm{RMI}} \models \mathrm{RG}_{\mathrm{abs,root}} \leftrightarrow [\![\mathrm{Rely\_external}^{\mathrm{RLRM}}]\!]_{\mathrm{root}}$

(d) For all $i \in$ Clients,

$$\mathrm{Comp}^{\mathrm{RMI}} \models \mathrm{RG}_{i,\mathrm{abs}} \leftrightarrow [\![\mathrm{Guar\_client}^{\mathrm{LRM}}]\!]_i.$$

Proof - Straightforward, using the invariance of Neighbor and the definition of the decomposition map $\underline{\delta}^{\mathrm{RMI}}$. ∎

Lemma II.3 - Under the definitions given above for the sentences $\mathrm{RG}_{i,j}$, conditions (RMI1)-(RMI3) hold for the resource manager implementation.

Proof - Assume $\mathrm{Comp}^{\mathrm{RMI}}$.

To prove that (RMI1)(a) holds, we must show

$$[\![\mathrm{Rely}^{\mathrm{RM}}]\!]_{\mathrm{abs}} \rightarrow \bigwedge_{i \in \mathrm{Client}} \mathrm{RG}_{\mathrm{abs},i}$$

Suppose that $[\![\mathrm{Rely}^{\mathrm{RM}}]\!]_{\mathrm{abs}}$ holds. It suffices to prove that $\mathrm{RG}_{\mathrm{abs,root}}$ holds, since $\mathrm{RG}_{\mathrm{abs},i} \equiv$ **true** for all $i \in$ Clients – {root}. $[\![\mathrm{Rely}^{\mathrm{RM}}]\!]_{\mathrm{abs}}$ is defined by:

$[\![\mathrm{Rely}^{\mathrm{RM}}]\!]_{\mathrm{abs}} \equiv \Box(|\mathbf{Now}_{\mathrm{abs}}(\mathrm{free})| \geq |\mathbf{Now}_{\mathrm{abs}}(\mathrm{pending})|)$. Using this and the invariance of the predicate $\mathrm{Abs}^{\mathrm{RMI}}$, we infer the truth of $\Box(\Sigma_{i \in \mathrm{Clients}} (|\mathbf{Now}_i(\mathrm{free})| - \mathbf{Now}_i(\mathrm{pending})) \geq 0)$, which is equivalent to

(A) $\qquad \Box(\Sigma_{i \in \mathrm{Clients}} \mathrm{lBalance}(\mathbf{Now}_i) \geq 0)$.

From Lemma II.1 and the fact that Desc(root) = Clients, we infer that $\Box(\mathbf{Now}_{root}(\text{p\_balance}) > 0 \rightarrow \mathbf{Now}_{root}(\text{p\_balance}) \leq -\Sigma_{i \in \text{Clients}}\ \text{lBalance}(\mathbf{Now}_i))$. From this and (A), we conclude that $\Box(\mathbf{Now}_{root}(\text{p\_balance}) \leq 0)$, which is precisely the statement that $\text{RG}_{abs,root}$ holds.

To prove (RMI1)(b), we must show

$$\wedge_{i \in \text{Clients}} \text{RG}_{i,abs} \rightarrow [\![\text{Guar}^{\text{RM}}]\!]_{abs}$$

Suppose that $\wedge_{i \in \text{Clients}} \text{RG}_{i,abs}$ holds. From the definition of $\text{RG}_{i,abs}$ we know that

$$\wedge_{i \in \text{Clients}} \Box(\mathbf{Now}_i(\text{pending}) > 0 \rightarrow \Diamond(\exists r \in \text{Resources})(\delta_i^{\text{RMI}}(\mathbf{Occurs}) = \text{reply:}r))$$

holds. From the invariance of Abs^{RMI} and the definition of the abstraction map α^{RMI} we infer that

$$\wedge_{i \in \text{Clients}} \Box(i \in \mathbf{Now}_{abs}(\text{pending}) \rightarrow \\ \Diamond(\exists r \in \text{Resources})(\alpha^{\text{RMI}}(\mathbf{Occurs}) = \text{reply:}\langle i,r\rangle))$$

holds. This is precisely the statement that $[\![\text{Guar}^{\text{RM}}]\!]_{abs}$ holds.

We next prove (RMI2)(a). In case (*root*), we must show

(*root*) $\wedge_{i \in \text{Clients}+\{abs\}} \text{RG}_{i,root} \rightarrow [\![\text{Rely}^{\text{RLRM}}]\!]_{root}$.

From the root LRM specifications we know that

$$[\![\text{Rely\_external}^{\text{RLRM}}]\!]_{root} \wedge \wedge_{i \in \text{Children(root)}} [\![\text{Rely\_child}^{\text{LRM}}(i)]\!]_{root} \leftrightarrow [\![\text{Rely}^{\text{RLRM}}]\!]_{root}.$$

Using Lemma II.2 (a) and (c) we infer that

$$\text{RG}_{abs,root} \wedge \wedge_{i \in \text{Children(root)}} \text{RG}_{i,root} \leftrightarrow [\![\text{Rely}^{\text{RLRM}}]\!]_{root},$$

which implies formula (*root*).

In case (*node*) we must show that for all $j \in$ Clients,

(*node*) $\wedge_{i \in \text{Clients}+\{abs\}} \text{RG}_{i,j} \rightarrow [\![\text{Rely}^{\text{NLRM}}]\!]_j$.

Fix j to be an arbitrary element of Clients. From the node LRM specifications we know that

$$[\![\text{Rely\_parent}^{\text{NLRM}}]\!]_j \wedge \wedge_{i \in \text{Children}(j)} [\![\text{Rely\_child}^{\text{LRM}}(i)]\!]_j \leftrightarrow [\![\text{Rely}^{\text{NLRM}}]\!]_j.$$

Using Lemma II.2 (a) and (b) we infer that

$$\text{RG}_{\text{parent}(j),j} \wedge \wedge_{i \in \text{Children}(j)} \text{RG}_{i,j} \leftrightarrow [\![\text{Rely}^{\text{NLRM}}]\!]_j,$$

which implies formula (*node*).

We next show (RMI2)(b). In case (*root*) we must show

(*root*) $[\![\text{Guar}^{\text{RLRM}}]\!]_{root} \rightarrow \wedge_{j \in \text{Clients}+\{abs\}} \text{RG}_{root,j}$.

From the root LRM specifications we have

$$[\![\text{Guar}^{\text{RLRM}}]\!]_{root} \leftrightarrow [\![\text{Guar\_client}^{\text{LRM}}]\!]_{root} \wedge \wedge_{j \in \text{Children(root)}} [\![\text{Guar\_child}^{\text{LRM}}(j)]\!]_{root}$$

Using Lemma II.2 (b) and (d), we infer

$$[\![\text{Guar}^{\text{RLRM}}]\!]_{\text{root}} \leftrightarrow \text{RG}_{\text{root,abs}} \wedge \bigwedge_{j \in \text{Children(root)}} \text{RG}_{\text{root},j}.$$

This implies formula (*root*), since $\text{RG}_{\text{root},j} \equiv$ **true** unless j = abs or $j \in$ Children(root).

In case (*node*) we must show that, for all $i \in I$:

(*node*) $[\![\text{Guar}^{\text{NLRM}}]\!]_i \rightarrow \bigwedge_{j \in \text{Clients}+\{\text{abs}\}} \text{RG}_{i,j}$.

Let i be an arbitrary element of Clients, and assume $[\![\text{Guar}^{\text{NLRM}}]\!]_i$. From the node LRM specifications we have

$$[\![\text{Guar}^{\text{NLRM}}]\!]_i \leftrightarrow [\![\text{Guar\_client}^{\text{LRM}}]\!]_i \wedge [\![\text{Guar\_parent}^{\text{NLRM}}]\!]_i \wedge \bigwedge_{j \in \text{Children}(i)} [\![\text{Guar\_child}^{\text{LRM}}(j)]\!]_i$$

Using Lemma II.2 (a), (b), and (d), we infer

$$[\![\text{Guar}^{\text{NLRM}}]\!]_i \leftrightarrow \text{RG}_{i,\text{abs}} \wedge \text{RG}_{i,\text{parent}(i)} \wedge (\forall j \in \text{Children}(i))\text{RG}_{i,j}.$$

This implies formula (*node*), since $\text{RG}_{i,j} \equiv$ **true** unless j = abs, j = parent(i), or $j \in$ Children(i).

To prove (RMI3) it suffices to show that $\text{RG}_{i,\text{parent}(i)} \vee \text{RG}_{\text{parent}(i),i}$ holds for all $i \in$ Clients − {root}. This is because every cycle $\{\langle i_0, i_1\rangle, \ldots, \langle i_{n-1}, i_n\rangle\}$ of Clients either contains a link $\langle i_k, i_{k+1}\rangle$ for which $\text{RG}_{i_k,i_{k+1}} \equiv$ **true** by definition, or else contains both links $\langle i$, parent(i)$\rangle$ and $\langle$parent(i), $i\rangle$ for some $i \subset$ Clients − {root}.

To show $\text{RG}_{i,\text{parent}(i)} \vee \text{RG}_{\text{parent}(i),i}$ holds for all $i \in$ Clients − {root}, let i be arbitrarily fixed, and suppose that $\neg\text{RG}_{i,\text{parent}(i)}$ holds, to show that $\text{RG}_{\text{parent}(i),i}$ holds. By definition of $\text{RG}_{i,\text{parent}(i)}$ we know that $\Diamond\Box(\text{Now}_i(\text{p\_balance}) < 0)$ holds. By the invariance of Neighbor, we infer that $\Diamond\Box(\text{Now}_{\text{parent}(i)}(\text{c\_balance})(i) > 0)$ holds. This implies that $\text{RG}_{\text{parent}(i),i}$ holds. ∎

II.10 A Message Transmission System

In this section we consider the specification and implementation of a *message transmission module* TM, whose function is to reliably deliver messages input by one user, called the *sender*, to another user, called the *receiver*. Messages should be delivered in the order in which they are sent, and should not be subject to loss or duplication. The message transmission module therefore behaves as a FIFO buffer between the sender and the receiver. The interesting part of this example is how the reliable FIFO buffer behavior of the message transmission module is implemented by a *transmission module implementation* TMI, which consists, in part, of unreliable *transmission line* components. This is accomplished through the use of a *send protocol module* and a *receive protocol module*, which together implement the *alternating bit* *protocol* [Bartlett69].

The alternating bit protocol is a standard example for which correctness proofs in varying styles have been given by other researchers. Most analyses treat only safety properties, however the proofs given by Hailpern and Owicki [Hailpern80] and Lamport [Lamport83] treat liveness properties of the protocol in addition to safety properties. The major deficiency of Hailpern and Owicki's treatment is the unstructured and apparently *ad hoc* nature of the specifications and the correctness proof. It is difficult to discern from their work very much in the way of a general method (with the exception of their use of history variables, which can be seen as a special case of the state-transition approach espoused here) likely to be applicable to other examples. In contrast, the specifications and correctness proof given below are an instance of a general strategy, which is embodied in the state-transition approach to specification, the use of rely- and guarantee-conditions, and the Correctness Theorem.

Of the extant proofs of the correctness of the alternating bit protocol, that of Lamport [Lamport83] is perhaps the most similar to the one given here. The modules are specified in a state-transition style quite like that proposed here. It is possible to identify portions of Lamport's proof that correspond to the proof of invariance of the abstraction relation and implementation invariant given below. The major difference between Lamport's proof and the one given here is in the statement and proof of the liveness (i.e. validity) properties. Lamport's liveness specifications for the send protocol module take the form: "If the send protocol module has an unprocessed message, then it will eventually give a packet containing that message to the unreliable transmission medium for transmission to the receive protocol module;" "If a correct acknowledgement is received by the send protocol module, then eventually the protocol will progress to the next unprocessed message;" etc. These are "low-level" specifications that can be thought of as essentially a set of assertions that might appear in an assertional proof that a particular program satisfies the specification. In contrast, the specifications given here are of the form: "If the send protocol module can rely on the fact that sufficiently many transmissions of packet *p* will eventually result in the receipt of an acknowledgement for packet *p*, then it guarantees eventually to process every message given to it as input." This is a "higher-level" specification that states what the send protocol module accomplishes without detailing a chain of intermediate steps by which it is accomplished.

A feature that distinguishes the proof presented here from previous proofs, is that the proof here is to a great extent independent of the precise assumptions on the reliability of the transmission line components. The specifications of the transmission line module are expressed in the form: "If a message *m* is transmitted according to certain conditions Xmit(*m*), then eventually *m* will be delivered according to conditions Dlvr(*m*)." For concreteness, we use "*m* is transmitted repeatedly, without intervening transmission of any different message *m* '" for Xmit(*m*), and a symmetric condition for Dlvr(*m*). However, Xmit(*m*) and Dlvr(*m*) can easily be replaced with alternative conditions without change to the proof structure.

II.10.1 Specification of the Message Transmission Module

The interface of the message transmission module TM consists of two kinds of events: those of the form *TM_in*:*m*, in which message *m* is presented to the transmission module by the sender, and events of the form *TM_out*:*m*, in which message *m* is delivered by the transmission module to the receiver. We wish to state that the transmission module delivers messages in FIFO order without loss or duplication. We can think of the state of the transmission module as a sequence of the messages input by the sender but not yet delivered to the receiver. Equivalently, and for our purposes more conveniently, the state of the transmission module can be thought of as a pair <inq, outq> of sequences of messages, where inq represents the entire history of messages that have ever been input to the transmission module by the sender, and outq represents the entire history of messages that have ever been output to the receiver by the transmission module. The sequence of messages sent but not yet delivered by the transmission module is represented, in this alternative state set, by the sequence inq–outq.

Based on this selection of state set, let us now derive a precise specification of the transmission module.

Let Values be a finite set of message values, given as a parameter. The interface of the transmission module is defined as follows:

$$\begin{aligned} \text{Events}^{\text{TM}} &= \{\lambda\} + [\text{TM\_in: Values} + \text{TM\_out: Values}]. \\ \text{In}^{\text{TM}} &= \{\lambda\} + [\text{TM\_in: Values}] \\ \text{Out}^{\text{TM}} &= \{\lambda\} + [\text{TM\_out: Values}]. \end{aligned}$$

The state set for the transmission module is defined by:

$\text{States}^{TM} = [\text{inq}: \text{Seq}[\text{Values}] \times \text{outq}: \text{Seq}[\text{Values}]]$.

If $q \in \text{States}^{TM}$, then we write $q(\text{inq-outq})$ as an abbreviation for $q(\text{inq}) - q(\text{outq})$.

In an initial state for the transmission module, the queue is empty.

$\text{Init}^{TM}(q) \equiv q(\text{inq}) = q(\text{outq})$.

The state transition relation Trans^{TM} is defined by precondition/next-state predicate pairs as follows:

An input event with message m can occur at any time, and causes message m to be appended to the end of inq.

(TM_in) $\text{Pre}_{TM\_in}(q, e, m) \equiv e = \text{TM\_in}{:}m$

$\text{Next}_{TM\_in}(q, r, m) \equiv r = q[(q(\text{inq}))\cdot m/\text{inq}]$

An output event with message m can occur only if there is a message that has been sent but not yet delivered, and m is the first such message. The effect is to append m to the end of outq.

(TM_out) $\text{Pre}_{TM\_out}(q, c, m) \equiv e = \text{TM\_out}{:}m \wedge q(\text{outq}) < q(\text{inq}) \wedge m = (q(\text{inq-outq})(0)$

$\text{Next}_{TM\_out}(q, r, m) \equiv r = q[(q(\text{outq}))\cdot m/\text{outq}]$.

We wish the validity condition for the transmission module to capture the requirement that every message sent is eventually delivered. This is captured by the definition below, which states that, given any prefix s of inq, there is eventually a later time at which s is also a prefix of outq.

$\text{Valid}^{TM} \equiv \text{Rely}^{TM} \rightarrow \text{Guar}^{TM}$

where

$\text{Rely}^{TM} \equiv \textbf{true}$

$\text{Guar}^{TM} \equiv \Box(\forall s \in \text{Seq}[\text{Values}])(s \leq \textbf{Now}(\text{inq}) \rightarrow \Diamond(s \leq \textbf{Now}(\text{outq})))$.

II.10.2 Implementation of the Transmission Module

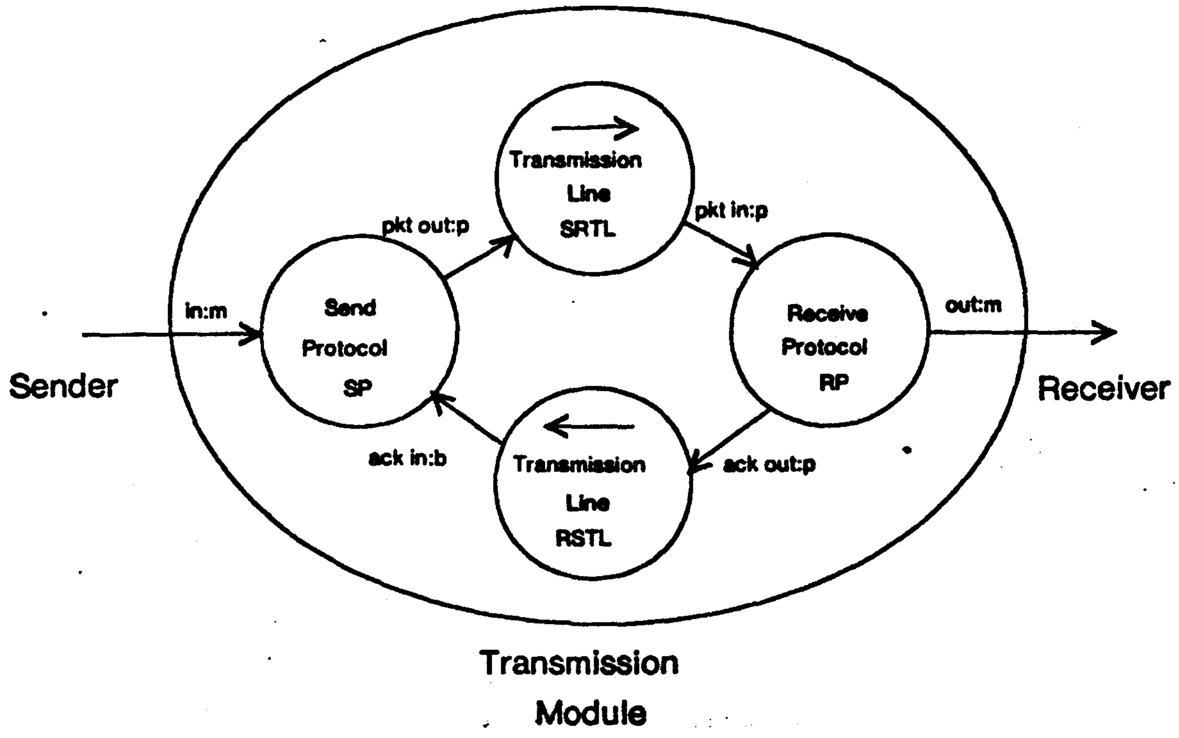
Figure 4 shows the implementation of the transmission module by a *send protocol module* SP, a *receive protocol module* RP, a sender-to-receiver *transmission line module* SRTL, and a receiver-to-sender transmission line module RSTL. Messages received from the sender by the send protocol module (an "in" event) are placed in a queue for transmission to the receive protocol module. When this queue is nonempty, a *packet* consisting of the first message in the queue and a current boolean *sequence number* is transmitted (via a "pkt_out" event) by the send protocol module SP over the transmission line SRTL. In contrast to the reliable transmission module specified in the preceding section, the transmission line module is inherently unreliable, and might lose or duplicate messages. We require, however, that the transmission line not reorder messages. Since messages might be lost, in general it will be necessary for the send protocol module to transmit the same packet a number of times before it is delivered to the receive protocol module. Thus the send protocol module continues to send the packet until an *acknowledgement* for the sequence number it contains is received (an "ack_in" event) over the transmission line module RSTL. Receipt of a correct acknowledgement by the send protocol module causes the first message to be removed from its queue. In addition the send protocol module complements its sequence number.

When a packet arrives at the receive protocol module (via a "pkt_in" event), it is checked to see if its sequence number is current. If the sequence number is current, then the message is extracted and placed in a queue of messages to be delivered to the receiver. Also, the sequence number expected by the receive protocol module is complemented. The receive protocol module ignores packets that do not contain the current sequence number. The receive protocol module transmits acknowledgements for the most recently received packet over the transmission line module RSTL (via "ack_out" events). Whenever the queue of messages to be delivered to the receiver is nonempty, then a message can be removed and sent to the receiver (via "out" events).

II.10.3 Specification of the Transmission Line Module

The interface of the transmisssion line module contains events of the form *TL_in:m*, which correspond to the presentation of message *m* for transmission, and of the form *TL_out:m*, which correspond to the delivery of message *m* to its destination. Thus, the interface of the transmission line module TL is isomorphic to that of the message transmission module. The difference between the two modules lies in the fact that, whereas the transmission module guarantees to deliver each message exactly once, the transmission line module is permitted to lose or duplicate messages any number of times. We require, however, that the transmission line module not reorder messages. Also, we require that repeated input of messages to the transmission line

Fig. 4. Transmission Module Implementation



module will eventually cause messages to be delivered.

We will use the same state set for the transmission line specification as we used for the transmission module specification. However, the intuitive meanings of the components *inq* and *outq* of the state are significantly changed, as is the state-transition relation and validity condition. For the transmission line module, the sequence *inq* represents a sequence of messages, each of which is destined to be delivered at least once. However, each message in *inq* might be delivered more than once. The sequence *outq* represents the messages in *inq*, each of which has already had all its copies delivered, and will therefore never be delivered again. The state transition relation is modified to permit message loss and duplication as follows: The possibility of message loss is captured by the fact that input events are permitted either to produce no state change (corresponding to the loss of the associated message) or to append the message exactly once to the end of *inq* (indicating that the message is destined to be delivered eventually at least once). The possibility of message duplication is captured by the fact that output events are permitted either to produce no state change (corresponding to the duplication of the message just delivered) or to add the message to *outq* (corresponding to the delivery of the final copy of the message).

Note that the preceding description is only one of many possible ways of presenting the same transmission line specification. For example, we could have captured the possibility of message loss or duplication by stating that the occurrence of a *TL_in:m* event causes the message *m* to be appended *k* times to *inq*, where *k* is a nondeterministically chosen natural number. Occurrence of a *TL_out:m* event would then be possible only if *m* is the first element of *inq* not also in *outq*, and would cause *m* to be appended precisely once to *outq*. The transmission line specification is an example of an *indeterminate* specification (see Section 6.2), which means that a single observation can be produced in more than one computation. Although we could give a determinate transmission line specification equivalent to the indeterminate version used here, the use of an indeterminate specification seems more natural.

We now make the above informal specification more precise. As in the case of the transmission module specification, let Values be a finite set of message values, given as a parameter. Define the interface of the transmission line module as follows:

$$\text{Events}^{\text{TL}} = \{\lambda\} + [\text{TL\_in: Values} + \text{TL\_out: Values}].$$
$$\text{In}^{\text{TL}} = \{\lambda\} + [\text{TL\_in: Values}]$$
$$\text{Out}^{\text{TL}} = \{\lambda\} + [\text{TL\_out: Values}]$$

Define the state set of the transmission line module by:

$$\text{States}^{\text{TL}} = [\text{inq: Seq[Values]} \times \text{outq: Seq[Values]}].$$

In an initial state, the transmission line queue is empty.

$$\text{Init}^{\text{TL}}(q) \equiv q(\text{inq}) = q(\text{outq}).$$

The state-transition relation Trans$^{\text{TL}}$ is defined as follows:

An input event with message *m* can occur at any time, and either causes no change in state (the message is lost) or appends the message to the end of the queue (the message is destined to be delivered).

(TL_in) $\text{Pre}_{\text{TL\_in}}(q, e, m) \equiv e = \text{TL\_in}{:}m$

$\text{Next}_{\text{TL\_in}}(q, r, m) \equiv r = q \vee r = q[(q(\text{inq}))\cdot m/\text{inq}]$

An output event with message *m* can occur only if there is a message that has been sent but for which the last copy has not yet been delivered, and *m* is the first such message. The message *m* is either appended to *outq* (corresponding to the last copy of *m* being delivered), or there is no state change (corresponding to the duplication of m).

(TL_out) $\quad \mathrm{Pre}_{\mathrm{TL\_out}}(q, e, m) \equiv e = \mathrm{TL\_out}{:}m \wedge q(\mathrm{outq}) < q(\mathrm{inq}) \wedge$

$m = (q(\mathrm{inq}\text{-}\mathrm{outq}))(0)$

$\mathrm{Next}_{\mathrm{TL\_out}}(q, r, m) \equiv r = q \vee r = q[(q(\mathrm{outq}))\bullet m/\mathrm{outq}]$.

The validity condition for the transmission line module should express the requirement that, for each message m, if the transmission of m satisfies certain minimal conditions (e.g. that m is transmitted repeatedly, without intervening transmission of other messages), then the transmission line module will ensure that m will eventually be delivered according to certain conditions (e.g. m will eventually be delivered repeatedly, without intervening transmission of other messages). Formally,

$\mathrm{Valid}^{\mathrm{TL}} \equiv \quad \mathrm{Rely}^{\mathrm{TL}} \rightarrow \mathrm{Guar}^{\mathrm{TL}}$

$\mathrm{Rely}^{\mathrm{TL}} \equiv \quad \mathbf{true}$

$\mathrm{Guar}^{\mathrm{TL}} \equiv \quad \Box(\forall m \in \mathrm{Values})(\mathrm{Xmit}^{\mathrm{TL}}(m) \rightarrow \Diamond \mathrm{Dlvr}^{\mathrm{TL}}(m))$,

where $\mathrm{Xmit}^{\mathrm{TL}}(m)$ describes the conditions required on the transmission of message m and $\mathrm{Dlvr}^{\mathrm{TL}}(m)$ describes the corresponding conditions according to which m will be delivered. Aside from the requirement that the resulting specification be consistent, there is a reasonable amount of flexibility in the choice of the conditions $\mathrm{Xmit}^{\mathrm{TL}}(m)$ and $\mathrm{Dlvr}^{\mathrm{TL}}(m)$. We will see later that the particular choice of conditions does not significantly affect the proof of correctness of the transmission module implementation, as long as the conditions $\mathrm{Xmit}^{\mathrm{TL}}(m)$ and $\mathrm{Dlvr}^{\mathrm{TL}}(m)$ interact properly with corresponding conditions appearing in the specifications for the send and receive protocol modules. For concreteness though, we make the following definitions:

$\mathrm{Xmit}^{\mathrm{TL}}(m) \equiv \Box\Diamond(\mathbf{Occurs} = \mathrm{TL\_in}{:}m) \wedge$

$\Box(\forall m' \in \mathrm{Values})(\mathbf{Occurs} = \mathrm{TL\_in}{:}m' \rightarrow m' = m)$.

$\mathrm{Dlvr}^{\mathrm{TL}}(m) \equiv \Box\Diamond(\mathbf{Occurs} = \mathrm{TL\_out}{:}m) \wedge$

$\Box(\forall m' \in \mathrm{Values})(\mathbf{Occurs} = \mathrm{TL\_out}{:}m' \rightarrow m' = m)$.

Intuitively, the condition $\mathrm{Xmit}^{\mathrm{TL}}(m)$ states that the message m is transmitted repeatedly, without any transmission of other messages m'. The condition $\mathrm{Dlvr}^{\mathrm{TL}}(m)$ states that the message m is delivered repeatedly, without any delivery of other messages m'.

II.10.4 Specification of the Send Protocol Module

The send protocol module SP interfaces between the sender and the SRTL and RSTL transmission line modules. Its function is to implement one half of the alternating bit transmission protocol. The interface of the send protocol module consists of three kinds of events: *SP_in:m*, which represents the receipt of message *m* from the sender, *SP_pkt_out:p*, which represents the transmission of packet *p* over the unreliable transmission line SRTL, and *SP_ack_in:b*, which represents the receipt of an acknowledgement for sequence number *b* from the unreliable transmission line RSTL.

The state of the send protocol consists of three components: a sequence *inq* of all messages that have ever been received from the sender, a sequence *outq* of all messages that have been acknowledged by the receive protocol module, and a boolean component *sn*, which records the current sequence number. Choosing *outq* to be the sequence of acknowledged messages, rather than the sequence of all messages transmitted to the SRTL transmission line, allows us to obtain a simpler correctness proof than that presented by Hailpern and Owicki [Hailpern80]. In that paper, the use of the actual history of messages transmitted requires the correctness proof to define and reason about certain functions whose purpose is essentially to extract the history of acknowledged messages from the history of all transmitted messages.

Informally, the send protocol module behaves as follows: Occurrence of a *SP_in:m* event causes the message *m* to be appended to *inq*. When there is a message to be sent, and processing of all previous messages has been completed, the message is paired with the current sequence number to form a packet *p*, which is then given to the unreliable transmission line SRTL to be transmitted to the receive protocol module. The send protocol module continues to transmit the packet *p* until an acknowledgement for its current sequence number arrives over the unreliable transmission line RSTL. When a correct acknowledgement arrives, the message acknowledged is appended to *outq*, signifying that it has been successfully delivered, and the current sequence number is complemented.

More precisely, let Values be a finite set of message values, given as a parameter. The interface of the send protocol module is defined as follows:

$$\mathrm{Events}^{SP} = \{\lambda\} + [\mathrm{SP\_in: Values} + \mathrm{SP\_pkt\_out: Pkts} + \mathrm{SP\_ack\_in: Bool}]$$
$$\mathrm{In}^{SP} = \{\lambda\} + [\mathrm{SP\_in: Values} + \mathrm{SP\_ack\_in: Bool}]$$

$\mathrm{Out}^{SP} = \{\lambda\} + [\mathrm{SP\_pkt\_out}: \mathrm{Pkts}]$

$\mathrm{Pkts} = [\mathrm{msg}: \mathrm{Values} \times \mathrm{sn}: \mathrm{Bool}]$,

where Pkts is the set of packets. The state set for the send protocol module is defined by:

$\mathrm{States}^{SP} = [\mathrm{inq}: \mathrm{Seq}[\mathrm{Values}] \times \mathrm{outq}: \mathrm{Seq}[\mathrm{Values}] \times \mathrm{sn}: \mathrm{Bool}]$.

In an initial state, the queue is empty, and the sequence number is false.

$\mathrm{Init}^{SP}(q) \equiv q(\mathrm{inq}) = q(\mathrm{outq}) \wedge q(\mathrm{sn}) = \mathrm{false}$.

The state-transition relation Trans^{SP} is defined as follows:

An *SP_in:m* event can occur at any time, and causes the message *m* to be appended to *inq*.

(SP_in) $\mathrm{Pre}_{SP\_in}(q, e, m) \equiv e = \mathrm{SP\_in}{:}m$

$\mathrm{Next}_{SP\_in}(q, r, m) \equiv r = q[(q(\mathrm{inq}))\bullet m/\mathrm{inq}]$

An *SP_pkt_out:p* event can occur only if there is a message that has been received from the sender but not yet successfully transmitted to the receiver, *p*(msg) is the first such message, and *p*(sn) is the current sequence number. There is no effect on the state.

(SP_pkt_out) $\mathrm{Pre}_{SP\_pkt\_out}(q, e, p) \equiv e = \mathrm{SP\_pkt\_out}{:}p \wedge q(\mathrm{outq}) < q(\mathrm{inq}) \wedge$
$p(\mathrm{msg}) = (q(\mathrm{inq}\text{-}\mathrm{outq}))(0) \wedge p(\mathrm{sn}) = q(\mathrm{sn})$.

$\mathrm{Next}_{SP\_pkt\_out}(q, r, p) \equiv r = q$.

An SP_ack_in event for acknowledgment *b* can occur at any time. If *b* does not match the current sequence number, or if there is no message currently being transmitted, then there is no change in state. If *b* does match the current sequence number and there is a message currently being transmitted, then this indicates that the message has been successfully transmitted. In this case, the current message is appended to *outq*, and the sequence number is complemented.

(SP_ack_in) $\mathrm{Pre}_{SP\_ack\_in}(q, e, b) \equiv e = \mathrm{SP\_ack\_in}{:}b$

$\mathrm{Next}_{SP\_ack\_in}(q, r, b) \equiv ((q(\mathrm{inq}) = q(\mathrm{outq}) \vee b \neq q(\mathrm{sn})) \rightarrow r = q) \wedge$
$((q(\mathrm{outq}) < q(\mathrm{inq}) \wedge b = q(\mathrm{sn})) \rightarrow$
$r = q[\neg(q(\mathrm{sn}))/\mathrm{sn},$
$(q(\mathrm{outq}))\bullet q(\mathrm{inq}\text{-}\mathrm{outq})(0)/\mathrm{outq}])$.

With the validity condition for the send protocol module, we would like to capture the following: If the send protocol can rely on the fact that repeated transmissions of a packet eventually result in the repeated receipt of acknowledgements for that packet, then it guarantees that every message appearing in *inq* will eventually also appear also in *outq*. This requirement is stated in rely-/guarantee-condition form as follows:

$$\begin{aligned}
\mathrm{Valid}^{SP} &\equiv \mathrm{Rely}^{SP} \rightarrow \mathrm{Guar}^{SP}\\
\mathrm{Rely}^{SP} &\equiv \Box(\forall p \in \mathrm{Pkts})(\mathrm{Xmit}^{SP}(p) \rightarrow \Diamond \mathrm{Dlvr}^{SP}(p(\mathrm{sn})))\\
\mathrm{Guar}^{SP} &\equiv \Box(\forall s \in \mathrm{Seq[Values]})(s \leq \mathbf{Now}(\mathrm{inq}) \rightarrow \Diamond(s \leq \mathbf{Now}(\mathrm{outq}))),
\end{aligned}$$

where the formula $\mathrm{Xmit}^{SP}(p)$ is the formalization of the statement: "packet p is transmitted repeatedly, without any transmissions of other packets," and the formula $\mathrm{Dlvr}^{SP}(b)$ is the formalization of "acknowledgements for sequence number b are received repeatedly, without receipt of any other acknowledgements." These formulas must be defined to be compatible (in a way that is made precise by Lemma II.6 below) with the formulas $\mathrm{Xmit}^{TL}(m)$ and $\mathrm{Dlvr}^{TL}(m)$ in the transmission line specification. Thus,

$$\begin{aligned}
\mathrm{Xmit}^{SP}(p) &\equiv \Box\Diamond(\mathbf{Occurs} = \mathrm{SP\_pkt\_out}{:}p) \wedge\\
&\qquad \Box(\forall p' \in \mathrm{Pkts})(\mathbf{Occurs} = \mathrm{SP\_pkt\_out}{:}p' \rightarrow p' = p)\\
\mathrm{Dlvr}^{SP}(b) &\equiv \Box\Diamond(\mathbf{Occurs} = \mathrm{SP\_ack\_in}{:}b) \wedge\\
&\qquad \Box(\forall b' \in \mathrm{Bool})(\mathbf{Occurs} = \mathrm{SP\_ack\_in}{:}b' \rightarrow b' = b).
\end{aligned}$$

II.10.5 Specification of the Receive Protocol Module

The receive protocol module interfaces between the SRTL and RSTL transmission lines, and implements the complementary half of the transmission protocol. It operates as follows: The state of the receive protocol module consists of two sequences, *inq* and *outq*, of messages, and a boolean sequence number *sn*. The sequence *inq* records the history of valid messages (with duplications removed) that have been received from the unreliable transmission line SRTL. The sequence *outq* records the history of messages that have been delivered to the receiver. Initially the sequence number *sn* in the receive protocol module's state matches the sequence number in the state of the send protocol module. The receiver waits for packets to be delivered by the SRTL transmission line. If a received packet has a sequence number that does not match the current sequence number, then it is ignored. If a received packet has a sequence number that matches the current sequence number, then the message is extracted from the packet and placed at the end of *inq*. In addition, the current sequence number is complemented. At any time, the receive protocol module can transmit acknowledgements for the complement of its current sequence number (i.e. for the sequence number of the last valid packet received).

As in the previous specifications, let the finite set Values be given as a parameter. Define the interface of the receive protocol module as follows:

$$\begin{aligned}
\text{Events}^{RP} &= \{\lambda\} + [\text{RP\_pkt\_in: Pkts} + \text{RP\_out: Values} + \text{RP\_ack\_out: Bool}]\\
\text{In}^{RP} &= \{\lambda\} + [\text{RP\_pkt\_in: Pkts}]\\
\text{Out}^{RP} &= \{\lambda\} + [\text{RP\_out: Values} + \text{RP\_ack\_out: Bool}]\\
\text{Pkts} &= [\text{msg: Values} \times \text{sn: Bool}].
\end{aligned}$$

Define the state set by:

$$\text{States}^{RP} = [\text{inq: Seq[Values]} \times \text{outq: Seq[Values]} \times \text{sn: Bool}].$$

In an initial state, both queues are empty, and the sequence number is false.

$$\text{Init}^{RP}(q) \quad \equiv q(\text{inq}) = q(\text{outq}) \wedge q(\text{sn}) = \text{false}$$

The pairs that define the state transition relation Trans^{RP} are given below.

A RP_pkt_in event with packet p can occur at any time. If the sequence number in p does not match the current sequence number, then there is no effect on the state. If the sequence number in p does match the current sequence number, then the message contained in p is appended to *inq*, and the current sequence number is complemented.

(RP_pkt_in) $\text{Pre}_{\text{RP\_pkt\_in}}(q, e, p) \equiv e = \text{RP\_pkt\_in}{:}p$

$$\begin{aligned}
\text{Next}_{\text{RP\_pkt\_in}}(q, r, p) \equiv\ & (p(\text{sn}) \neq q(\text{sn}) \rightarrow r = q) \wedge\\
& (p(\text{sn}) = q(\text{sn}) \rightarrow r = q[\neg q(\text{sn})/\text{sn},\\
& \qquad (q(\text{inq}))\cdot p(\text{msg})/\text{inq}])
\end{aligned}$$

A RP_ack_out event can occur only for the complement of the current sequence number. There is no effect on the state.

(RP_ack_out) $\text{Pre}_{\text{RP\_ack\_out}}(q, e) \equiv e = \text{RP\_ack\_out}{:}(\neg q(\text{sn}))$

$\text{Next}_{\text{RP\_ack\_out}}(q, r) \equiv r = q$

An RP_out event with message m can occur only if there is a message in *inq* that has not yet appeared in *outq*, and m is the first such message. The effect is to append m to *outq*.

(RP_out)

$$\begin{aligned}
\text{Pre}_{\text{RP\_out}}(q, e, m) &\equiv e = \text{RP\_out}{:}m \wedge q(\text{outq}) < q(\text{inq}) \wedge\\
&\qquad m = (q(\text{inq-outq}))(0)\\
\text{Next}_{\text{out}}(q, r, m) &\equiv r = q[(q(\text{outq}))\cdot m/\text{outq}]
\end{aligned}$$

The validity condition for the receive protocol module should capture the following two requirements: (1) If packet p is received repeately, then eventually acknowledgements for the sequence number contained in that packet will be transmitted repeatedly; and (2) Every message that appears in *inq* will eventually appear in *outq*. Formally,

$$\begin{aligned}\mathrm{Valid}^{RP} &\equiv \mathrm{Rely}^{RP} \rightarrow \mathrm{Guar}^{RP}\\ \mathrm{Rely}^{RP} &\equiv \mathbf{true}\\ \mathrm{Guar}^{RP} &\equiv \Box(\forall p \in \mathrm{Pkts})(\mathrm{Dlvr}^{RP}(p) \rightarrow \Diamond \mathrm{Xmit}^{RP}(p(\mathrm{sn}))) \wedge\\ &\quad \Box(\forall s \in \mathrm{Seq}[\mathrm{Values}])(s \leq \mathbf{Now}(\mathrm{inq}) \rightarrow \Diamond(s \leq \mathbf{Now}(\mathrm{outq}))),\end{aligned}$$

where, as in the previous specifications, $\mathrm{Dlvr}^{RP}(p)$ formalizes the statement, "Packet p is received repeately, without any receipt of other packets" and $\mathrm{Xmit}^{RP}(b)$ formalizes the statement, "Acknowledgement b is transmitted repeately, without any transmission of other packets." These formulas are defined as follows:

$$\begin{aligned}\mathrm{Dlvr}^{RP}(p) &\equiv \Box\Diamond(\mathbf{Occurs} = \mathrm{RP\_pkt\_in}{:}p) \wedge\\ &\quad \Box(\forall p' \in \mathrm{Pkts})(\mathbf{Occurs} = \mathrm{RP\_pkt\_in}{:}p' \rightarrow p' = p)\\ \mathrm{Xmit}^{RP}(b) &\equiv \Box\Diamond(\mathbf{Occurs} = \mathrm{RP\_ack\_out}{:}b) \wedge\\ &\quad \Box(\forall b' \in \mathrm{Bool})(\mathbf{Occurs} = \mathrm{RP\_ack\_out}{:}b' \rightarrow b' = b)\end{aligned}$$

II.10.6 The Transmission Module Implementation Algebra

In this section we define the transmission module implementation algebra A^{TMI}. Let the finite set Msgs of message values be given as a parameter. Define

$$\mathrm{Pkts} = [\mathrm{msg}{:}\ \mathrm{Msgs} \times \mathrm{sn}{:}\ \mathrm{Bool}].$$

The index set for the interconnection is the set {SP, RP, SRTL, RSTL}, corresponding to the send protocol, receive protocol, send-protocol-to-receive-protocol transmission line, and receive-protocol-to-send-protocol transmission line component modules. Define the embedded algebras A_{abs}, A_{SP}, A_{RP}, A_{SRTL}, and A_{RSTL} as follows:

A_{abs}: is the message transmission module event/state algebra A^{TM}, with the parameter set Values instantiated as the set Msgs.

A_{SP}: is the send protocol module event/state algebra A^{SP}, with parameter Values instantiated as the set Msgs.

A_{RP}: is the receive protocol module event/state algebra A^{RP}, with parameter Values instantiated as the set Msgs.

A_{SRTL}: is the transmission line module event/state algebra A^{TL}, with parameter Values instantiated as the set Pkts.

A_{RSTL}: is the transmission line module event/state algebra A^{TL}, with parameter Values instantiated as the set Bool.

Let the composite interface for the transmission module interconnection be defined as follows:

$$\begin{aligned}\text{Events}^{TMI} &= \{\lambda\} + [\text{in: Msgs} + \text{out: Msgs} + \text{pkt\_out: Pkts} + \text{pkt\_in: Pkts} + \\ &\qquad \text{ack\_out: Bool} + \text{ack\_in: Bool}] \\ \text{In}^{TMI} &= \{\lambda\} + [\text{in: Msgs}] \\ \text{Out}^{TMI} &= \{\lambda\} + (\text{Events}^{TMI} - \text{In}^{TMI})\end{aligned}$$

Intuitively, events *in*:*m* and *out*:*m* represent, respectively, the receipt of message *m* from the sender and the delivery of message *m* to the receiver. Events *pkt_out*:*p* and *pkt_in*:*p* represent, respectively, the presentation of packet *p* by the send protocol module to the SRTL transmission line and the receipt of packet *p* by the receive protocol module from the SRTL transmission line. Events *ack_out*:*b* and *ack_in*:*b* represent, respectively, the presentation of acknowledgement *b* by the receive protocol module to the RSTL transmission line and the receipt of acknowledgement *b* by the send protocol module from the RSTL transmission line.

Define the abstraction map α^{TMI}, and the decomposition map $\underline{\delta}^{TMI}$ as follows:

$$\begin{aligned}\alpha^{TMI}(e) &= \text{TM\_in:}m && \text{if } e = \text{in:}m \\ &= \text{TM\_out:}m && \text{if } e = \text{out:}m \\ &= \lambda && \text{otherwise.}\end{aligned}$$

$$\begin{aligned}\delta^{TMI}_{SP}(e) &= \text{SP\_in:}m && \text{if } e = \text{in:}m \\ &= \text{SP\_pkt\_out:}p && \text{if } e = \text{pkt\_out:}p \\ &= \text{SP\_ack\_in:}b && \text{if } e = \text{ack\_in:}b \\ &= \lambda && \text{otherwise.}\end{aligned}$$

$$\begin{aligned}\delta^{TMI}_{RP}(e) &= \text{RP\_out:}m && \text{if } e = \text{out:}m \\ &= \text{RP\_pkt\_in:}p && \text{if } e = \text{pkt\_in:}p \\ &= \text{RP\_ack\_out:}b && \text{if } e = \text{ack\_out:}b \\ &= \lambda && \text{otherwise.}\end{aligned}$$

$$\begin{aligned}\delta^{TMI}_{SRTL}(e) &= \text{TL\_in:}p && \text{if } e = \text{pkt\_out:}p \\ &= \text{TL\_out:}p && \text{if } e = \text{pkt\_in:}p \\ &= \lambda && \text{otherwise.}\end{aligned}$$

$$\delta^{\mathrm{TMI}}_{\mathrm{RSTL}}(e) = \mathrm{TL\_in}{:}b \qquad \text{if } e = \mathrm{ack\_out}{:}p$$
$$= \mathrm{TL\_out}{:}b \qquad \text{if } e = \mathrm{ack\_in}{:}p$$
$$= \lambda \qquad \text{otherwise.}$$

II.10.7 Proof of Correctness

In this section we prove the correctness of the implementation $\langle \mathcal{I}_{A^{\mathrm{TMI}}}, S_{\mathrm{abs}}, \langle S_i \rangle_{i \in \{\mathrm{SP,RP,RSTL,SRTL}\}} \rangle$, where S_{abs} is defined by $\langle A_{\mathrm{abs}}, \mathrm{Valid}^{\mathrm{TM}} \rangle$, and S_{SP}, S_{RP}, S_{RSTL}, S_{SRTL} are defined by $\langle A_{\mathrm{SP}}, \mathrm{Valid}^{\mathrm{SP}} \rangle$, $\langle A_{\mathrm{RP}}, \mathrm{Valid}^{\mathrm{RP}} \rangle$, $\langle A_{\mathrm{RSTL}}, \mathrm{Valid}^{\mathrm{TL}} \rangle$, and $\langle A_{\mathrm{SRTL}}, \mathrm{Valid}^{\mathrm{TL}} \rangle$, respectively.

Invariance

The correctness of the transmission module implementation depends only on the invariance of the following:

(1) $q_{\mathrm{abs}}(\mathrm{inq}) = q_{\mathrm{SP}}(\mathrm{inq}) \wedge q_{\mathrm{abs}}(\mathrm{outq}) = q_{\mathrm{RP}}(\mathrm{outq})$

(2) $q_{\mathrm{SP}}(\mathrm{outq}) \leq q_{\mathrm{RP}}(\mathrm{inq}) \leq q_{\mathrm{SP}}(\mathrm{inq})$.

Condition (1) is the abstraction relation $\mathrm{Abs}^{\mathrm{TMI}}(q)$, and states that the abstract transmission module's *inq* is identical to the *inq* for the send protocol module, and that the abstract transmission module's *outq* is identical to the *outq* for the receive protocol module. Condition (2) is Lemma II.4 below, and says that the receive protocol module's *inq* is always an extension of the send protocol module's *outq* and a prefix of the send protocol module's *inq*.

Condition (2) is not inductive as stated, and must be strengthened to permit an inductive proof of invariance. We therefore define the implementation invariant $\mathrm{Inv}^{\mathrm{TMI}}(q)$ by

$$\mathrm{Inv}^{\mathrm{TMI}}(q) \equiv \mathrm{Rep}^{\mathrm{TMI}}(q) \wedge \mathrm{Abs}^{\mathrm{TMI}}(q),$$

where $\mathrm{Rep}^{\mathrm{TMI}}(q)$ is the representation invariant and $\mathrm{Abs}^{\mathrm{TMI}}(q)$ is the abstraction relation. The abstraction relation is:

$$\mathrm{Abs}^{\mathrm{TMI}}(q) \equiv q_{\mathrm{abs}}(\mathrm{inq}) = q_{\mathrm{SP}}(\mathrm{inq}) \wedge q_{\mathrm{abs}}(\mathrm{outq}) = q_{\mathrm{RP}}(\mathrm{outq}).$$

The representation invariant $\mathrm{Rep}^{\mathrm{TMI}}(q)$ is defined as follows:

$$\mathrm{Rep}^{\mathrm{TMI}}(q) \equiv \mathrm{Queue}(q) \wedge (\mathrm{Start}(q) \vee \mathrm{Send}(q) \vee \mathrm{Flip}(q) \vee \mathrm{Ack}(q)),$$

where

$$\mathrm{Queue}(q) \equiv q_{\mathrm{SP}}(\mathrm{inq}) \geq q_{\mathrm{SP}}(\mathrm{outq}) \wedge q_{\mathrm{RP}}(\mathrm{inq}) \geq q_{\mathrm{RP}}(\mathrm{outq})$$

and the formal definitions of Start, Send, Flip, and Ack will be given below. This invariant says that, at any instant of time, the histories *inq* and *outq* for the send and receive protocol modules satisfy certain prefix relationships captured by the predicate Queue. In addition, the transmission system is always in one of four kinds of states, corresponding to the four predicates Start(q), Send(q), Flip(q), and Ack(q). The situations covered by these four predicates, and how they evolve during execution, will now be described.

In a state that satisfies Start, the send and receive protocol modules have the same sequence number, the send protocol module's *outq* and the receive protocol module's *inq* are identical, and no new packets or acknowledgements are currently in transit over the transmission lines. The predicate Start is satisfied by all initial states.

States satisfying Start give rise to states satisfying Send when there is an unprocessed message at the send protocol module that has been output to (but possibly lost by) the transmission line RSTL. In a state that satisfies Send, the send and receive protocol modules have the same current sequence number, the *outq* of the send protocol module and the *inq* of the receive protocol module are identical, there is an unprocessed message at the send protocol module, there may be packets containing this message in transit over the transmission line SRTL, and there are no new acknowledgements in transit over RSTL.

States satisfying Send give rise to states satisfying Flip when the first packet containing an unprocessed message arrives at the receive protocol module. In a state that satisfies Flip, the send and receive protocol modules have complementary current sequence numbers, the *inq* of the receive protocol module is equal to the *outq* of the send protocol module with the newly arrived message appended, and all packets in transit over SRTL or acknowledgements in transit over RSTL are old in the sense that they are for a sequence number that is not the one currently expected by the send protocol module.

States satisfying Flip give rise to states satisfying Ack when the first acknowledgement for the newly arrived packet is transmitted over RSTL. In a state satisfying Ack, the send and receive protocol modules have complementary current sequence numbers, the *inq* of the receive protocol module is equal to the *outq* of the send protocol module with the still-unacknowledged message appended, all packets in transit over SRTL are old, but there may be new acknowledgements in transit over RSTL.

To complete the cycle, states satisfying Ack give rise to states satisfying Start when the first new acknowledgement is received by the send protocol module.

For the formal statement of these predicates, it is convenient to define some auxiliary predicates, which describe possible states of the transmission lines SRTL and RSTL.

The predicate SRTL_old is true of a state iff all packets in the SRTL transmission line are old, in the sense that they are for the opposite sequence number than the one currently expected by the receive protocol module.

$$\text{SRTL\_old}(q) \equiv (\forall n < |q_{\text{SRTL}}(\text{inq-outq})|)(q_{\text{SRTL}}(\text{inq-outq})(n)(\text{sn}) \neq q_{\text{RP}}(\text{sn}))$$

Similarly, the predicate RSTL_old is true of a state iff all acknowledgements in the RSTL transmission line are old, in the sense that they are for the opposite sequence number than the one expected by the send protocol module.

$$\text{RSTL\_old}(q) \equiv (\forall n < |q_{\text{RSTL}}(\text{inq-outq})|)(q_{\text{RSTL}}(\text{inq-outq})(n)(\text{sn}) \neq q_{\text{SP}}(\text{sn}))$$

The predicate SRTL_new is true of a state iff the SRTL transmission line queue consists of a (possibly empty) sequence of old packets, followed by a (possibly empty) sequence of new packets, each of which contains the first unprocessed message held by the send protocol module.

$$\begin{aligned}\text{SRTL\_new}(q) \equiv\ & (\exists m \leq |q_{\text{SRTL}}(\text{inq-outq})|)(\forall n < |q_{\text{SRTL}}(\text{inq-outq})|)\\ & ((n < m \rightarrow q_{\text{SRTL}}(\text{inq-outq})(n)(\text{sn}) \neq q_{\text{RP}}(\text{sn})) \wedge\\ & (n \geq m \rightarrow q_{\text{SRTL}}(\text{inq-outq})(n) =\\ & \qquad \langle \text{msg: } q_{\text{SP}}(\text{inq-outq})(0), \text{ sn: } q_{\text{RP}}(\text{sn})\rangle))\end{aligned}$$

Similarly, the predicate RSTL_new is true of a state iff the RSTL transmission line queue currently consists of a (possibly empty) sequence of old acknowledgments, followed by a (possibly empty) sequence of new acknowledgements.

$$\begin{aligned}\text{RSTL\_new}(q) \equiv\ & (\exists m \leq |q_{\text{RSTL}}(\text{inq-outq})|)(\forall n < |q_{\text{RSTL}}(\text{inq-outq})|)\\ & ((n < m \rightarrow q_{\text{RSTL}}(\text{inq-outq})(n) \neq q_{\text{SP}}(\text{sn})) \wedge\\ & (n \geq m \rightarrow q_{\text{RSTL}}(\text{inq-outq})(n) = q_{\text{SP}}(\text{sn}))).\end{aligned}$$

The formal definitions of the predicates Start, Send, Flip, and Ack are as follows:

$$\begin{aligned}
\text{Start}(q) &\equiv q_{SP}(\text{sn}) = q_{RP}(\text{sn}) \wedge q_{SP}(\text{outq}) = q_{RP}(\text{inq}) \wedge \\
&\quad \text{SRTL\_old}(q) \wedge \text{RSTL\_old}(q) \\
\text{Send}(q) &\equiv q_{SP}(\text{sn}) = q_{RP}(\text{sn}) \wedge q_{SP}(\text{outq}) < q_{SP}(\text{inq}) \wedge q_{SP}(\text{outq}) = q_{RP}(\text{inq}) \wedge \\
&\quad \text{SRTL\_new}(q) \wedge \text{RSTL\_old}(q) \\
\text{Flip}(q) &\equiv q_{SP}(\text{sn}) \neq q_{RP}(\text{sn}) \wedge q_{SP}(\text{outq}) < q_{SP}(\text{inq}) \wedge \\
&\quad (q_{SP}(\text{outq}))q_{SP}(\text{inq-outq})(0) = q_{RP}(\text{inq}) \wedge \\
&\quad \text{SRTL\_old}(q) \wedge \text{RSTL\_old}(q) \\
\text{Ack}(q) &\equiv q_{SP}(\text{sn}) \neq q_{RP}(\text{sn}) \wedge q_{SP}(\text{outq}) < q_{SP}(\text{inq}) \wedge \\
&\quad (q_{SP}(\text{outq}))q_{SP}(\text{inq-outq})(0) = q_{RP}(\text{inq}) \wedge \\
&\quad \text{SRTL\_old}(q) \wedge \text{RSTL\_new}(q).
\end{aligned}$$

We now consider the proof that $\text{Inv}^{\text{TMI}}(q)$ is invariant.

(*Basis*): $\models (\forall q \in \text{States}^{\text{TMI}})(\text{Init}^{\text{TMI}}(q) \rightarrow \text{Inv}^{\text{TMI}}(q))$.
If q is an initial state then all queues are empty and the *sn* components of the state of both the send protocol module and the receive protocol module have value false. It is easily verified from this that $\text{Abs}^{\text{TMI}}(q) \wedge \text{Queue}(q) \wedge \text{Start}(q)$ holds.

(*Induction*): $\models (\forall q,r \in \text{States}^{\text{TMI}}, e \in \text{Events}^{\text{TMI}})(\text{Trans}^{\text{TMI}}(q, e, r) \rightarrow (\text{Inv}^{\text{TMI}}(q) \rightarrow \text{Inv}^{\text{TMI}}(r)))$.
Suppose that $\text{Inv}^{\text{TMI}}(q)$ holds and that $\text{Trans}^{\text{TMI}}(q, e, r)$ holds.

We first examine the problem of showing that $\text{Abs}^{\text{TMI}}(r)$ holds. $\text{Abs}^{\text{TMI}}(r)$ is easily seen to be true, since the only events that affect components of the state upon which Abs^{TMI} depends are the events *in:m* and *out:m*. Comparison of the definitions of Trans^{TM}, Trans^{SP}, and Trans^{RP} shows that the events *in:m* and *out:m* maintain the desired correspondence between the abstract module state and the states of the send and receive protocol modules.

To see that Queue(r) holds, note that the definitions of Trans^{SP} and Trans^{RP} imply that the *inq* and *outq* components of the states of the send and receive protocol modules can only change in one of the following two ways:

- A new message is appended to the end of *inq*.
- The first element of *inq* – *outq* is appended to the end of *outq*.

Neither of these two kinds of changes can cause *outq* not to be a prefix of *inq*, and thus Queue(r) must hold.

To show that Start(r) $\vee$ Send(r) $\vee$ Flip(r) $\vee$ Ack(r) holds, we claim that all events preserve the truth of the predicates Start, Send, Flip, and Ack, except in the following cases:

- If Start(q) is true and e = *pkt_out:p*, then Send(r) is true.
- If Send(q) is true and e = *pkt_in:p*, with p(sn) = q_{RP}(sn), then Flip(r) is true.
- If Flip(q) is true and e = *ack_out:b*, then Ack(r) is true.
- If Ack(q) is true and e = *ack_in:b*, with b = q_{SP}(sn), then Start(r) is true.

It is a straightforward, but tedious process to verify the truth of this claim by exhaustive case analysis.

The following consequence of the invariance of $\mathrm{Inv}^{TMI}(q)$ is the crucial fact used in the maximality and validity proofs below.

Lemma II.4 - The following are invariant for the transmission module implementation:

(a) $q_{SP}(\mathrm{outq}) \leq q_{RP}(\mathrm{inq})$

(b) $q_{RP}(\mathrm{inq}) \leq q_{SP}(\mathrm{inq})$

Proof - The invariance of Start(q) $\vee$ Send(q) $\vee$ Flip(q) $\vee$ Ack(q) implies the invariance of

(1) $$q_{RP}(\mathrm{inq}) = q_{SP}(\mathrm{outq}) \vee$$
$$(q_{SP}(\mathrm{inq}) > q_{SP}(\mathrm{outq}) \wedge q_{RP}(\mathrm{inq}) = (q_{SP}(\mathrm{outq}))q_{SP}(\mathrm{inq}\text{-}\mathrm{outq})(0)).$$

Suppose the first disjunct of (1) holds, that is $q_{SP}(\mathrm{outq}) = q_{RP}(\mathrm{inq})$. Then (a) is immediate. The invariance of Queue(q) implies that $q_{SP}(\mathrm{outq}) \leq q_{SP}(\mathrm{inq})$, thus yielding (b). Now suppose that the second disjunct of (1) holds. It is a fact about finite sequences that if s, s' are finite sequences, and $s > s'$, then $s \geq s'm$, where $m = (s - s')(0)$. This fact permits us to conclude, from the second disjunct of (1), that $q_{SP}(\mathrm{inq}) \geq (q_{SP}(\mathrm{outq}))q_{SP}(\mathrm{inq}\text{-}\mathrm{outq})(0) = q_{RP}(\mathrm{inq}) \geq q_{SP}(\mathrm{outq})$, yielding (a) and (b). ∎

Maximality

The maximality verification condition is:

$$\models (\forall q \in \mathrm{States}^{TMI}, e \in \mathrm{Events}^{TMI})(\mathrm{Inv}^{TMI}(q) \wedge \mathrm{Enabled}_{SP}(q, e) \wedge \mathrm{Enabled}_{RP}(q, e) \wedge$$
$$\mathrm{Enabled}_{SRTL}(q, e) \wedge \mathrm{Enabled}_{RSTL}(q, e)$$
$$\rightarrow \mathrm{Enabled}_{abs}(q, e)).$$

Examination of the definition of Trans^{TM} shows that $\mathrm{Enabled}_{abs}(q, e)$ is identically true unless e = *out:m*. Thus it suffices to show that, for all $q \in \mathrm{States}^{TMI}$ and all $m \in$ Msgs, if $\mathrm{Inv}^{TMI}(q)$ and $\mathrm{Enabled}_{RP}(q, \mathrm{out}{:}m)$ hold, then $\mathrm{Enabled}_{abs}(q, \mathrm{out}{:}m)$ holds as well.

Suppose now that $\mathrm{Inv}^{\mathrm{TMI}}(q)$ and $\mathrm{Enabled}_{\mathrm{RP}}(q, \mathrm{out}{:}m)$ hold. It suffices to show that

$$(1)\ q_{\mathrm{SP}}(\mathrm{inq}) > q_{\mathrm{RP}}(\mathrm{outq})$$

holds, for then the assumption that $\mathrm{Inv}^{\mathrm{TMI}}(q)$ (and hence $\mathrm{Abs}^{\mathrm{TMI}}(q)$) holds implies that $q_{\mathrm{abs}}(\mathrm{inq}) > q_{\mathrm{abs}}(\mathrm{outq})$ holds, which in turn implies that $\mathrm{Enabled}_{\mathrm{abs}}(q, \mathrm{out}{:}m)$ holds.

By definition of $\mathrm{Enabled}_{\mathrm{RP}}(q, \mathrm{out}{:}m)$, we know that

$$(2)\ q_{\mathrm{RP}}(\mathrm{outq}) < q_{\mathrm{RP}}(\mathrm{inq}) \wedge q_{\mathrm{RP}}(\mathrm{inq}\text{-}\mathrm{outq})(0) = m$$

holds. Informally, if $e = out{:}m$, then m must be the first message in the receive protocol module's *inq*, that has not yet been transferred to its *outq*. The truth of (1) follows from (2) and Lemma II.4 (b). ▮

Validity

To prove that the validity verification condition holds for the transmission module implementation, we use Corollary I.4. We use the well-founded partial ordering $<$ on the set {SP, RP, RSTL, SRTL} that includes exactly the pairs SRTL $<$ SP, RP $<$ SP, and RSTL $<$ SP. Under this ordering, hypotheses (1) and (2) of Corollary I.4 are as follows:

(TMI1) $\mathrm{Comp}^{\mathrm{TMI}} \models [\![\mathrm{Guar}^{\mathrm{SP}}]\!]_{\mathrm{SP}} \wedge [\![\mathrm{Guar}^{\mathrm{RP}}]\!]_{\mathrm{RP}} \wedge [\![\mathrm{Guar}^{\mathrm{TL}}]\!]_{\mathrm{SRTL}} \wedge [\![\mathrm{Guar}^{\mathrm{TL}}]\!]_{\mathrm{RSTL}} \rightarrow [\![\mathrm{Guar}^{\mathrm{TM}}]\!]_{\mathrm{abs}}$

(TMI2) $\mathrm{Comp}^{\mathrm{TMI}} \models [\![\mathrm{Guar}^{\mathrm{TL}}]\!]_{\mathrm{SRTL}} \wedge [\![\mathrm{Guar}^{\mathrm{RP}}]\!]_{\mathrm{RP}} \wedge [\![\mathrm{Guar}^{\mathrm{TL}}]\!]_{\mathrm{RSTL}} \rightarrow [\![\mathrm{Rely}^{\mathrm{SP}}]\!]_{\mathrm{SP}}$.

These two conditions capture abstractly the important relationships between the validity conditions of the various modules.

We now prove that (TMI1) and (TMI2) are consequences of the module specifications.

Lemma II.5 - Condition (TMI1) holds for the transmission module implementation.

Proof - Assume $\mathrm{Comp}^{\mathrm{TMI}}$ and $[\![\mathrm{Guar}^{\mathrm{SP}}]\!]_{\mathrm{SP}}$ and $[\![\mathrm{Guar}^{\mathrm{RP}}]\!]_{\mathrm{RP}}$. Using the definition of $[\![\mathrm{Guar}^{\mathrm{SP}}]\!]_{\mathrm{SP}}$, we have

$$\Box(\forall s\in\mathrm{Seq}[\mathrm{Msgs}])(s \leq \mathbf{Now}_{\mathrm{SP}}(\mathrm{inq}) \rightarrow \Diamond(s \leq \mathbf{Now}_{\mathrm{SP}}(\mathrm{outq}))).$$

From this and Lemma II.4 (a), we obtain

$$\Box(\forall s\in\mathrm{Seq}[\mathrm{Msgs}])(s \leq \mathbf{Now}_{\mathrm{SP}}(\mathrm{inq}) \rightarrow \Diamond(s \leq \mathbf{Now}_{\mathrm{RP}}(\mathrm{inq}))).$$

Using the assumption $[\![\mathrm{Guar}^{\mathrm{RP}}]\!]_{\mathrm{RP}}$ gives

$$\Box(\forall s\in\mathrm{Seq}[\mathrm{Msgs}])(s \leq \mathbf{Now}_{\mathrm{SP}}(\mathrm{inq}) \rightarrow \Diamond(s \leq \mathbf{Now}_{\mathrm{RP}}(\mathrm{outq}))).$$

From an application of the invariance of $\mathrm{Abs}^{\mathrm{TMI}}$, we conclude

$\Box(\forall s \in \mathrm{Seq[Msgs]})(s \leq \mathbf{Now}_{\mathrm{abs}}(\mathrm{inq}) \rightarrow \Diamond(s \leq \mathbf{Now}_{\mathrm{abs}}(\mathrm{outq})))$. ▮

The proof of condition (TMI2) makes use of the following lemma, which expresses the principle that guided our choices for the definitions of the various Xmit and Dlvr formulas in the specifications above.

Lemma II.6 - The following hold for the transmission module implementation:

$\models [\![\mathrm{Xmit}^{SP}(p)]\!]_{SP} \leftrightarrow [\![\mathrm{Xmit}^{TL}(p)]\!]_{SRTL}$

$\models [\![\mathrm{Dlvr}^{SP}(b)]\!]_{SP} \leftrightarrow [\![\mathrm{Dlvr}^{TL}(b)]\!]_{RSTL}$

$\models [\![\mathrm{Xmit}^{RP}(b)]\!]_{RP} \leftrightarrow [\![\mathrm{Xmit}^{TL}(b)]\!]_{RSTL}$

$\models [\![\mathrm{Dlvr}^{RP}(p)]\!]_{RP} \leftrightarrow [\![\mathrm{Dlvr}^{TL}(p)]\!]_{SRTL}$.

Proof - Straightforward from the module specifications and the definition of the decomposition map $\underline{\delta}^{TMI}$. ▮

Lemma II.7 - Condition (TMI2) holds for the transmission module implementation.

Proof - Suppose that Comp^{TMI}, $[\![\mathrm{Valid}^{TL}]\!]_{SRTL}$, $[\![\mathrm{Valid}^{RP}]\!]_{RP}$, and $[\![\mathrm{Valid}^{TL}]\!]_{RSTL}$ hold. Suppose, to obtain a contradiction, that $\neg[\![\mathrm{Rely}^{SP}]\!]_{SP}$ holds. From the definition of $[\![\mathrm{Rely}^{SP}]\!]_{SP}$, we know that

(1) $\Diamond(\exists p \in \mathrm{Pkts})([\![\mathrm{Xmit}^{SP}(p)]\!]_{SP} \wedge \Box\neg[\![\mathrm{Dlvr}^{SP}(p(\mathrm{sn}))]\!]_{SP})$

holds. That is, eventually a point is reached after which the packet p is transmitted infinitely often, without intervening transmission of other packets, but infinitely many acknowledgements for the sequence number contained in p are not received by the send protocol module. From (1) and Lemma II.6 we infer

(2) $\Diamond(\exists p \in \mathrm{Pkts})([\![\mathrm{Xmit}^{TL}(p)]\!]_{SRTL} \wedge \Box\neg[\![\mathrm{Dlvr}^{SP}(p(\mathrm{sn}))]\!]_{SP})$.

From (2) and the assumption that $[\![\mathrm{Valid}^{TL}]\!]_{SRTL}$ holds, we deduce

(3) $\Diamond(\exists p \in \mathrm{Pkts})([\![\mathrm{Dlvr}^{TL}(p)]\!]_{SRTL} \wedge \Box\neg[\![\mathrm{Dlvr}^{SP}(p(\mathrm{sn}))]\!]_{SP})$.

That is, packet p is delivered infinitely often to the receive protocol module, without intervening delivery of other packets, but infinitely many acknowledgements for the sequence number contained in p are not received by the send protocol module. From (3), another application of Lemma II.6 shows

(4) $\Diamond(\exists p \in \mathrm{Pkts})([\![\mathrm{Dlvr}^{RP}(p)]\!]_{RP} \wedge \Box\neg[\![\mathrm{Dlvr}^{SP}(p(\mathrm{sn}))]\!]_{SP})$.

From this, an application of $[\![\mathrm{Valid}^{RP}]\!]_{RP}$ shows

(5) $\Diamond(\exists p \in \mathrm{Pkts})([\![\mathrm{Xmit}^{RP}(p(\mathrm{sn}))]\!]_{RP} \wedge \Box\neg[\![\mathrm{Dlvr}^{SP}(p(\mathrm{sn}))]\!]_{SP})$.

That is, an acknowledgement for the sequence number contained in packet p is repeatedly transmitted by the receive protocol module, but infinitely many acknowledgements for p are not received by the send protocol module. Applying Lemma II.6, $[\![\mathrm{Valid}^{\mathrm{TL}}]\!]_{\mathrm{RSTL}}$, and Lemma II.6 again, shows that

(6) $\Diamond(\exists p \in \mathrm{Pkts})([\![\mathrm{Dlvr}^{\mathrm{SP}}(p(\mathrm{sn}))]\!]_{\mathrm{SP}} \wedge \Box\neg[\![\mathrm{Dlvr}^{\mathrm{SP}}(p(\mathrm{sn}))]\!]_{\mathrm{SP}})$.

This is a contradiction, and we conclude that (TMI2) must hold. ∎

Appendix III - Index of Definitions