Hierarchical Correctness Proofs for Distributed Algorithms

by

Mark R. Tuttle
B.S., University of Nebraska–Lincoln
(1984)

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

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

April 1987

Signature of Author _____
Department of Electrical Engineering and Computer Science
April 3, 1987

Certified by _____
Nancy A. Lynch
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Hierarchical Correctness Proofs for Distributed Algorithms

by

Mark R. Tuttle

Submitted to the
Department of Electrical Engineering and Computer Science
on April 3, 1987, in partial fulfillment of the requirements
for the Degree of
Master of Science in Computer Science

Abstract

This thesis introduces a new model for distributed computation in asynchronous networks, the *input-output automaton*. This simple, powerful model captures in a novel way the game-theoretic interaction between a system and its environment, and allows fundamental properties of distributed computation such as fair computation to be naturally expressed. Furthermore, this model can be used to construct modular, hierarchical correctness proofs of distributed algorithms. This thesis defines the input-output automaton model, and presents an interesting example of how this model can be used to construct such proofs.

Thesis supervisor: Nancy A. Lynch
Title: Ellen Swallow Richards Professor of Computer Science and Engineering

Acknowledgments

Nancy Lynch has been everything a thesis advisor could be expected to be. Her courage and fortitude when plowing through difficult drafts is well-known among her graduate students. She is able to rekindle enthusiasm even in the most discouraging situations. In a more playful moment, Nancy was once compared with an excited electron moving from person to person, leaving a portion of her energy with everyone she meets.

Michael Merritt has also been of great support and assistance. Much of the preliminary work for this thesis (including the conception of an input-output automaton) was done concurrently with work by Nancy and Michael in [LM86], and their work has had a profound impact on this thesis.

Brian Coan, Alan Fekete, Ken Goldman, Yoram Moses, and Jennifer Welch have also made significant contributions to this thesis. Discussions with all of them, and the experience of Alan, Ken, and Jennifer as they use input-output automata in their work, have motivated a number of improvements in this work. I am particularly grateful to Jennifer for her detailed reading of an earlier draft of this thesis. In addition to their intellectual assistance, I greatly value their friendship.

Most valued of all is the friendship of my wife, Margaret, whom I wish to thank for her love, support, and knowledge of spelling and grammar.

Contents

Chapter 1

Introduction

A major obstacle to progress in the field of distributed computation is that many of the important algorithms, especially communications algorithms, seem to be too complex for rigorous understanding. Although the designers of these algorithms are often able to convey an intuitive understanding of how their algorithms work, it is often difficult to make this intuition formal and precise. When these algorithms are rigorously analyzed, the work is generally carried out at a very low level of abstraction, involving messages and local process variables. Reasoning precisely about the interaction between these messages and variables can be extremely difficult, and the resulting proofs of correctness are generally quite difficult to understand.

An indication that the situation is not completely hopeless is the fact that the designers *are* able to give high-level, although informal, descriptions of the key ideas behind their algorithms. For instance, the distributed minimum spanning tree algorithm of [GHS83] can be interpreted as several familiar manipulations of a graph. What is needed is a way of formalizing these high-level ideas, and incorporating them into a proof of the detailed algorithm's correctness.

One promising approach is to begin by constructing a high-level description of the algorithm. This description could *itself* be an algorithm in which high-level data structures (such as graphs) serve as states, and process actions manipulate these data structures. This algorithm could then be proven correct using rigorous versions of the high-level, intuitive arguments given by the algorithm's designers. Successive refinements of this algorithm could then be defined at successively lower levels of detail, and each shown (rigorously) to simulate the preceding algorithm. Ideally, this approach would allow us to use in the proof of simulation any property that has already been proven for preceding levels. In this way, the high-level intuition used to explain the algorithm would become part of a rigorous proof of the full algorithm's correctness.

Two years ago, we began to consider this approach for a fairly simple but interesting algorithm for resource allocation in an asynchronous network, an algorithm originally

suggested by Schönhage in [Sch80]. Correctness conditions for this resource arbitration problem include both safety and liveness conditions:[1] the *mutual exclusion* condition that at most one user is using the resource at any given time; and the *no lockout* condition that if every user holding the resource eventually returns the resource to the arbiter, then the arbiter will eventually grant the resource to every requesting user. The algorithm can be described at three levels of abstraction. At the top level is a simple, set-theoretic statement of the problem, itself described as an algorithm. At the second level is a graph-theoretic description of the arbiter, and how it moves the resource around the network. At the third and lowest level is a distributed implementation of the arbiter, describing in terms of messages and local process variables the protocol individual processors must follow.

It soon became apparent, however, that traditional models and proof techniques (see [OG76], [LS84b], and [Hoa85], for example) are not adequate to describe interesting aspects of the problem statement, algorithm, and correctness proofs. In particular, while the problem seems most naturally formulated in terms of the game-theoretic interaction between the users of the arbiter and the arbiter itself, these models require that the problem be formulated in terms of system states, and do not capture this game-theoretic aspect of the problem in a natural way. Furthermore, the interaction between the users and the arbiter clearly distinguishes the arbiter's input actions from its output actions. Input to the arbiter (a request for the resource) can occur at any time, regardless of whether the arbiter is in a position to grant the resource. Output (the granting of requests) occurs only under the control of the arbiter. This notion of control, the notion that one system component may completely determine when a particular action is performed, is not easily expressed in these models. We note that satisfaction of the no lockout condition requires that the arbiter be given "fair turns" to produce output, rather than simply being overwhelmed by a flood of input. The ability to express this notion of "fair turns" depends heavily on the ability to express the notion of one process controlling the performance of an action.

We were therefore led to the development of a new model of distributed computation in asynchronous systems, the *input-output automaton*. This model is based on (possibly infinite-state) nondeterministic automata. Automaton transitions are labeled with the names of process actions they represent. These actions are partitioned into sets of input and output actions, as well as internal actions representing internal process actions. Input actions have the unique property of being enabled from every state; that is, for every input action there is a transition labeled with this action from every state. In other words, the system must be able to accept any input at any time. Thus, a very strong distinction is

[1] Informally, properties required of a program can be partitioned into *safety properties* and *liveness properties*. A safety property (such as mutual exclusion [Dij65]) says that nothing "bad" will ever happen, and a liveness property (such as termination) says that something "good" will eventually happen. Alternatively, safety properties describe allowed behavior, and liveness properties describe required behavior. Alpern and Schneider give formal definitions of safety and liveness in [AS86] in terms of Büchi automata.

made between actions locally-controlled by the system (output and internal actions) and actions controlled by the system's external environment (input actions). This distinction captures the game-theoretic interaction between the system and its environment alluded to above, and gives our model an event-driven flavor characteristic of many asynchronous distributed algorithms.

In order to construct models of complex systems from models of simpler system components, we define a simple notion of automaton composition. Loosely speaking, the composition of a collection of automata is their Cartesian product, with a state of the composition being a tuple of states from the component automata, one from each component. In order to model communication, we require that automata synchronize the performance of common (shared) actions. If π is an output action of A and an input action of B, then performance of π by both automata models communication from A to B. With simple syntactic restrictions on the composition of automata, we ensure that composition preserves the notion of control mentioned above: No system component may block the performance of an output action by any other component.

Since automata are able to receive every input in every state, it is possible for an automaton to be flooded with input without having the opportunity to perform actions required in response to the input received. The satisfaction of most interesting liveness conditions, however, requires that this does not happen. The notion of fair computation therefore plays a fundamental role in our model. Informally, a computation of a system is said to be fair if every system component is always eventually given the chance to take a step. Since an automaton may model an entire system as well as a single system component, it is necessary to retain certain information about the structure of the system being modeled. In particular, it is necessary to retain information about which actions are controlled by the same system component. With this information it is possible to determine from a given system behavior whether each system component has been given the chance to make computational progress infinitely often. We therefore associate with every automaton a partition of its locally-controlled actions (i.e., its internal and output actions). The interpretation of this partition is that each class consists of the locally-controlled actions of one system component. With this partition, we are able to define a simple notion of fair computation in our model.

Since our model concentrates on the input-output interaction between a system and its environment (rather than system states), our notion of a problem to be solved is a collection of system behaviors (sequences of input and output actions) considered acceptable (rather than conditions on system states). An automaton may be considered a solution to such a problem if every behavior exhibited by the automaton is contained in this set of acceptable behaviors. The automaton solves the problem in the sense that any correctness condition satisfied by each behavior in this set is satisfied by each behavior of the automaton. As previously mentioned, however, fair computation is crucial to the satisfaction of most interesting liveness conditions. We therefore require only that the *fair* behaviors of an automaton solving the problem be contained in the set of acceptable

behaviors. We note that it is easy to fall into trivial correctness definitions, allowing trivial or uninteresting solutions to a problem. Our condition that an automaton be required to accept any input in any state, together with our notion of fairness, avoids this problem. The requirement that input be constantly enabled ensures that our solutions are able to respond to all patterns of input. The use of fairness ensures that the correctness of an solution will be judged only by those behaviors in which the system is actually given the chance to make progress.

Our simple correctness condition, the requirement that the fair behaviors of an automaton be contained in some set of acceptable behaviors, is not a new style of correctness conditions. It can be found, for instance, in the work of Lynch and Fischer in [LF81], and is similar to Hoare's notion of specification satisfaction in [Hoa85]. The simplicity of such correctness conditions do, however, lend a uniform structure to correctness proofs in our model. Recall that our notion of a well-structured correctness proof involves a sequence of models $M_1, \ldots, M_n$, each modeling an algorithm at successively lower levels of detail. The proof of the algorithm's correctness involves showing that each model "simulates" the previous model in the sequence. That is, that the set of (fair) behaviors exhibited by M_i are contained in the set of (fair) behaviors exhibited by M_{i-1}. In this sense, each model M_{i-1} determines a problem that the model M_i is required to satisfy. The problem of showing that M_i "simulates" M_{i-1} is therefore the problem of showing that M_i solves the problem determined by M_{i-1}. As an aid in doing so, we develop the notion of *possibilities mappings* that enable us to relate behaviors of one automaton to another.

We note that our model may be considered a special case of other models such as Milner's CCS and Hoare's CSP (see [Mil80] and [Hoa85]). Neither of these models, however, is entirely suitable for our purposes. In the first place, although Milner has found them to be mathematically superfluous in CCS, we find the notion of a process state a convenient descriptive tool when describing algorithms. More important, however, is the fact that fairness is difficult to express in CCS. Notions of fairness that have been studied in connection with CCS can be classified as either *weak fairness* or *strong fairness* (see [CS84], [Par85], and [Fra86]). Weak fairness requires that if an action π is continuously enabled, then it must be performed infinitely often. Strong fairness, on the other hand, requires that π be performed infinitely often even if it is enabled only infinitely often. These notions of fairness, however, are not satisfactory in event-driven systems. In such a system, for example, a process is always able to accept interrupts, but should not be required to interrupt itself unless some external source requests the interrupt. The problem is again the notion of control discussed above. There is no notion in CCS of an interface between processes from which we can deduce which which process controls the performance of a given action. By making a clear distinction between input and output actions, and by restricting ourselves to a simple notion of composition, we find that fairness is very simple to describe in our model.

Similar comments can also be made for CSP with respect to fairness (see [KdR83],

[Rei84], and [Fra86]). In fact, CSP further complicates the problem by identifying a process with (among other things) all *finite* behaviors of the process. Since it is impossible to deduce the infinite behavior of a process from its finite behaviors, CSP precludes the study of infinitary properties such as fairness without extending the semantics of a CSP process.

We note further that the complexity of the operations defined in CSP dooms the language to a complex semantics, making reasoning about systems of processes unintuitive and cumbersome. Reading between the lines of Hoare's book [Hoa85], it seems that Hoare himself would prefer to retain for nondeterministic processes the automata-theoretic (trace-theoretic[2]) semantics he develops for deterministic processes. However, the first nondeterministic operation introduced by Hoare is the nondeterministic OR, $\sqcap$, an operation combining two processes P and Q to form a process $P \sqcap Q$ that nondeterministically chooses (itself) to behave either like P or Q. A second operation, $\Box$, combines P and Q to form a process $P \Box Q$ allowing the *environment* to determine whether $P \Box Q$ behaves like P or Q. Both $P \sqcap Q$ and $P \Box Q$ have the same traces (since each behaves either like P or Q), but differ subtly in the fact that the environment has no control or knowledge of the choice $P \sqcap Q$ makes between P and Q. Thus, it is possible for $P \sqcap Q$ and $P \Box Q$ to be placed in an environment offering an action π as input, causing $P \sqcap Q$ to deadlock while $P \Box Q$ does not. This forces Hoare to make his first break from the trace-theoretic semantics of deterministic processes and define the notion of a *refusal*, a set of actions a process might refuse to perform. In our model, however, due to the unique property of input actions, a process will not block if its environment offers π as input. Thus, by suitably restricting our model, we are able to retain the intuitive, mathematically-tractable semantics of automata.

We note that there are systems of processes that can not be expressed in our model. Clearly, one such example is a system in which one process can block the progress of another as in CSP. These omissions, however, are the result of deliberate decisions, since, for example, it would be easy to define a notion of composition that allows us to express the process blocking of CSP. The simplicity of our model and its ease of use are the result of a careful limitation of its scope. Our experience has been that our model is sufficiently general to allow description of a significant number of interesting systems. We note that our model has already been found expressive enough to describe work in network algorithms (see [LLW87] and the third chapter of this thesis), concurrency control algorithms (see [LM86], [HLMW87], [FLMW87], and [GL87]), mutual exclusion algorithms (see [Wel87]), hardware register algorithms (see [Blo87]), and dataflow computation (see [Lyn86]). Furthermore, in many of these papers our model has been found to be extremely useful when identifying the interface between system components, and discovering a system's natural decomposition.

Just as popular models of computation do not seem appropriate for our work, popular proof techniques also seem inappropriate. For example, "Hoare logics" are a well-known

[2] A *trace* is a sequence of actions performed by a system or process.

method for proving that programs satisfy partial correctness assertions. Loosely speaking, a *partial correctness assertion* is a statement about the behavior of a terminating program. A program is said to satisfy such an assertion if it is satisfied by every terminating execution of the program. Therefore, a partial correctness assertion says nothing about program termination, but describes what will be true if and when the program halts. Hoare describes in [Hoa69] a method for proving that sequential programs satisfy partial correctness assertions. His method makes use of the observation, first noted by Floyd in [Flo67], that partial correctness assertions satisfied by a program S can be expressed in terms of predicates P and Q describing the program state before and after the execution of S. More formally, if P and Q are assertions about program variables and S is a program statement, $P\{S\}Q$ denotes the assertion that if P is true before the execution of S begins, then Q will be true if and when S terminates. Given a few simple axioms, Hoare shows how to derive partial correctness assertions $P\{S\}Q$ for arbitrary programs S. In the first step of the derivation, each statement S_i of S is annotated with assertions P_i and Q_i. In the second step, each assertion $P_i\{S_i\}Q_i$ is proven using axioms describing the various programming language constructs. Finally, general rules of inference (independent of any programming language) are used to combine these assertions into a proof of $P\{S\}Q$.

Hoare's method has proven to be a very effective method of verifying sequential programs. Most interestingly, it is possible to write hierarchical correctness proofs. Each program module S can be specified by a partial correctness assertion $P\{S\}Q$. Having proven each assertion $P\{S\}Q$, these assertions can be used in the proof of the larger program without reference to the implementation of S. Furthermore, since reasoning begins with a collection of partial correctness assertions characterizing program behavior and proceeds via rules of inference, this process can be automated if programmers are willing to supply certain intermediate assertions. Compilers for the language Euclid, for example, attempt to construct as much of the proof as possible (see [LGH+78]). Apt has written a comprehensive survey of Hoare logics in [Apt81] and [Apt84].

In [OG76], Owicki and Gries extend Hoare's method to distributed and parallel programs. Here, too, each statement S_i of each process S is annotated with assertions P_i and Q_i, and partial correctness assertions $P\{S\}Q$[3] are proven for each process S in isolation using a sequential programming logic similar to Hoare's. Unlike sequential algorithms, however, it is possible for one process action to affect the state of another. In order to prove partial correctness of an entire system of process, it is necessary to prove that no process can invalidate assertions appearing in the sequential proof of another process's partial correctness. Owicki and Gries refer to this condition as *noninterference*. For example, if $P\{S\}Q$ appears in the proof of one process and the assertion R labels one statement appearing in another process, noninterference requires that the assertion $(P \wedge R)\{S\}(Q \wedge R)$ hold; that is, the execution of S does not invalidate R. This method of Owicki and Gries has been found to be quite successful, just as Hoare's method has been found to be successful for sequential programs. Gries has constructed a nice proof

[3]Owicki and Gries actually use the notation $\{P\}S\{Q\}$.

of Dijkstra's on-the-fly garbage collector in [Gri77], an algorithm with such fine interleaving that the only atomic action required is memory reference. Levin and Gries show in [LG81] how the method of Owicki and Gries can be used to verify CSP processes. Furthermore, Schlichting and Schneider show in [SS84] how message passing primitives can be incorporated into this framework.

As with sequential programs, the partial correctness of systems may be specified with partial correctness assertions of the form $P\{S\}Q$. Due to the possibility of process interference, however, it is *not* possible to specify the partial correctness of individual processes in terms of such assertions. The specification of a process must also describe its environment if such assertions are to be used. Without a description of its environment, it is impossible to prove that a process satisfies most partial correctness assertions. Furthermore, modification of a single process requires redoing a major portion of a system's proof of correctness since it must be shown that this modification does not violate partial correctness assertions appearing in the correctness proofs of other processes. Thus, both specifications and correctness proofs using partial correctness assertions of the form $P\{S\}Q$ lack an important modularity. We consider this lack of modularity to be a major problem in protocol verification.

Lamport attempts to resolve this lack of modularity in [Lam80]. Here Lamport redefines the assertion $P\{S\}Q$ to mean that if execution is begun anywhere inside S with P true, then executing S will leave P true while control is inside S, and will make Q true if and when S terminates. Such a definition is possible for Lamport since he allows the predicates P and Q to refer to program locations, whereas Owicki and Gries restricted P and Q to program variables. The advantage of Lamport's scheme is that partial correctness assertions for an entire system can be verified given partial correctness assertions specifying each system component. After system correctness has been proven from component specifications, any implementation of the components satisfying their specifications can be used in the system's implementation. Lamport's method, however, is not without its difficulties. For example, suppose that S is a system component making heavy use of shared variables. It seems difficult to construct assertions P that are invariant throughout the execution of S without knowing how S uses these shared variables.

In our method, the problem of modular specification disappears since an environment is implicitly specified by the fact that input actions are continuously enabled. (In other words, anything can happen in the environment of a process.) As a result, if a partial correctness assertion can be proven about process behavior, the partial correctness assertion holds regardless of the process's actual environment. Thus in our method it is no longer necessary to prove noninterference after proving the correctness of individual processes. Furthermore, it is no longer necessary to redo any part of a correctness proof when a process is modified, other than the correctness of the modified process itself. (A similar consequence of such input requirements can be found in [MCS82], [Sta84], and [LM86].) Also, notice that Hoare-style specifications do not make clear the interface

between a system component and its environment. As previously mentioned, this interface is crucial to the definition of fair computation. In contrast, our model clearly defines this interface as the set of actions the process can perform, together with information about which actions denote input and output of the process.

We note that due to the generality of automaton transitions, partial correctness assertions describing automaton transitions similar to those of Hoare describing common programming language constructs may not always be easy to find. However, if transitions are described in terms of preconditions that must be satisfied before an action can be performed, and the effect of an action on an automaton state, then partial correctness assertions can be constructed for each action. Furthermore, the general, language-independent rules of inference used in Hoare-like systems are clearly valid in our model of computation. Thus, while we do not make use of such arguments in our work, it *is* possible to construct Hoare-like proofs of partial correctness assertions in our model.

Notice that partial correctness assertions describe safety properties, and not liveness properties. Since there is no notion of system computation in these Hoare logics, there is no notion of eventuality. We note that safety properties can often be used to prove liveness properties. For example, Owicki and Gries show in [OG76] how well-foundedness arguments can be incorporated into Hoare logics to prove termination of programs. Alpern and Schneider go farther in [AS85] and show that the verification of both liveness and safety properties can be reduced to proving what are essentially partial correctness assertions. However, the specification of a liveness condition in terms of partial correctness assertions is often an unintuitive formulation.

A more natural expression of such properties is possible with temporal logic. Temporal logic was introduced by Pnueli in [Pnu77] as an adaptation of classical modal logic suitable for reasoning about concurrent programs. The two paper series [MP81b] and [MP81a] by Manna and Pnueli is a thorough introduction to the expression of properties of concurrent programs, and the verification of these properties, using temporal logic. Here the meaning of a system computation is a sequence of system states. The fundamental temporal operators are the unary operator $\Box$ and its dual $\Diamond$. Loosely speaking, a computation satisfies the expression $\Box P$, pronounced "henceforth P," if P is true throughout the computation; and a computation satisfies the expression $\Diamond P$, pronounced "eventually P," if there is a point during the computation at which P is true. Many interesting properties of computation may be specified with these simple operators. For instance, the expression $\Box(P \supset \Diamond Q)$ states that the property P causes the property Q to hold; the expression $\Box\Diamond P$ states that the property P holds infinitely often.

Temporal logic is a useful abstraction with which to specify and reason about program behavior. Since the meaning of a computation is a *sequence* of states, temporal logic is able to express liveness properties as well as safety properties, and these expressions are typically quite concise. Since reasoning in temporal logic begins with a collection of axioms characterizing program behavior, and proceeds via general rules of inference, reasoning in temporal logic has potential for automation. Furthermore, while Hoare logics

make use of inference rules that are independent of any programming language, most of the work in a Hoare-style proof deals with language-specific semantics. In contrast, reasoning in temporal logic is valid for all programs. The difficulty, of course, is in abstracting from an implementation to a temporal logic characterization of its behavior, and this problem is often swept under the rug.

A great deal of work in temporal logic concerns reasoning about system correctness after system components have been specified in terms of temporal logic (see, for example, [HO80], [SMS81], [OL82], [Lam83], [Sta84] and [NGO85]). The most dramatic distinction between these works is the way in which temporal logic is used to describe system behavior. Schwartz and Melliar-Smith give purely temporal specifications of programs in [SMS81]. In these specifications, even the notion of a process state has been replaced by temporal specifications. Consequently, the resulting specifications are quite complex, involving nested "until" operators in addition to the temporal operators described above. These specifications are often difficult to understand, and difficult to reason about. On the other hand, Hailpern and Owicki make great use of the notion of program state in [HO80]. They add *history variables* to the program state that describe the history of events over communication links, and reason about the values assumed by these variables. History variables are a convenient descriptive tool found in many proof styles, and the specifications produced by Hailpern and Owicki are generally easy to understand. The history variables, however, do not affect program behavior, and in proofs reasoning about history variables the history variables themselves seem extraneous. Between the extremes of [SMS81] and [HO80] is the work of Lamport in [Lam83]. Here the process state modeled consists only of program variables, and temporal logic assertions describe the sequence of values these variable assume. Although an automaton state can be seen as a natural extension of history variables, our proofs tend to have a flavor similar to those of Lamport's in [Lam83].

While a great deal of work has studied the problem of reasoning about systems after system components have been specified in terms of temporal logic, less has been devoted to proving that an implementation actually meets its temporal logic specification. One attempt is that of Owicki and Lamport in [OL82], improving on the work of Lamport in [Lam77]. Since safety properties can be proven using methods of Owicki and Gries, of particular interest is the style of proving liveness properties Owicki and Lamport describe. Owicki and Lamport construct diagrams called *proof lattices* that outline the structure of a proof of a liveness property. Informally, a proof lattice is an acyclic directed graph with a single *entry node* having no incoming edges, and a single *exit node* having no outgoing edges. Nodes of the graph are labeled with assertions. A node labeled A with outgoing edges to nodes labeled $A_1, \ldots, A_n$ denotes the assertion that if A holds, then one of the assertions $A_1, \ldots, A_n$ must eventually hold; that is $A \supset \Diamond(A_1 \vee \ldots \vee A_n)$. Suppose each such assertion can be proven for a program. If the entry node is labeled with A and the exit with B, then the proof lattice amounts to a proof of the liveness property $A \supset \Diamond B$ for the program. Manna and Pnueli extend the use of proof lattices in [MP84]. In this

work, however, an automata-theoretic model of computation is explicitly defined, and proof rules are given for proving that each assertion denoted by edges of the proof lattice is satisfied by the system modeled by an automaton. We find this work a very satisfying example of how an automata-theoretic model of computation and temporal logic can be used together. Given an automata-theoretic description of system implementation, temporal logic provides a useful abstraction for reasoning about system behavior. While we have not fixed on one particular specification language, we feel that temporal logic and our automata-theoretic model of computation can work well together. In particular, through the use of automata we are able to incorporate temporal logic into hierarchical correctness proofs.

The use of abstraction is an important aspect of our style of algorithm verification. Most work in the literature claiming to produce proofs with a hierarchical structure actually allow system components to be verified independently, and then combined to verify the correctness of the system. This notion of hierarchical verification is a restricted version of the notion we use in this work. Here we actually construct models of the entire system at conceptually different levels of abstraction, rather than merely combining local process states into global system states.

Our work most closely resembles that of Lamport in [Lam83]. Here Lamport specifies a program with a collection of *state functions* mapping program states into sets of values, a collection of *initial conditions* essentially defining the set of states in which the system may begin computation, and a collection of *properties* describing safety and liveness conditions. We note that the values to which states are mapped by state functions can be thought of as state variables describing relevant aspects of the system to be implemented. Furthermore, the properties included in the system specification define allowed and required changes in the values these variables assume. If these variables are collected into states, then the variables together with the properties essentially define an automaton together with a collection of eventuality conditions restricting the computations of the automaton. If the program implementing the specified system is considered to be an automaton, as is implicitly the case in Lamport's work, then the state functions can be thought of as mappings from an automaton describing the system at one level of abstraction to an automaton describing the system at a higher level of abstraction. This is the technique used in our work. Our work is an improvement on that of Lamport's in the sense that we carry his style of specifications to its natural conclusion, making the automata-theoretic flavor of his work explicit. Furthermore, we make explicit his underlying assumption that what is important about a process is the externally observable behavior of the process. His work seems to imply that the variables and state functions must be describing some aspect of the system that must appear in the implementation. We feel, however, that they are to be considered merely descriptive tools, and that the notion of subset containment used as the notion of correctness in this work is the notion of correctness actually underlying Lamport's work.

Other work similar to ours is that of Stark in [Sta84]. Many of the aims and ideas

underlying his work are the same as ours, but his model is much more general than ours. We find our model to be simpler and easier to use than Stark's, and expressive enough to describe most systems of interest. Work on hierarchical verification also includes that of Lam and Shankar in [LS84a]; Harel in [Har87]; and Lamport, Lynch, and Welch in [LLW87]. Each of these techniques analyzes an algorithm by abstracting away certain portions of the algorithm (rather than mapping to an entirely different level of conceptual abstraction as we do here) and studying the remaining "image" of the original algorithm. To Lam and Shankar, the advantage of this method seems to be that it allows highly interdependent modules of a system to be studied in isolation. Lamport, Lynch, and Welch seem to be taking this notion of "projection" one step further. They show how projections onto different modules can be combined into a proof of the entire system, giving the proof a lattice-like structure. While still work in progress, their work seems to be shedding new light on the intellectual organization of protocol verification. The progress being made in their research can certainly be incorporated into ours.

The remainder of this thesis consists of two parts. First, in Chapter 2, we formally define our model of computation and develop the machinery needed to use our model in the construction of hierarchical correctness proofs. Then, in Chapter 3, we illustrate the use of our model by proving the correctness of Schönhage's distributed resource arbiter. Finally, in Chapter 4, we end with some concluding remarks, including some ideas for future work.

Chapter 2

The Input-Output Automaton Model

In this chapter we define the input-output automaton model. We begin with a formal definition of an input-output automaton, and define operations that may be performed on automata, including the composition of automata. We then show how fairness can be modeled with automata. Finally, we develop the machinery necessary to use automata in the construction of modular, hierarchical correctness proofs for distributed algorithms.

2.1 Input-Output Automata

Having informally described our model in the introduction, we now formally define an input-output automaton. Since the actions of an automaton define the interface between an automaton and its environment, it is convenient to be able to refer to this interface explicitly. Given three disjoint sets *in*, *out*, and *int* of input, output, and internal actions, respectively, we refer to the triple (*in*, *out*, *int*) as an *action signature* S. We denote the sets *in*, *out*, and *int* by $in(S)$, $out(S)$, and $int(S)$, respectively; and we denote the entire set of actions $in \cup out \cup int$ by $acts(S)$. Since *int* is the set of internal actions, it is natural to refer to $in \cup out$ as the set of *external actions*, denoted by $ext(S)$. Finally, we denote the set $int \cup out$ of locally-controlled actions by $local(S)$.

An *input-output automaton* (or *automaton*) A consists of five components:

1. a set $states(A)$ of *states*,
2. a set $start(A) \subseteq states(A)$ of *start states*,
3. an action signature $sig(A)$,

4. a transition relation $steps(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$, with the property that for every state a and input action π there is a transition (a, π, a') in $steps(A)$, and

5. an equivalence relation $part(A)$ on $local(sig(A))$.

Notice that the transition relation $steps(A)$ has the property that input actions are continuously enabled, as mentioned in the introduction. Notice, also, that the equivalence relation $part(A)$ is the partition of the locally-controlled actions alluded to in the introduction. This partition will be used when we define the notion of fair computation in Section 2.2.

We refer to an element (a, π, a') of $steps(A)$ as a *π-step* from a to a'. It will occasionally be convenient to denote the step (a, π, a') by $a \stackrel{\pi}{\rightarrow} a'$, and to denote the sequence of steps $a_0 \stackrel{\pi_1}{\rightarrow} a_1 \cdots \stackrel{\pi_n}{\rightarrow} a_n$ by $a_0 \stackrel{\pi_1 \cdots \pi_n}{\rightarrow} a_n$. The step (a, π, a') is called an *input step* if π is an input action, and *output steps*, *internal steps*, *external steps*, and *locally-controlled steps* are similarly defined. If (a, π, a') is a step of A, then π is said to be *enabled* from a. Since every input action is enabled from every state, automata are said to be *input-enabled*.

An *execution fragment* of A is a finite sequence $a_0\pi_1a_1 \ldots \pi_k a_k$ or infinite sequence $a_0\pi_1a_1\pi_2a_2 \ldots$ of alternating states and actions such that $(a_i, \pi_{i+1}, a_{i+1})$ is a step of A for every i. An execution fragment beginning with a start state is called an *execution*. We denote the set of executions of A by $execs(A)$. A state is said to be *reachable* if it is the final state of a finite execution. The *schedule* of an execution x is the subsequence of actions appearing in x, denoted by $sched(x)$. We denote the set of schedules of A by $scheds(A)$.

We will soon consider certain subsets of an automaton's executions or schedules (such as the set of fair computations) to be of particular interest. Since we will compose automata, it will be necessary to have ways of composing sets of executions or schedules as well. If these compositions are to be meaningfully related, however, certain information about the structure of the original automata must be retained. In particular, it is important to retain information about the action signatures of these automata. We are therefore led to define the notions of execution modules and schedule modules, essentially sets of executions or schedules, respectively, together with an action signature.

An *execution module* E consists of a set $states(E)$ of states, an action signature $sig(E)$, and a set $execs(E)$ of executions. Each execution of E is an alternating sequence of states and actions of E beginning with a state, and ending with a state if the sequence is finite. Each execution x has an associated schedule $sched(x)$ that consists of the subsequence of actions appearing in x. We denote the set of schedules of E by $scheds(E)$. An execution module E is said to be an execution module of an automaton A if E and A have the same states, the same action signature, and the executions of E are contained in the executions of A. Notice that an execution module E is always an execution module of some automaton. In particular, E is an execution module of the automaton having the

states and action signature of E, and the transition relation $states(E) \times acts(sig(E)) \times states(E)$. We denote the execution module of the automaton A having $execs(A)$ as its set of executions by $Execs(A)$. (We follow the convention of denoting sets with lower case names and modules with capitalized names.)

A *schedule module* S consists of an action signature $sig(S)$ together with a set $scheds(S)$ of schedules. Each schedule of S is a finite or infinite sequence of the actions of S. Given an execution module E, there is a natural schedule module associated with E consisting of the action signature and schedules of E. We denote this schedule module by $Scheds(E)$, and write $Scheds(A)$ as shorthand for $Scheds(Execs(A))$.

We refer collectively to automata, execution modules, and schedule modules as *objects*, the *type* of an object determining whether it is an automaton, execution module, or schedule module. For notational convenience, given an object O we often omit reference to its action signature and write, for example, $in(O)$ for $in(sig(O))$.

Since it is typically the case that more than one automaton can model the same process, some notion of equivalence is needed. Intuitively, the external observer of a process (a user of the process, for instance) can detect only the sequence of actions performed by the process. In fact, the only actions detectable by such an observer are the external actions of the process. We are therefore led to define a notion of equivalence determined by the externally visible sequences of actions produced by an object. Since we will consider in Section 2.2.2 a second notion of equivalence based on the fair behavior of an object, we refer the the current notion of equivalence as unfair equivalence.

We begin by defining an operation that essentially extracts the externally visible behavior of an object. An *external action signature* is an action signature consisting entirely of external actions; that is, having no internal actions. The external action signature of an object O is the action signature obtained by removing the internal actions from the action signature of O. An *external schedule module* is a schedule module with an external action signature. Given a sequence y of actions and a set Π of actions, we denote by $y|\Pi$ the subsequence of y consisting of actions from Π. The external schedule module of an object O, denoted by $External(O)$, is the external schedule module with the external action signature of O and the schedules $\{y|ext(O) : y \in scheds(O)\}$ obtained by removing the internal actions from the schedules of O. We define the *unfair behavior* of O, denoted by $Ubeh(O)$, to be the external schedule module $External(O)$.

Two objects O and P of the same type are said to be *unfairly equivalent*, denoted by $O \stackrel{unfair}{=} P$, if $Ubeh(O) = Ubeh(P)$. This equivalence is clearly an equivalence relation, and we will see that it is also a congruence with respect to the operations we now define on objects.

2.1.1 Composition

To build models of complex systems, we compose models of simpler system components. In this section we show how to compose objects to construct such models.

Composition of Automata

Informally, the composition of a collection of automata is their Cartesian product, with the added requirement that automata synchronize the performance of shared actions. That is, each automaton is allowed to take steps independently, with the restriction that if one automaton takes a π-step, then all automata sharing π as an action must also take a π-step. This synchronization models communication between system components: If π is an output action of A and an input action of B, then the simultaneous performance of π models communication from A to B. Since synchronization is meant only to model communication, however, two automata sharing π as an output action should not be required to perform π simultaneously. We note that two processors cannot be expected to perform an output action simultaneously in an asynchronous system. Rather than complicate the notion of composition, we require instead that the output actions of composed automata be disjoint. Since internal actions are meant to model externally undetectable actions, we are faced with the need for a similar restriction for internal actions. We require that the internal actions of each automaton in a composition be disjoint from the actions of the remaining automata.

Having restricted the composition of automata to those with suitably compatible action signatures, determining the type of an action in a composition is fairly simple: Output actions of the component automata become output actions of the composition, internal actions of component automata become internal actions of the composition, and all remaining (input) actions of the component automata become input actions of the composition. Notice that the composition of automata does *not* hide communication between component automata. To hide such communication will require the use of a hiding operation defined later in Section 2.1.2.

Finally, recall that associated with every automaton (in particular, with a composition of automata) is a partition of its locally-controlled actions. Our intuitive understanding of this partition is that each class represents the locally-controlled actions of one system component. A natural partition of a composition's locally-controlled actions is to place the locally-controlled actions of each component automaton in a separate class. Since the restrictions we impose on the composition of automata ensure that the locally-controlled actions of the component automata are disjoint, this is indeed a partition. However, each component automaton may model many system components. We therefore partition a composition's locally-controlled actions by taking each class of each component automaton as a separate class of the composition's partition. That is, the partition of a composition's locally-controlled actions is the union of its components' partitions.

We are now in a position to formally define the composition of automata. We begin by defining a composition of action signatures. Previous discussion suggests that the action signatures $\{S_i : i \in I\}$ be called *compatible* if for all $i, j \in I$ we have

1. $out(S_i) \cap out(S_j) = \emptyset$, and
2. $int(S_i) \cap acts(S_j) = \emptyset$.

In general, we say that the objects $\{O_i : i \in I\}$ are *compatible* if their action signatures are compatible. The composition $S = \prod_{i \in I} S_i$ of compatible action signatures $\{S_i : i \in I\}$ is defined to be the action signature with

1. $in(S) = \bigcup_{i \in I} in(S_i) \; - \; \bigcup_{i \in I} out(S_i)$,
2. $out(S) = \bigcup_{i \in I} out(S_i)$, and
3. $int(S) = \bigcup_{i \in I} int(S_i)$.

Notice that this composition is commutative and associative.

The composition $A = \prod_{i \in I} A_i$ of compatible automata $\{A_i : i \in I\}$ is defined to be the automaton with

1. $states(A) = \prod_{i \in I} states(A_i)$,
2. $start(A) = \prod_{i \in I} start(A_i)$,
3. $sig(A) = \prod_{i \in I} sig(A_i)$,
4. $part(A) = \bigcup_{i \in I} part(A_i)$, and
5. $steps(A)$ equal to the set of triples $(\{a_i\}, \pi, \{a_i'\})$ such that for all $i \in I$
   (a) if $\pi \in acts(A_i)$ then $(a_i, \pi, a_i') \in steps(A_i)$, and
   (b) if $\pi \notin acts(A_i)$ then $a_i = a_i'$.

Notice that since the automata A_i are input-enabled, so is their composition, and hence their composition *is* an automaton. When I is a finite set $\{1, \ldots, n\}$, we will frequently denote the composition $\prod_i A_i$ by $A_1 \cdot \ldots \cdot A_n$.

As a simple example of automaton composition, consider the two automata A and B shown at the top of Figure 2.1, and their composition $A \cdot B$ shown at the bottom of the same figure. (A caret points to the single initial state of each automaton.) The action α

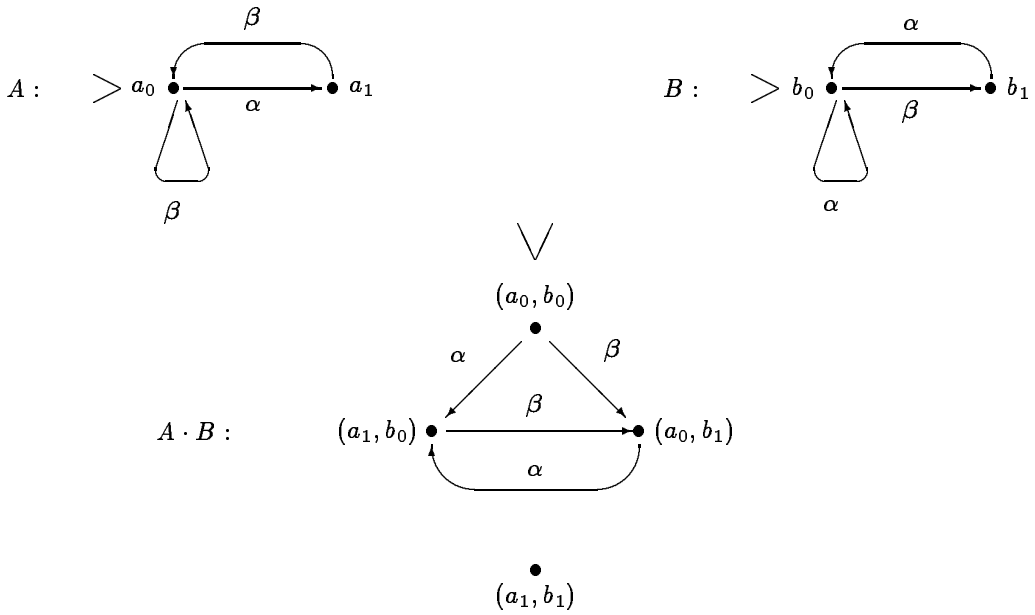


Figure 2.1: An example of automaton composition.

is an output action of A and an input action of B, and the action β is an output action of B and and input action of A. Notice that since each waits for the other to take an output step before taking an output step itself, the automata A and B alternate output steps in executions of the composition $A \cdot B$. Notice, furthermore, that since α and β are output actions of A and B, respectively, all actions of the composition $A \cdot B$ are output actions. Finally, notice that the partition of the composition's locally-controlled actions (in this case, the output actions) places α and β in separate equivalence classes.

The composition of automata has two simple properties. First, an execution of a composition $A = \prod_i A_i$ always induces executions in the component automata A_i. If $a = \{a_i\}$ is a state of A, let $a|A_i = a_i$. If $x = a_0\pi_1 a_1 \ldots$ is an execution of A, let $x|A_i$ be the sequence obtained by deleting $\pi_j a_j$ when π_j is not an action of A_i, and replacing the remaining a_j with $a_j|A_i$. We now have the following:

Lemma 1: If $x \in execs(\prod_{i \in I} A_i)$, then $x|A_i \in execs(A_i)$ for all $i \in I$.

Proof: Let $A = \prod_i A_i$, and suppose that $x = a_0\pi_1 a_1 \ldots$. By the definition of an execution, a_0 is a start state of A, and every triple (a_{k-1}, π_k, a_k) is a step of A. Two facts follow from the definition of the composition A. First, $a_0|A_i$ must be a start state of A_i. Second, if π_k is an action of A_i then $(a_{k-1}|A_i,\ \pi_k,\ a_k|A_i)$ is a step of A_i. If π_k is not an action of A_i then $a_{k-1}|A_i = a_k|A_i$. Thus, if $x|A_i = s_0\sigma_1 s_1 \ldots$, then s_0 is a start state

of A_i, and every triple (s_{j-1}, σ_j, s_j) is a step of A_i. Therefore, $x|A_i$ is an execution of A_i. □

Conversely, under certain conditions an execution of a composition is induced by executions of its components. Here and elsewhere, we denote $y|acts(O)$ by $y|O$ for arbitrary objects O.

Lemma 2: Let $\{A_i : i \in I\}$ be a collection of compatible automata. Let x_i be an execution of A_i for every $i \in I$, and let y be a sequence of actions from the A_i. If $y|A_i = sched(x_i)$ for every $i \in I$, then there is an execution x of $\prod_{i \in I} A_i$ such that $y = sched(x)$, and $x_i = x|A_i$ for every $i \in I$.

Proof: Let $A = \prod_i A_i$. Suppose that $y = \pi_1 \pi_2 \ldots$. Since $y|A_i = sched(x_i)$, we can write $x_i = a_0^i \pi_{i_1} a_1^i \pi_{i_2} a_2^i \ldots$. Let $i_0 = 0$. Let $x = a_0 \pi_1 a_1 \ldots$ where a_j is defined as follows: If $i_k \leq j < i_{k+1}$, then $a_j|A_i = a_k^i$. That is, the automaton A_i remains in state a_k^i between the performance of actions π_{i_k} and $\pi_{i_{k+1}}$, and changes state to a_{k+1}^i upon the performance of $\pi_{i_{k+1}}$. First, we claim that a_0 is a start state of A. Since for all i we have that $i_0 = 0$ implies $a_0|A_i = a_0^i$, a start state of A_i, we are done. Second, we claim that (a_{j-1}, π_j, a_j) is a step of A for all j. Suppose $\pi_j \in acts(A_i)$. Then $\pi_j = \pi_{i_k}$ for some k. It follows that $a_{j-1}|A_i = a_{k-1}^i$ and $a_j|A_i = a_k^i$ since $i_{k-1} < j = i_k$. Thus, $(a_{j-1}|A_i, \pi_j, a_j|A_i)$ is a step of A_i. Conversely, suppose $\pi_j \notin acts(A_i)$. Then $i_k < j < i_{k+1}$, and it follows that $a_{j-1}|A_i = a_k^i = a_j|A_i$. In either case, (a_{j-1}, π_j, a_j) is a step of A for all j. It follows that x is an execution of A, and furthermore that $y = sched(x)$ and $x|A_i = x_i$ for all i. □

The following corollary, essentially Lemma 4 from [LM86], ensures that composition preserves the notion that a system component controls the performance of its own locally-controlled actions. As a result, when reasoning about the enabling of an action in a composition, it is enough to reason about the enabling of the action at one component.

Corollary 3: Let y be a finite schedule of a composition $A = \prod_{i \in I} A_i$. Let π be a locally-controlled action of A_i, and let $y' = y\pi$. If $y'|A_i$ is a schedule of A_i, then y' is a schedule of A.

Proof: Since y is a finite schedule of A, there is a finite execution x of A such that $y = sched(x)$. By Lemma 1, $x|A_j$ is an execution of A_j for every $j \in I$. Since π is a locally-controlled action of A_i, if π is an action of A_j (for any $j \neq i$), then π is an input action of A_j. Since the A_j are input-enabled, and since $y'|A_i$ is a schedule of A_i, for every $j \in I$ there is an execution x_j' of A_j such that $y'|A_j = sched(x_j')$. By Lemma 2, there is an execution x' of A such that $y = sched(x')$, and hence y is an execution of A. □

Composition of Execution Modules

We now define the composition of execution modules. The composition $E = \prod_{i \in I} E_i$ of compatible execution modules $\{E_i : i \in I\}$ is defined as follows. The states of E are $\prod_{i \in I} states(E_i)$, and the action signature is $\prod_{i \in I} sig(E_i)$. Given a state $s = \{s_i\}$ of the composition, we define $s|E_i = s_i$. Given a sequence $x = s_0 \pi_1 s_1 \ldots$ of states and actions of E, we define $x|E_i$ to be the sequence obtained by removing $\pi_j s_j$ if π_j is not an action of E_i, and replacing the remaining s_j by $s_j|E_i$. The executions of E are those sequences $s_0 \pi_1 s_1 \ldots$ such that for every $i \in I$ we have that $x|E_i$ is an execution of E_i, and that $s_{j-1}|E_i = s_j|E_i$ whenever π_j is not an action of E_i. The next lemma gives an alternative characterization of the composition of execution modules.

Lemma 4: Let $\{E_i : i \in I\}$ be a collection of compatible execution modules. Suppose E_i is an execution module of an automaton A_i for every $i \in I$. Then $\prod_{i \in I} E_i$ is the execution module of $\prod_{i \in I} A_i$ with executions x such that $x|A_i$ is an execution of E_i for every $i \in I$.

Proof: Let $E = \prod_i E_i$ and $A = \prod_i A_i$. Since E_i is an execution module of A_i, it follows that E_i and A_i have the same states and action signature, and hence so do E and A. We need only check that the executions of E are the executions x of A such that $x|A_i$ is an execution of E_i. Suppose x is an execution of E. The execution x is a sequence $s_0 \pi_1 s_1 \ldots$ of states and actions of E such that $x|E_i$ is an execution of E_i, and $s_{j-1}|E_i = s_j|E_i$ whenever π_j is not an action of E_i. Since E_i is an execution module of A_i, $(s_{j-1}|A_i, \pi_j, s_j|A_i)$ is a step of A_i whenever π_j is an action of A_i, and $s_{j-1}|A_i = s_j|A_i$ whenever π_j is not an action of A_i. It follows that x is an execution of A, and furthermore that $x|A_i$ is an execution of E_i for every $i \in I$. Conversely, suppose x is an execution of A such that $x|A_i$ is an execution of E_i for every $i \in I$. Clearly, x is a sequence $s_0 \pi_1 s_1 \ldots$ of states and actions of E such that $x|E_i$ is an execution of E_i for every $i \in I$. Furthermore, from the definition of the composition of automata we see that $s_{j-1}|E_i = s_j|E_i$ whenever π_j is not an action of E_i. It follows that x is an execution of E, as desired. □

This composition is defined so that the following result holds.

Lemma 5: For all compatible automata $\{A_i : i \in I\}$,

$$Execs(\prod_{i \in I} A_i) = \prod_{i \in I} Execs(A_i).$$

Proof: Let $A = \prod_i A_i$. Furthermore, let $EC = Execs(\prod_i A_i)$ and $CE = \prod_i Execs(A_i)$. Notice that EC is an execution module of A. Furthermore, since $Execs(A_i)$ is an execution module of A_i for every $i \in I$, Lemma 4 implies that CE is also an execution module

of A. It follows that EC and CE have the same states and action signature. We need only show that they have the same executions. By Lemmas 1 and 4, x is an execution of EC iff x is an execution of A such that $x|A_i$ is an execution of A_i for every $i \in I$ iff x is an execution of CE. Thus, EC and CE have the same executions, and hence are equal. □

Composition of Schedule Modules

We now define the composition of schedule modules. The composition $\prod_{i \in I} S_i$ of compatible schedule modules $\{S_i : i \in I\}$ is defined to be the schedule module with action signature $\prod_{i \in I} sig(S_i)$, and schedules y such that $y|S_i$ is a schedule of S_i for every $i \in I$. This composition is defined so that the following result holds.

Lemma 6: For all compatible execution modules $\{E_i : i \in I\}$,

$$Scheds(\prod_{i \in I} E_i) = \prod_{i \in I} Scheds(E_i).$$

Proof: Let $SC = Scheds(\prod_i E_i)$ and $CS = \prod_i Scheds(E_i)$. Since SC and CS clearly have the same action signatures, we need only show that they have the same schedules. Suppose E_i is an execution module of an automaton A_i for every $i \in I$. Notice that y is a schedule of SC iff y is the schedule of an execution x of $\prod_i E_i$. Lemma 4 implies this is the case iff y is the schedule of an execution x of $\prod_i A_i$ such that $x|E_i = x_i$ is an execution of E_i for every $i \in I$. Lemma 2 implies this is the case iff $y|E_i$ is the schedule of an execution x_i of E_i. From the definition of schedule module composition we see this is the case iff y is a schedule of CS. Thus, SC and CS have the same schedules, and hence are equal. □

In addition, we have the following.

Lemma 7: For all compatible schedule modules $\{S_i : i \in I\}$,

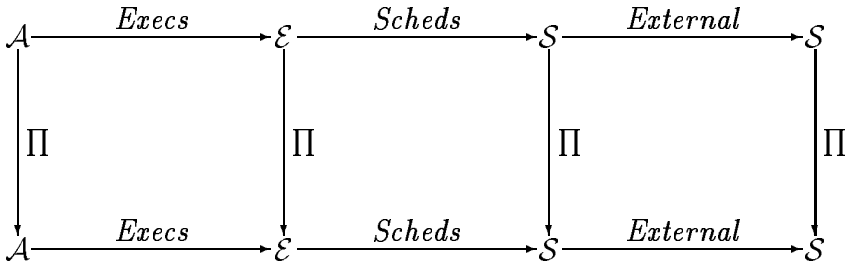
$$External(\prod_{i \in I} S_i) = \prod_{i \in I} External(S_i).$$

Proof: Let $S = \prod_i S_i$, and let $EC = External(\prod_i S_i)$ and $CE = \prod_i External(S_i)$. Since the schedule modules S_i are compatible, $int(S_i) \cap acts(S_j) = \emptyset$ for all $i \neq j$. That is, the internal actions of each schedule module are disjoint from the actions of the others. With this observation, it follows from the definition of action signature composition that EC and CE have the same action signature. We need only show they have the same schedules. If y is a schedule of EC, then $y = y'|ext(S)$ for some schedule y' of S. Since $y'|S_i$ is a schedule of S_i, $y|External(S_i) = y'|External(S_i)$ is a schedule of $External(S_i)$,

and hence y is a schedule of CE. Conversely, suppose y is a schedule of CE. Then $y|External(S_i) = y_i|ext(S_i)$ for some schedule y_i of S_i. Suppose $y = \pi_1\pi_2\ldots$. Let us write $y_i = \alpha_0^i\beta_1^i\alpha_1^i\ldots$ where α_j^i is a (possibly empty) sequence of internal actions of S_i, and β_j^i is π_j if π_j is an external action of S_i and the empty string otherwise. Let $y' = \gamma_0\pi_1\gamma_1\ldots$ where γ_j is an arbitrary interleaving of the actions appearing in the α_j^i. Then y' is a sequence of actions of S such that $y'|S_i = y_i$ is a schedule of S_i, so y' is a schedule of S. Since $y = y'|ext(S)$, y is a schedule of EC. □

Lemmas 5, 6, and 7 can be summarized as follows.

Corollary 8: Let $\mathcal{A}$ denote the class of automata, $\mathcal{E}$ denote the class of execution modules, and $\mathcal{S}$ denote the class of schedule modules. The following diagram commutes:



One important consequence of Corollary 8 is the following result, which says that the (unfair) behavior of a composition is the composition of its components' (unfair) behaviors.

Lemma 9: $Ubeh(\prod_{i\in I} O_i) = \prod_{i\in I} Ubeh(O_i)$ for all compatible objects $\{O_i : i \in I\}$.

It is now possible to see that composition satisfies a number of natural axioms. We note that the following result is an immediate consequence of the definition of schedule module composition.

Lemma 10: Suppose $S = \prod_i S_i$, $T = \prod_i T_i$, $U = \prod_i U_i$, and $V = \prod_i V_i$ where the S_i, T_i, U_i, and V_i are schedule modules.

1. $S \cdot T = T \cdot S$.
2. $(S \cdot T) \cdot U = S \cdot (T \cdot U)$.
3. If $S = T$ and $U = V$, then $S \cdot U = T \cdot V$ whenever the compositions $S \cdot U$ and $T \cdot V$ are defined.

As a consequence of Lemmas 9 and 10, we have the following.

Lemma 11: Suppose $O = \prod_i O_i$, $P = \prod_i P_i$, $Q = \prod_i Q_i$, and $R = \prod_i R_i$ where the O_i, P_i, Q_i, and R_i are objects.

1. $O \cdot P \stackrel{unfair}{=} P \cdot O$.
2. $(O \cdot P) \cdot Q \stackrel{unfair}{=} O \cdot (P \cdot Q)$.
3. If $O \stackrel{unfair}{=} P$ and $Q \stackrel{unfair}{=} R$, then $O \cdot Q \stackrel{unfair}{=} P \cdot R$ whenever the compositions $O \cdot Q$ and $P \cdot R$ are defined.

Proof: Recall that $O \cdot P \stackrel{unfair}{=} P \cdot O$ iff the external schedule modules $Ubeh(O \cdot P)$ and $Ubeh(P \cdot O)$ are equal. By Lemma 9 we see that $Ubeh(O \cdot P) = Ubeh(O) \cdot Ubeh(P)$ and $Ubeh(P \cdot O) = Ubeh(P) \cdot Ubeh(O)$. However, Lemma 10 implies that these schedule modules are equal. Therefore, $O \cdot P \stackrel{unfair}{=} P \cdot O$. The remaining parts are similar. □

Conditions 1 and 2 say that composition is commutative and associative up to equivalence. Condition 3 says that composition is a almost congruence with respect to composition. However, since the external behavior of O and Q contains no information about the internal actions of O and Q, their external behaviors do not determine whether they are compatible, and hence whether their composition is defined. Thus, equivalence is not quite a congruence. We call an equivalence satisfying condition 3 a *weak congruence*. Notice that this weakness is due only to conflicting *internal* actions names, actions not affecting the external behavior of an object. In Section 2.1.3 we will see how to perform a syntactic renaming of internal action names to avoid this conflict without affecting the external behavior of the object. This is reminiscent of variable renaming to avoid conflict during substitution in predicate calculus.

2.1.2 Action Hiding

Recall that composition does not hide actions modeling interprocess communication: In particular, if π is an output action of A and an input action of B modeling communication from A to B, then π becomes an (external) output action of $A \cdot B$. Since this communication is really internal to the system $A \cdot B$, we would like to be able to hide π from external view, to transform π into an internal action of $A \cdot B$.

Given an object O and a set of actions Σ, we define the object $Hide_\Sigma(O)$ to be the object differing from O only in that

1. $in(Hide_\Sigma(O)) = in(O) - \Sigma$,

2. $out(Hide_\Sigma(O)) = out(O) - \Sigma$, and

3. $int(Hide_\Sigma(O)) = int(O) \cup (acts(O) \cap \Sigma)$.

Since the hiding operation modifies only the action signature of an object (without modifying its executions or schedules), we have the following:

Lemma 12: For all automata A, execution modules E, schedule modules S, and sets of actions Σ,

1. $Execs(Hide_\Sigma(A)) = Hide_\Sigma(Execs(A))$

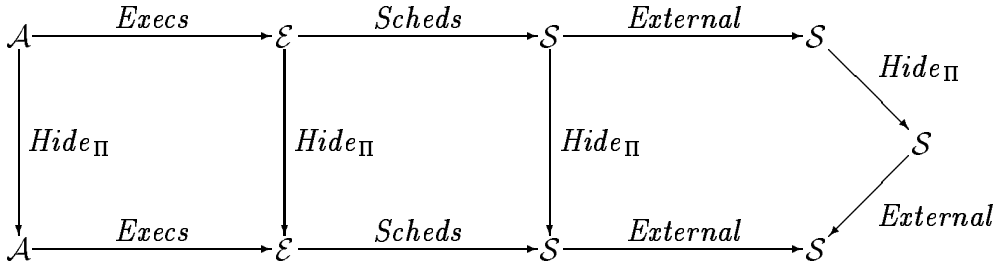
2. $Scheds(Hide_\Sigma(E)) = Hide_\Sigma(Scheds(E))$

3. $External(Hide_\Sigma(S)) = External(Hide_\Sigma(External(S)))$

Proof: Parts 1 and 2 are immediate from the definition of the hiding operation. Part 3 follows from the fact that $y|(ext(S) - \Sigma) = (y|ext(S))|(ext(S) - \Sigma)$ for every schedule y. □

As a corollary of Lemma 12, we have the following:

Corollary 13: Let $\mathcal{A}$ denote the class of automata, $\mathcal{E}$ denote the class of execution modules, and $\mathcal{S}$ denote the class of schedule modules. The following diagram commutes:



Suppose $\{O_i : i \in I\}$ are compatible objects, and consider their composition O. Suppose that π is an action of O_i not shared by O_j for every $i \neq j$. Intuitively, if π models some communication internal to the system component modeled by O_i, then whether π is hidden before or after forming the composition O should not affect the resulting object. This intuition is formalized in the following result.

Lemma 14: Let $\{O_i : i \in I\}$ be a collection of compatible objects, and let $\{\Sigma_i : i \in I\}$ be a collection of sets of actions. If $acts(O_i)$ and Σ_j are disjoint for all $i \neq j$, then $Hide_{\cup_i \Sigma_i}(\prod_{i \in I} O_i) = \prod_{i \in I} Hide_{\Sigma_i}(O_i)$.

Proof: Let $HC = Hide_{\cup_i \Sigma_i}(\prod_i O_i)$ and $CH = \prod_i Hide_{\Sigma_i}(O_i)$. First, we claim that the composition HC is defined iff CH is defined. Since for all $i \neq j$ the intersection $acts(O_i) \cap \Sigma_j$ is empty, for all $i \neq j$ we have

$$\begin{aligned} out(O_i) \cap out(O_j) &= (out(O_i) - \Sigma_i) \cap (out(O_j) - \Sigma_j) \\ &= out(Hide_{\Sigma_i}(O_i)) \cap out(Hide_{\Sigma_j}(O_j)) \end{aligned}$$

and

$$\begin{aligned} int(O_i) \cap acts(O_j) &= [int(O_i) \cup (\Sigma_i \cap acts(O_i))] \cap [acts(O_j) - \Sigma_j] \\ &= int(Hide_{\Sigma_i}(O_i)) \cap acts(Hide_{\Sigma_j}(O_j)). \end{aligned}$$

It follows that the objects O_i are compatible iff the objects $Hide_{\Sigma_i}(O_i)$ are compatible, and hence that HC is defined iff CH is defined.

Next, we claim that HC and CH have the same action signatures, and it will follow that HC and CH are equal. Notice that

$$\begin{aligned} in(HC) &= in(\prod_{i \in I} O_i) - \bigcup_{j \in I} \Sigma_j \\ &= (\bigcup_{i \in I} in(O_i) - \bigcup_{j \in I} out(O_j)) - \bigcup_{k \in I} \Sigma_k \\ &= (\bigcup_{i \in I} in(O_i) - \bigcup_{j \in I} \Sigma_j) - (\bigcup_{i \in I} out(O_i) - \bigcup_{j \in I} \Sigma_j) \\ &= \bigcup_{i \in I} (in(O_i) - \Sigma_i) - \bigcup_{j \in I} (out(O_j) - \Sigma_j) \\ &= \bigcup_{i \in I} in(Hide_{\Sigma_i}(O_i)) - \bigcup_{j \in I} out(Hide_{\Sigma_j}(O_j)) \\ &= in(\prod_{i \in I} Hide_{\Sigma_i}(O_i)) = in(CH). \end{aligned}$$

The fourth equality holds since $acts(O_i) \cap \Sigma_j$ is empty for all $i \neq j$. Similar arguments show that $out(HC) = out(CH)$ and $int(HC) = int(CH)$. Therefore, HC and CH have the same action signature, and hence are equal. □

2.1.3 Action Renaming

Our definition of composition makes the names of actions quite important. In particular, the notion of object compatibility depends entirely on the names of actions shared by

the objects. In this section, we define an operation that renames actions. With this operation, objects can be made compatible by renaming conflicting actions.

An *action mapping* f is an injective mapping between sets of actions. Such a mapping is said to be *applicable* to an object O if the domain of f contains the actions of O. Action mappings are extended to objects in the obvious way. If the action mapping f is applicable to an automaton A, then the automaton $f(A)$ is the automaton with the states and start states of A; with the input, output, and internal actions $f(in(A))$, $f(out(A))$, and $f(int(A))$, respectively; with the transition relation $\{(a, f(\pi), a') : (a, \pi, a') \in steps(A)\}$; and with the equivalence relation $\{(f(\pi), f(\pi')) : (\pi, \pi') \in part(A)\}$. Since f is injective, the partition of the locally-controlled actions of $f(A)$ is guaranteed to be an equivalence relation. Objects $f(O)$ are defined similarly for other types of objects. Such an object $f(O)$ is said to be a *renaming* of O. Since renaming affects only action names, the following result is easy to see.

Lemma 15: Let f be an action mapping applicable to the automaton A, the execution module E, and the schedule module S.

1. $Execs(f(A)) = f(Execs(A))$
2. $Scheds(f(E)) = f(Scheds(E))$
3. $External(f(S)) = f(External(S))$

In addition, since action mappings are injective, it is easy to see that actions may be hidden before or after renaming:

Lemma 16: $Hide_{f(\Sigma)}(f(O)) = f(Hide_{\Sigma}(O))$ for any object O and applicable action mapping f.

Let us consider how renaming interacts with composition. Suppose an action mapping f_i is applicable to the object O_i for every $i \in I$. First, notice that if each f_i maps some output action π_i of O_i to the action π, then the $f_i(O_i)$ are incompatible; and $\prod_i f_i(O_i)$ is not be defined even though $\prod_i O_i$ may be. Furthermore, if each f_i maps an action π to a different action π_i, then executions of $\prod_i f_i(O_i)$ may have no relationship to the executions of $\prod_i O_i$ since the objects $f_i(O_i)$ may no longer be required to synchronize on the actions π_i. We are therefore led to define a collection $\{f_i : i \in I\}$ of action mappings to be *compatible* if for all actions π_i and π_j we have $f_i(\pi_i) = f_j(\pi_j)$ iff $\pi_i = \pi_j$. We define their composition $f = \prod_i f_i$ to be the action mapping having as its domain the union of the domains of the f_i, and mapping the action π to $f_i(\pi)$ if π is in the domain of f_i. The fact that the f_i are compatible ensures that f is well-defined. It is obvious that if each f_i is applicable to an object O_i, then f is applicable to their composition. In addition, the following result verifies that the renaming of such objects may occur either before or after the formation of their composition without affecting the resulting object.

Lemma 17: Let $\{O_i : i \in I\}$ be compatible objects, and let $\{f_i : i \in I\}$ be compatible action mappings. If f_i is applicable to O_i for every $i \in I$, then $(\prod_{i \in I} f_i)(\prod_{i \in I} O_i) = \prod_{i \in I} f_i(O_i)$.

Proof: We prove the result for automata A_i; the proofs for other types of objects are similar. Let $f = \prod_i f_i$, $A = \prod_i A_i$, and $A' = \prod_i f_i(A_i)$. We show that $f(A)$ is defined iff A' is defined, and that in this case $f(A) = A'$. To do so, we must verify the following: (i) that the A_i are compatible iff the $f_i(A_i)$ are compatible, (ii) that $f(A)$ and A' have the same states and start states, (iii) that $f(A)$ and A' have the same action signature, (iv) that $f(A)$ and A' have the same transition relation, and (v) that $f(A)$ and A' have the same partition of locally-controlled actions. Since the f_i are injective mappings such that $f_i(\pi_i) = f_j(\pi_j)$ iff $\pi_i = \pi_j$, the only nontrivial part of this proof to check is part (iv). Suppose that (a, π, a') is a step of $f(A)$. For some action σ we must have that (a, σ, a') is a step of A, and that $f(\sigma) = \pi$. Furthermore, for each i, the action σ is an action of A_i iff π is an action of $f_i(A_i)$. If π is an action of $f_i(A_i)$, then σ is an action of A_i, so $(a|A_i, \sigma, a'|A_i)$ is a step of A_i and $(a|f_i(A_i), \pi, a|f_i(A_i))$ is a step of $f_i(A_i)$. If π is not an action of $f_i(A_i)$, then σ is not an action of A_i, so $a|A_i = a'|A_i$ and $a|f_i(A_i) = a'|f_i(A_i)$. In either case, (a, π, a') is a step of $A' = \prod_i f_i(A_i)$. A similar argument shows that if (a, π, a') is a step of A', then it is a step of $f(A)$. It follows that $f(A)$ and A' have the same transition relation, and hence are equal. □

2.1.4 Remarks

Since the definitions given so far have been independent of such considerations, we have chosen to ignore until this point issues of cardinality that appear in most models of computation. For example, we have not restricted our model to automata with countable sets of states and actions, and hence to countable nondeterminism. Furthermore, we have not restricted our theory to the composition of a finite (or even countable) number of automata. While these are natural restrictions (and all of the results presented thus far still hold when these restrictions are imposed), we note that Lynch and Merritt have made effective use of the composition of a countable number of automata in [LM86]. In the remainder of this thesis, we restrict our attention to automata modeling systems with a countable number of components. In particular, we restrict our attention to countable compositions, and to automata A for which $part(A)$ partitions A's locally-controlled actions into a countable number of equivalence classes. This restriction becomes relevant in the following section where we define the notion of fair computation.

2.2 Fairness

Fair computation is of central importance to distributed computation. The mutual exclusion problem, for example, has been formulated in [EM72] with the "no lockout"

condition that if every process is allowed to take steps infinitely often, then every process trying to enter its critical region will eventually do so. That is, during fair computation, every process wishing to enter its critical region will eventually do so. More generally, the specification of a distributed system typically includes conditions of the form "if condition P holds, then eventually condition Q will hold." The ability of a process to satisfy such conditions clearly depends on fair computation. In this section we show how fair computation can be described in our model, and we show how fair computation induces an interesting equivalence of automata.

2.2.1 Fair Executions

As previously mentioned, computation in a system of processes is said to be fair if every process is given the chance to make computational progress infinitely often. The phrase "given the chance" is important, since a process may not be in a position to make progress every time it is given the chance. Recall that associated with an automaton A is a partition $part(A)$ of its locally-controlled actions. Intuitively, each class of this partition consists of the locally-controlled action of a process in the system being modeled by A. A *fair execution* of an automaton A is defined to be an execution x such that the following conditions hold for each class C of $part(A)$:

1. If x is a finite execution, then no action of C is enabled from the final state of x.
2. If x is an infinite execution, then either actions from C appear infinitely often in x, or states from which no action of C is enabled appear infinitely often in x.

These conditions may be interpreted as follows. If x is finite, then computation in the system has halted since no process is able to take another step. If x is infinite, then every process has been given the chance to take a step infinitely often, although it may be that some process was unable to make computational progress every time it was given the chance to do so. Notice that this definition of fairness is essentially what is called *weak fairness* in the literature (see [Fra86], for example). As mentioned in the introduction, however, our definition is different in an important way in that it takes into consideration the notion of one process controlling the performance of an action. In particular, it is possible for an (input) action to be continuously enabled, and yet never be performed. We note in passing that our notion of fairness defines the notion of a *finite* fair computation without the usual requirement that finite computations be extended in some trivial way to infinite computations.

The set $fair(A)$ is the set of fair executions of the automaton A, and $Fair(A)$ is the execution module of A having $fair(A)$ as its set of executions.

One simple consequence of this definition of fair executions is the following.

Lemma 18: If x is a finite execution of an automaton A, then x can be extended to a fair execution $x\pi_1 a_1 \ldots$ of A (in which every π_i is a locally-controlled action of A).

Proof: Let f be a function mapping the natural numbers to the classes of $part(A)$, with the property that every class of $part(A)$ appears in the range of f infinitely often. There is an execution $x' = x\pi_1 a_1 \ldots$ of A with the property that π_i is an action from the class $f(i)$ if such and action is enabled from a_{i-1}, and an arbitrary locally-controlled action of A otherwise. (If from some state a_{i-1} no locally-controlled action of A is enabled, then x' is a finite execution ending in state a_{i-1}.) The execution x' is a fair execution of A. □

More important, however, is the next lemma which says that the fair executions of a composition are a composition of the fair executions of its components. It is for the sake of this result that we associate a partition of an automaton's locally-controlled actions with an automaton.

Lemma 19: $Fair(\prod_{i \in I} A_i) = \prod_{i \in I} Fair(A_i)$ for all compatible automata $\{A_i : i \in I\}$.

Proof: Let $FC = Fair(\prod_i A_i)$ and $CF = \prod_i Fair(A_i)$. Since both are execution modules of $A = \prod_i A_i$, both have the same states and action signature. We need only show that they have the same executions. First, however, notice that since the A_i are compatible, their locally-controlled actions are disjoint. Furthermore, notice that each A_i is input-enabled. It follows that each A_i determines when its locally-controlled actions are enabled in the composition A: If π is a locally-controlled action of A_i and a is a state of A, then π is enabled from a in A iff π is enabled from $a|A_i$ in A_i.

Suppose x is a fair execution of A, and let us show that x is an execution of CF. We must show that $x|A_i$ is a fair execution of A_i for all i. Let C be a class of locally-controlled actions of A_i, and hence a class of A. Suppose x is finite. Since x is a fair execution of A, no action of C is enabled in A from the final state a of x, and hence no action of C is enabled in A_i from the final state $a|A_i$ of $x|A_i$. Suppose x is infinite. If actions from C appear infinitely often in x, they do so in $x|A_i$. On the other hand, suppose states appear infinitely often in x from which no action of C is enabled in A. It follows that either $x|A_i$ is finite and no action of C is enabled from the final state of $x|A_i$ in A_i, or else infinitely many states of A_i appear in $x|A_i$ from which no action of C is enabled. In any case, $x|A_i$ is a fair execution of A_i. It follows that x is an execution of CF.

Conversely, suppose x is an execution of CF, and let us show that x is a fair execution of A. Let C be a class of locally-controlled actions of A, and therefore a class of A_i for some i. Since x is an execution of CF, the execution $x|A_i$ is a fair execution of A_i. Suppose x is finite, and therefore that $x|A_i$ is finite. Since $x|A_i$ is fair, no action of C is enabled from the final state of $x|A_i$, and hence no action of C is enabled from the final

state of x. Suppose x is infinite. If actions from C appear infinitely often in $x|A_i$, the same is true of x. If states appear infinitely often in $x|A_i$ from which no action of C is enabled, the same is true in x. However, $x|A_i$ may be finite. In this case, no action of C is enabled from the final state of $x|A_i$. Since x is infinite, there is a state appearing in x after which no action of C is ever enabled. In any case, x must be a fair execution of A. It follows that $FC = CF$. □

2.2.2 Fair Equivalence

In Section 2.1 we defined a notion of equivalence based on the external behavior of an object. We now define a similar notion of equivalence based on *fair* external behavior. The *fair behavior* of an automaton A, denoted by $Fbeh(A)$, is defined to be the schedule module $External(Fair(A))$. We extend this definition to objects of other types (execution modules and schedule modules) by setting $Fbeh(O) = Ubeh(O)$. It is convenient to denote the set of schedules of $Fbeh(O)$ by $fbeh(O)$, for any object O. In light of Corollary 8 and Lemma 19, we see that the fair behavior of a composition is the composition of the fair behavior of its components.

Lemma 20: $Fbeh(\prod_{i \in I} O_i) = \prod_{i \in I} Fbeh(O_i)$ for compatible objects $\{O_i : i \in I\}$.

We say that two objects O and O' are *fairly equivalent*, denoted $O \stackrel{fair}{=} O'$, if they have the same fair behavior; that is, if $Fbeh(O) = Fbeh(O')$. In light of Lemmas 10 and 20, fair equivalence satisfies the axioms stated for unfair equivalence in Lemma 11.

Lemma 21: Suppose $O = \prod_i O_i$, $P = \prod_i P_i$, $Q = \prod_i Q_i$, and $R = \prod_i R_i$ where the O_i, P_i, Q_i, and R_i are objects.

1. $O \cdot P \stackrel{fair}{=} P \cdot O$.
2. $(O \cdot P) \cdot Q \stackrel{fair}{=} O \cdot (P \cdot Q)$.
3. If $O \stackrel{fair}{=} P$ and $Q \stackrel{fair}{=} R$, then $O \cdot Q \stackrel{fair}{=} P \cdot R$ whenever the compositions $O \cdot Q$ and $P \cdot R$ are defined.

Thus, composition is commutative and associative up to fair equivalence, and fair equivalence is a weak congruence with respect to composition. With this we conclude that discussion of fairness directly related to program verification. In the remainder of this section we consider several interesting questions about how fairness is modeled in our model.

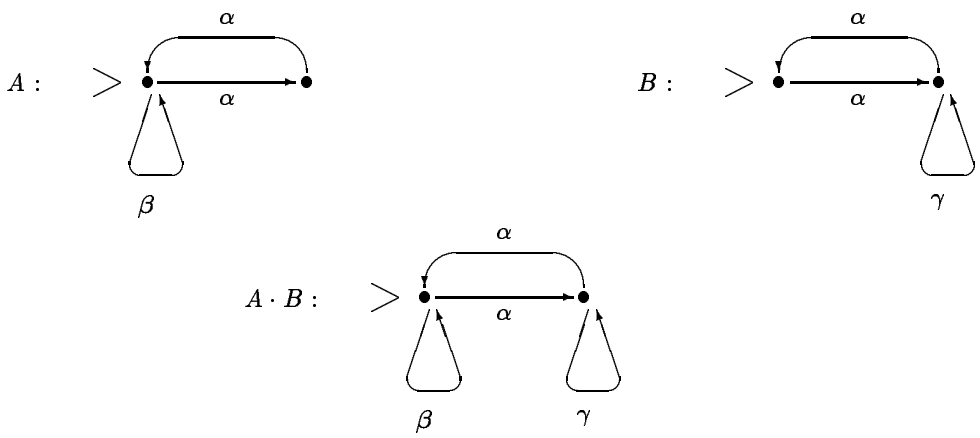


Figure 2.2: The importance of the partition of locally-controlled actions.

2.2.3 Fairness and System Decomposition

Having seen the definition of a fair execution, the role of the equivalence relation $part(A)$ associated with an automaton A is clear: The automaton models a system, and the locally-controlled actions of each system component form a separate class of the partition. It is worth considering, however, whether this partition is really of any importance. We claim that if relationships such as those stated in Lemma 20 are of importance (and we think they are), then the information about the system structure encoded in the partition of an automaton's locally-controlled actions must be retained. Suppose for a moment that we do away with the partition, so that all we know about an automaton's locally-controlled action is whether it is an internal or output action. Consider the automata A and B given in Figure 2.2, and consider their composition $A \cdot B$. Here α is an input action, and β and γ are output actions. In both automata A and B, the execution with the infinite sequence of α's as its schedule may be considered a fair execution since infinitely often each automaton passes through a state from which no locally-controlled action (either β or γ) is enabled. In the composition, however, a locally-controlled action is enabled from every state through which such an execution must pass, and yet none of these actions appear in the execution. This execution cannot be considered a fair execution of the system since the system is never allowed to make progress, even though it is able to do so at each stage of the execution. If, on the other hand, we recognize that β and γ are output actions of separate system components, we see that infinitely often each component passes through a state from which none of its locally-controlled actions is enabled. We therefore conclude that this *is* an execution of the system that is fair to all components, and hence can be considered a fair execution of the system. The partition of locally-controlled actions therefore seems to be an important component of an input-output automaton.

It is conceivable, however, that an automaton's actions can be partitioned in such a way that it is impossible for the automaton to model a system whose components have as their locally-controlled actions one class of the partition. It therefore seems possible for our intuitive understanding of an automaton's partition of its locally-controlled actions to be violated. Let us say that an automaton A is *primitive* if $part(A)$ consists of a single class. Intuitively, such an automaton can model only an "atomic" system component. It would be nice to know that every automaton A is (fairly) equivalent to a composition of primitive automata, where the locally-controlled actions of each primitive automaton form a class of A's partition. This would in effect be saying that every automaton does model a system in a way satisfying our intuition. What we can prove is the following. An automaton is said to be *deterministic* if it has one start state, and for every action π there is at most one π-step from every state.

Lemma 22: Let A be an automaton whose equivalence relation $part(A)$ partitions its locally-controlled actions into the classes $\{C_i : i \in I\}$. If A is deterministic, then there are primitive automata A_i such that C_i is the set of locally-controlled actions of A_i, and $A \stackrel{fair}{=} Hide_{int(A)}(\prod_{i \in I} A_i)$.

Proof: Since $A \stackrel{fair}{=} Hide_{int(A)}(A')$ where A' is the automaton differing from A only in that the internal actions of A are output actions of A', we may assume without loss of generality that A has no internal actions, and show that $A \stackrel{fair}{=} \prod_i A_i$. Let A_i be the primitive automaton obtained from A as follows. First, set $in(A_i) = acts(A) - C_i$ and $out(A_i) = C_i$. Second, add to A_i a dead state d. Finally, to ensure that A_i is input-enabled, if π is an input action that is not enabled from a state a, add the transition $a \stackrel{\pi}{\rightarrow} d$ from a to the dead state d. Let $B = \prod_i A_i$. We claim that $A \stackrel{fair}{=} B$.

Suppose x is a fair execution of A. Since x is also an execution of each A_i, there is an execution y of B such that $y|A_i = x$ for every i. We claim that y is a fair execution of B. If actions from C_i appear infinitely often in x, then the same is true of y. On the other hand, suppose that π is an action of C_i that is not enabled from a state a of A. Then π is an (output) action of A_i that is not enabled from the state a in A_i, and hence not from the state $\{a\}$ in B. It follows that if x is finite and no action from C_i is enabled from the final state of x, then the same is true of y; and that if x is infinite and there are infinitely many states appearing in x from which no action of C_i is enabled, then the same is true of y. Therefore, y is a fair execution of B.

Conversely, suppose y is a fair execution of B. We claim that $x = y|A_i$ is a fair execution of A for every i. We will soon show that if b is a reachable state of B, then all components $b|A_i$ of b are equal, and equal to a state other than d. From this it will follow that all $y|A_i$ are equal. Furthermore, since $x = y|A_i$, the state d must not appear in x. Since transitions to d were the only transitions added in the construction of A_i, x is an execution of A. Furthermore, since x is fair in A_i, either x is finite and no action

of C_i is enabled from the final state of x; or x is infinite and either actions of C_i appear infinitely often in x, or states appear infinitely often in x from which no action of C_i is enabled. Since this is true for every class C_i, x is must be a fair execution of A.

We now proceed by induction on the length ℓ of an execution required to reach b to show that $b|A_i = b|A_j \neq d$ for all i and j. Since A has a single start state, each A_i has the same (unique) start state, and the case of $\ell = 0$ is trivial. Suppose $\ell > 0$ and the inductive hypothesis holds for $\ell - 1$. Suppose b is reachable by an execution of length ℓ whose last transition is $b' \xrightarrow{\pi} b$. Since b' is reachable by an execution of length $\ell - 1$, the inductive hypothesis implies that $b'|A_i = b'|A_j \neq d$ for all i and j. Since π is either an input action of A or an output action of A (and hence of some A_i), there must be an automaton A_i for which no transition $b'|A_i \xrightarrow{\pi} d$ was added during its construction. It follows that $b'|A_i \xrightarrow{\pi} b|A_i$ must be a transition of A, and hence that no dead state transition was added from $b'|A_j$ during the construction of any A_j. Therefore, every step $b'|A_i \xrightarrow{\pi} b|A_i$ is a step of A. Since A is deterministic, there is only one such step, so $b|A_i = b|A_j \neq d$ for all i and j. □

This result says that our intuition (our understanding of an automaton's partition of its locally-controlled actions) is satisfied by a very restricted class of automata. It does not seem to be true, however, for arbitrary automata (although Lemma 22 does hold for arbitrary automata if fair equivalence is replaced by unfair equivalence, the proof of this using the same construction as in the proof of Lemma 22). The reason the construction given above will not work for nondeterministic automata is clear: The existence of nondeterminism allows the components to diverge during computation. Each component may then pass through states from which none of its locally-controlled actions are enabled, from which it follows that no locally-controlled actions appear in the executions generated by any of the components. Since, however, each component may pass through states from which all locally-controlled actions of all remaining components are always enabled, none of the executions generated by any of the components are fair executions of the original automaton A, whose classes are the output actions of the component automata. What is obviously required is a coordinator or scheduler S to ensure that all automata choose the same transition at every step. With this intuition in mind, we now prepare to show the following.

Theorem 23: Let A be an automaton whose equivalence relation $part(A)$ partitions its locally-controlled actions into the classes $\{C_i : i \in I\}$. There are primitive automata A_i and S such that C_i is the set of locally-controlled actions of A_i, Σ is the set of locally-controlled actions of S, and $A \stackrel{fair}{=} Hide_{int(A)\cup\Sigma}(\prod_{i\in I} A_i \cdot S)$.

The primitive automata A_i used in this construction are essentially the primitive automata used in the proof of Lemma 22. However, when the A_i perform an action, the scheduler S must be able to direct all of them to take the same step. These directions

take the form of certain input actions of the A_i, where the performance of such an action by the scheduler tells the component automata which transition they are supposed to make. We add these actions to the A_i (although initially as internal actions) with the following result.

Lemma 24: For every automaton A, there is a deterministic automaton B such that $A \stackrel{fair}{=} B$. The locally-controlled actions of B are partitioned into the classes of A, together with an additional class Σ of internal actions.

Proof: For ease of exposition, we construct a nondeterministic automaton B, and then show how it can be transformed into an equivalent deterministic automaton. The states of B are of the form (a, α) where a is a state and α is a (possibly empty) sequence of actions. The start state of B is (s, ϵ), where s is a distinguished state (not a state of A) and ϵ is the empty sequence of actions. The states of B are (a, α) and (s, α), where a is a state of A and α is a (possibly empty) sequence of actions of A. The action signature and partition of B are precisely those of A, except that an additional *scheduling action* π (an internal action) forms its own class of B's partition. The transitions of B from a state (a, α), where a is a state of A, are as follows:

$$\begin{array}{rcl} (a,\alpha) \stackrel{\pi}{\rightarrow} (a',\epsilon) \text{ in } B & \text{iff} & a \stackrel{\alpha}{\rightarrow} a' \text{ in } A \\ (a,\alpha) \stackrel{\sigma}{\rightarrow} (a,\alpha\sigma) \text{ in } B & \text{iff} & a \stackrel{\alpha\sigma}{\rightarrow} a' \text{ in } A \text{ for some } a' \end{array}$$

That is, π determines what transitions A actually makes from the state a when the sequence of actions α is actually performed. All other actions are simply recorded as actions to be performed by A at a later time. The transitions of B from a state (s, α) are as follows:

$$\begin{array}{rcl} (s,\alpha) \stackrel{\pi}{\rightarrow} (a,\epsilon) \text{ in } B & \text{iff} & a_0 \stackrel{\alpha}{\rightarrow} a \text{ in } A \text{ for some start state } a_0 \\ (s,\alpha) \stackrel{\sigma}{\rightarrow} (s,\alpha\sigma) \text{ in } B & \text{iff} & \sigma \text{ is an input action of } A \end{array}$$

In this case, only input actions and π are enabled from a state of the form (s, α). In this way, fair computation will guarantee that π is eventually performed, and hence that an initial state is chosen for A. Thus, the scheduling action π chooses the initial state of A, as well as the steps taken by A during computation. We claim that $A \stackrel{fair}{=} B$. Suppose that A's locally-controlled actions are partitioned into the classes $\{C_i : i \in I\}$. These classes together with the class $\{\pi\}$ are the classes of B.

Let x be a fair execution of A. Let y be the execution of B obtained by replacing each transition $a \stackrel{\sigma}{\rightarrow} a'$ of x by the transitions $(a, \epsilon) \stackrel{\sigma}{\rightarrow} (a, \sigma) \stackrel{\pi}{\rightarrow} (a', \epsilon)$, followed by the infinite sequence of transitions $(a, \epsilon) \stackrel{\pi}{\rightarrow} (a, \epsilon) \stackrel{\pi}{\rightarrow} \cdots$ in the case that x is a finite execution ending in the state a. Suppose x is finite. Since x is fair, no locally-controlled action is enabled in A from the final state a of x. It follows that no locally-controlled action of B

is enabled from any of the infinite occurrences of (a, ϵ) in y, except for π which occurs infinitely often. Hence, y is a fair execution of B. Conversely, suppose that x is infinite. Since x is fair, for each class C_i either actions from C_i appear infinitely often in x, or from infinitely many states appearing in x no action from C_i is enabled. In the first case, actions from C_i appear infinitely often in y. In the second case, since an action σ is enabled from a state a of A iff it is enabled from (a, ϵ) in B, infinitely many states appear in y from which no action of C_i is enabled. Since, in addition, π appears infinitely often in the execution, y must be a fair execution of B.

Conversely, let y be a fair execution of B. From the definition of B we see that if $(a, \epsilon) \stackrel{\sigma_1}{\rightarrow} (a, \sigma_1) \cdots \stackrel{\sigma_n}{\rightarrow} (a, \sigma_1 \cdots \sigma_n) \stackrel{\pi}{\rightarrow} (a', \epsilon)$ is a sequence of transitions in B, then $a \stackrel{\sigma_1}{\rightarrow} a_1 \cdots \stackrel{\sigma_n}{\rightarrow} a'$ is a sequence of transitions of A. In addition, if $(s, \epsilon) \stackrel{\sigma_1}{\rightarrow} (s, \sigma_1) \cdots \stackrel{\sigma_n}{\rightarrow} (s, \sigma_1 \cdots \sigma_n) \stackrel{\pi}{\rightarrow} (a, \epsilon)$ is a sequence of transitions in B, then $a_0 \stackrel{\sigma_1}{\rightarrow} a_1 \cdots \stackrel{\sigma_n}{\rightarrow} a$ is a sequence of transitions of A for some start state a_0 of A. Let x be the execution of A obtained by replacing every such sequence in y by the corresponding sequence of transitions of A. Since y is fair, the action π must appear infinitely often in y, and hence y must be infinite. If actions from C_i appear infinitely often in y, then the same is true in x. If not, then there are infinitely many states appearing in y from which no action of C_i is enabled. Notice that if an action σ other than π is not enabled from from the state (a, α) in B, then for all states a' of A such that $a \stackrel{\alpha}{\rightarrow} a'$ it must be that σ is not enabled from a'. It follows that either x is finite and no action of C_i is enabled from the final state of x, or there are infinitely many states appearing in x from which no action of C_i is enabled. In either case, x must be a fair execution of A.

We have just shown that $A \stackrel{fair}{=} B$. However, we are not yet done since B is not yet deterministic: There are potentially many π-steps from every state of B. However, we can assign to each π-step a unique identifier, and tag the π labeling the step with this identifier. Replacing the action π with the set Σ of newly-tagged π's, it is easy to see that this automaton is fairly equivalent to B, and hence also to A. Since this automaton *is* a deterministic automaton (with an extra class Σ of internal actions), we are done. □

We are now able to prove Theorem 23.

Proof of Theorem 23: Given the automaton A, construct the automaton B of Lemma 24. The automaton B is fairly equivalent to A, and its locally-controlled actions are partitioned into the same classes as those into which A's actions are partitioned, together with an additional class Σ of internal actions. Furthermore, B is a deterministic automaton. Lemma 22 says there are primitive automata A_i and S with $local(A_i) = C_i$ and $local(S) = \Sigma$ such that B (and hence A) is fairly equivalent to $Hide_{int(B)}(\prod_i A_i \cdot S)$, which is just $Hide_{int(A) \cup \Sigma}(\prod_i A_i \cdot S)$. □

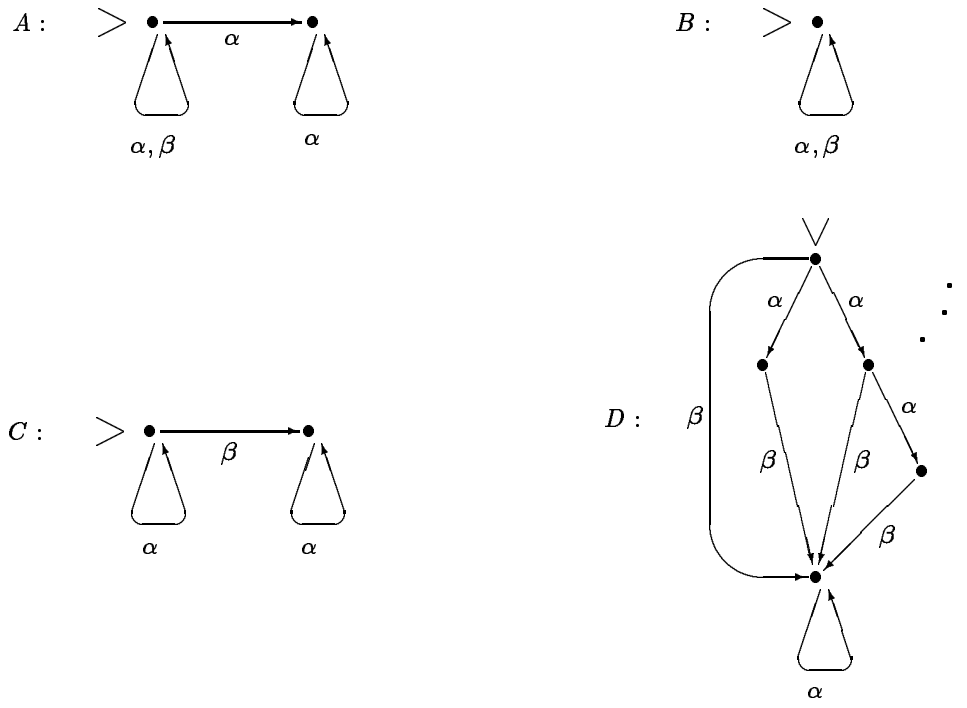


Figure 2.3: Fair equivalence and unfair equivalence are incomparable.

2.2.4 Comparing Fair and Unfair Equivalence

Having defined two types of equivalence, fair equivalence and unfair equivalence, it is natural to ask how they are related. Since $Fbeh(O) = Ubeh(O)$ when O is an execution module or schedule module, fair and unfair equivalence are identical for execution modules and schedule modules. For automata, however, they are incomparable.

Consider, for example, the automata of Figure 2.3. The (primitive) automata A and B each have an input action α and an output action β. The unfair behavior of both A and B consists of all sequences of α and β, so A and B are unfairly equivalent. The fair behavior of A, however, includes the infinite sequence of α's. Since the fair behavior of B does not, A and B are fairly inequivalent. On the other hand, C and D are two (nonprimitive) automata with output actions α and β, each forming a separate class in the partition of the locally-controlled actions. The fair behavior of C and D consist of finite sequences of α's followed by a β and an infinite sequence of α's, so C and D are fairly equivalent. The unfair behavior of C, however, includes the infinite sequence of α's. Since the unfair behavior of D does not, C and D are unfairly inequivalent.

Thus, in general, fair equivalence and unfair equivalence are incomparable. The

following lemma, however, indicates that fair equivalence implies unfair equivalence in the case of primitive automata. Since the primitive automata A and B of Figure 2.3 are unfairly equivalent but not fairly equivalent, we see that fair equivalence is a stronger equivalence that unfair equivalence in the case of primitive automata.

Lemma 25: Let A and B be two primitive automata. If A and B are fairly equivalent, then A and B are unfairly equivalent.

Proof: It is enough to check that $scheds(A)|ext(A) = scheds(B)|ext(B)$. Suppose x is an execution of A. If an infinite number of locally-controlled actions appear in x, then since A is a primitive automaton (with a single class of locally-controlled actions), x is a fair execution of A. Since A and B are fairly equivalent, there is a fair execution y of B such that $sched(x)|ext(A) = sched(y)|ext(B)$. On the other hand, if only a finite number of locally-controlled actions appear in x, then we may write $x = x'x''$ where x' is a finite execution of A, and every locally-controlled action appearing in x appears in x'. By Lemma 18, the finite execution x' can be extended to a fair execution z of A. Since A and B are fairly equivalent there is a fair execution y of B such that $sched(z)|ext(A) = sched(y)|ext(B)$. Thus, there is a finite execution y' of B, a prefix of y, such that $sched(x')|ext(A) = sched(y')|ext(B)$. Since B is input enabled and no locally-controlled action appears in x after x', y' may be extended to an execution y'' of B such that $sched(x)|ext(A) = sched(y'')|ext(B)$. Thus, $scheds(A)|ext(A) \subseteq scheds(B)|ext(B)$. Since the opposite containment follows by a symmetric argument, we are done. □

2.3 Hierarchical Correctness Proofs

The problem motivating this thesis is the construction of hierarchical correctness proofs for distributed algorithms. We have already mentioned in the introduction how such a proof might be constructed. First, a sequence of models $O_1, \ldots, O_n$ are defined, objects of some type modeling the algorithm at decreasing levels of abstraction. Each model O_i is then shown to "simulate" O_{i-1} in some appropriate sense of the word "simulate." In such a proof, each O_{i-1} can be viewed as the statement of a problem O_i is required to solve. O_i may be said to solve the problem specified by O_{i-1} if every behavior of O_i is a behavior of O_{i-1}. O_i solves the problem specified by O_{i-1} in the sense that every correctness condition satisfied by each behavior of O_{i-1} is also satisfied by each behavior of O_i. However, as previously mentioned, the satisfaction of certain liveness conditions depends on fair computation. We therefore require only that every fair behavior of O_i be a fair behavior of O_{i-1}. That is, O_i is said to *satisfy* O_{i-1} if $fbeh(O_i) \subseteq fbeh(O_{i-1})$. We also require that O_i and O_{i-1} have the same external action signature.

Notice, however, that this notion of correctness is not completely satisfactory. In particular, a schedule module O_i with *no* schedules trivially satisfies every problem O_{i-1}

(with the same external action signature). Furthermore, since the schedules of O_i are allowed to be arbitrary sequences of actions, it is conceivable that they may encode information allowing the solution of undecidable problems, and hence not be behaviors of an implementable system. In an attempt to avoid such anomalies, we say that the object O_{i-1} is *implementable* if there is an automaton satisfying O_{i-1}. The object O_{i-1} is implementable in the sense that there is a system satisfying every correctness condition satisfied by O_{i-1}. Furthermore, since O_{i-1} is satisfied by an automaton, and since every automaton is input-enabled, the object O_{i-1} must describe a response to every possible pattern of input. That is, the behavior of O_{i-1} is nontrivial. We say that O_{i-1} *solves* O_i if O_{i-1} is an implementable object satisfying O_i. In the context of constructing hierarchical correctness proofs, such a proof consists of a sequence $O_1, \ldots, O_n$ of objects, and the verification that each O_i solves O_{i-1}.

Clearly, the notion of satisfaction is the basis of each of these definitions. The remainder of this section concerns techniques for verifying that one object satisfies another. Two properties of satisfaction are very easy to see. The first is that satisfaction is transitive, and a weak congruence with respect to composition.

Lemma 26: Consider the objects O_i, P_i, and Q_i, for $i \in I$.

1. If O_i satisfies P_i and P_i satisfies Q_i, then O_i satisfies Q_i.
2. If O_i satisfies P_i for every $i \in I$, then $\prod_i O_i$ satisfies $\prod_i P_i$ whenever the compositions $\prod_i O_i$ and $\prod_i P_i$ are defined.

Proof: The proof of the first part is immediate from the definition of satisfaction. The second part requires some proof. As a result of Corollary 8, the external action signature of $\prod_i O_i$ is the composition of the external action signatures of the O_i, and similarly for $\prod_i P_i$. Since O_i and P_i have the same external action signature for all $i \in I$, so do $\prod_i O_i$ and $\prod_i P_i$. Since $\mathit{fbeh}(O_i) \subseteq \mathit{fbeh}(P_i)$ for all $i \in I$, it follows by Lemma 20 that $\mathit{fbeh}(\prod_i O_i) \subseteq \mathit{fbeh}(\prod_i P_i)$. Therefore, $\prod_i O_i$ satisfies $\prod_i P_i$. □

A second property of satisfaction is its invariance under action renaming.

Lemma 27: Let f be an action mapping applicable to the objects O and P. If O satisfies P, then $f(O)$ satisfies $f(P)$.

Proof: Since O and P have the same external action signature and since f is injective, $f(O)$ and $f(P)$ have the same external action signature. Using Lemma 15 we see that $\mathit{fbeh}(f(O)) \subseteq \mathit{fbeh}(f(P))$. Thus, $f(O)$ satisfies $f(P)$. □

While we have repeatedly indicated that our hierarchical correctness proofs consist of a sequence of objects $O_1, \ldots, O_n$ modeling an algorithm at different levels of abstraction,

our proofs typically have more structure than this. In the proof of Schönhage's resource arbiter (in the next chapter), for example, we actually construct for each level of abstraction an automaton A_i describing the algorithm at the appropriate level of abstraction. This automaton describes as much of the algorithm as can be described by its static nature. In particular, the automaton A_i encodes all safety conditions required. If liveness conditions are required, we construct an execution module E_i of A_i with those executions of A_i satisfying the desired liveness conditions. The objects O_i referred to above are actually the execution modules E_i. We note, however, that the execution module E_n at the lowest level of abstraction typically consists of the fair executions of A_n. Thus, at the lowest level of abstraction the protocol is completely described by an automaton, and we could use the object A_n in place of the execution module E_n in the correctness proof. Since automata and execution modules are the types of objects most frequently used in correctness proofs, in the remainder of this section we give techniques for proving the satisfaction of one automaton or execution module by another.

2.3.1 Automaton Satisfaction

We now describe one method for proving that an automaton A satisfies an automaton B. This method makes use of the notion of a possibilities mapping, a correspondence between the states of the two automata that can be used to prove that A satisfies B.

Suppose A and B are automata with the same external action signature, and suppose h is a mapping from $states(A)$ to the power set of $states(B)$. The mapping h is said to be a *possibilities mapping* from A to B if the following conditions hold:

1. For every start state a of A, there is a start state b of B such that $b \in h(a)$.
2. For every reachable state a of A, every step (a, π, a') of A, and every reachable state $b \in h(a)$ of B:
   (a) If $\pi \in acts(B)$, then there is a step (b, π, b') of B such that $b' \in h(a')$.
   (b) If $\pi \notin acts(B)$, then $b \in h(a')$.

If a is a state of A, then a state $b \in h(a)$ of B is referred to as a *possibility* for a. Informally, b is an abstract state corresponding to the less abstract state a. The fact that h maps a to a *set* of possibilities allows for the chance that many abstract states may correspond to the single concrete state a. The first condition of a possibilities mapping says that every start state of A has as one of its possibilities a start state of B. The second condition says that steps A and B preserve possibilities: If b is a possibility for a, then for every step (a, π, a') of A either b is also a possibility for a', or there is a step (b, π, b') of B with the property that b' is a possibility for a. This definition generalizes the definition of a possibilities mapping used in the context of Event-State Algebras in [Lyn83]. It is

also reminiscent of the notion of bisimulation from CCS presented in [Mil80]. Roughly speaking, a possibilities mapping from A to B is a mapping from the states of A to the states of B with the property that if a corresponds to b, and if A can make a transition via the action π from a to a', then B can make a transition via the action π from b to a state b' corresponding to a'. Milner's notion of bisimulation is essentially a pair of possibilities mappings, one from A to B and another from B to A.

We now show how to use a possibilities mapping to prove that A satisfies B. Our first step is to show how such a mapping relates the executions of A to the executions of B. Given two finite executions x and y of A and B, respectively, we say that y *finitely corresponds* to x under h if $sched(y) = sched(x)|B$ and the final state of y is a possibility for the final state of x. In general, if x and y are two executions of A and B, we say that y *corresponds* to x under h if for every finite prefix $x_i = a_0\pi_1a_1 \ldots a_i$ of x there is a finite prefix y_i of y finitely corresponding to x_i under h such that y is the limit of the y_i. Informally, the executions x and y model the same computation at different levels of abstraction. Our next result shows that by inductively constructing the y_i it is always possible to construct such an execution y.

Lemma 28: Let h be a possibilities mapping from A to B. If x is an execution of A, then there is an execution y of B corresponding to x under h.

Proof: Let $x = a_0\pi_1a_1 \ldots$. For each $i \geq 0$, let $x_i = a_0\pi_1a_1 \ldots a_i$. We construct the finitely corresponding y_i inductively, and take y to be the limit of the y_i. Since a_0 is a start state of A, the set $h(a_0)$ must contain a start state of B, and hence it is easy to choose an execution y_0 finitely corresponding to x_0 under h. Suppose y_{i-1} finitely corresponds to x_{i-1} under h, and let us construct y_i. First, a_{i-1} is a reachable state of A, and (a_{i-1}, π_i, a_i) is a step of A. Second, the final state b of y_{i-1} is a reachable state of B in $h(a_{i-1})$. If $\pi_i \in acts(B)$, then by the definition of h there is a state b' in $h(a_i)$ such that (b, π_i, b') is a step of B. If $y_i = y_{i-1}\pi_i b'$, then the final state of y_i is in $h(a_i)$ and $sched(x_i)|B = sched(y_i)$. If $\pi_i \notin acts(B)$, then from the definition of h we see that $b \in h(a_i)$. If $y_i = y_{i-1}$, then the final state of y_i is in $h(a_i)$ and $sched(x_i)|B = sched(y_i)$. In either case, y_i finitely corresponds to x_i under h. □

Since each pair of prefixes x_i and y_i satisfies the condition $sched(x_i)|B = sched(y_i)$, it is easy to see that the executions x and y do so as well.

Lemma 29: Let h be a possibilities mapping from A to B. If the execution y of B corresponds to the execution x of A under h, then $sched(x)|B = sched(y)$.

Proof: Suppose that $sched(x)|B \neq sched(y)$. Since x and y are the limits of finitely corresponding prefixes x_i and y_i, respectively, there must be an i such that $sched(x_i)|B \neq sched(y_i)$. However, since y_i finitely corresponds to x_i under h, this is impossible. Thus, $sched(x)|B = sched(y)$. □

Having established a correspondence between the executions of A and B, we show with the following result how this correspondence can be used to show that A satisfies B. We say that one equivalence relation is a *contained in* a second if every class of the first is contained in a class of the second.

Lemma 30: Let A and B be automata such that $part(B)$ is contained in $part(A)$. Let h be a possibilities mapping from A to B. Suppose the following condition holds for all reachable states a of A and for all classes C and D of $part(A)$ and $part(B)$, respectively, such that $C \supseteq D$: If an action of D is enabled from a reachable state of $h(a)$, then an action of D is enabled from a and no action of $C - D$ is enabled from a.

Proof: Since h is a possibilities mapping from A to B, both automata have the same external action signature. We need only show that $fbeh(A) \subseteq fbeh(B)$. Let x be a fair execution of A, and let y be an execution of B corresponding to x under h. We claim that y is a fair execution of B. Since $sched(x)|B = sched(y)$ and $ext(A) = ext(B)$, we will have that $sched(x)|ext(A) = sched(y)|ext(B)$, and hence that $fbeh(A) \subseteq fbeh(B)$. For each $i \geq 0$, let x_i be the prefix $a_0\pi_1 a_1 \ldots a_i$ of x, and let y_i be the prefix of y finitely corresponding to x_i under h.

Suppose y is finite. Suppose there is a class D of B such that an action of D is enabled from the final state of y. Since y is finite, $y = y_i$ for some i. Since an action of D is enabled in B from a reachable state in $h(a_j)$ for all $j \geq i$ (namely, the final state of y), for all $j \geq i$ an action from D is enabled in A from a_j, and no action from $C - D$ is enabled in A from a_j. If x is finite, then an action of C is enabled from the final state of x. If x is infinite, then from every state a_j $(j \geq i)$ an action of C is enabled and yet no action of C is performed (or it would appear in y). In either case, this contradicts our initial assumption that x is a fair execution, so y must be a fair execution of B.

Conversely, suppose y is infinite. Suppose there is a class D such that an action from D is enabled from all but finitely many states appearing in y. It follows that for all but finitely many i, an action of D is enabled from a reachable state of $h(a_i)$ in B. Therefore, for all but finitely many i, there is an action of D enabled from a_i in A, and no action from $C - D$ enabled from a_i. Since x is a fair execution of A, there must be infinitely many actions from D appearing in x, and hence in y. Therefore, y must be a fair execution of B. □

We remark that the requirement that $part(B)$ be contained in $part(A)$ is not unreasonable when B models an algorithm at a higher level of abstraction than A. The restriction implies that the actions of B are a subset of the actions of A. Since A and B have the same external action signature (h is a possibilities mapping from A to B), this implies that some low-level internal actions of A may not be internal actions of B. Even when this requirement is not met, however, the correspondence between states established by a possibilities mapping is still a useful correspondence when reasoning about the behavior

of the automaton. For example, in Section 2.3.2 we will see how this correspondence can be used to verify that one execution module (of an automaton) satisfies a second.

Our final result concerning possibilities mappings shows that possibilities mappings have a very nice local behavior: Given two automata $A = \prod_i A_i$ and $B = \prod_i B_i$ together with a possibilities mapping from A_i to B_i for every i, these possibilities mappings induce a possibilities mapping from A to B.

Lemma 31: Suppose for all $i \in I$ that h_i is a possibilities mapping from A_i to B_i, and that $acts(A_i) \supseteq acts(B_i)$. Let $A = \prod_i A_i$ and $B = \prod_i B_i$. If h is the mapping from $states(A)$ to the power set of $states(B)$ defined by $h(a) = \{b : b|B_i \in h_i(a|A_i)\}$, then h is a possibilities mapping from A to B.

Proof: As a result of Corollary 8, the external action signature of a composition is the composition of the external action signatures of its components. Since the A_i and B_i have the same external action signatures, A and B must also have the same external action signature. Thus, we need only check that conditions 1 and 2 of a possibilities mapping hold. For the first condition, for every $a_i \in start(A_i)$ there is a $b_i \in states(B_i)$ such that $b_i \in h_i(a_i)$. Thus, for every $a \in start(A)$ there is a $b \in start(B)$ such that $b \in h(a)$. For the second condition, suppose that a is a reachable state of A, (a, π, a') is a step of A, and $b \in h(a)$ is a reachable state of B. Let $a_i = a|A_i$, $a'_i = a'|A_i$, and $b_i = b|B_i$ for every $i \in I$. Notice that, since a and b are reachable states of A and B, a_i and b_i must be reachable states of A_i and B_i.

Suppose that $\pi \in acts(B)$. We must construct a step (b, π, b') of B with $b' \in h(a')$. Suppose $\pi \in acts(B_i)$. Then $\pi \in acts(A_i)$, so (a_i, π, a'_i) must be a step of A_i. Since h_i is a possibilities mapping from A_i to B_i, there is a step (b_i, π, b'_i) of B_i with $b'_i \in h_i(a'_i)$. Suppose $\pi \notin acts(B_i)$. If $\pi \in acts(A_i)$, then (a_i, π, a'_i) is a step of A_i, and $b_i \in h_i(a'_i)$ by definition of h_i. If $\pi \notin acts(A_i)$, then $a_i = a'_i$, and so $b_i \in h_i(a_i) = h_i(a'_i)$. In either case, let $b'_i = b_i$. It follows that (b_i, π, b'_i) is a step of B_i if $\pi \in acts(B_i)$, and that $b_i = b'_i$ if $\pi \notin acts(B_i)$. If b is the state of B such that $b'_i = b'|B_i$ for all i, then (b, π, b') is a step of B. Furthermore, $b' \in h(a')$ as desired.

Suppose that $\pi \notin acts(B)$. Then $\pi \notin acts(B_i)$ for all i. As above, $b_i \in h_i(a'_i)$ for all i, and so $b \in h(a')$ as desired. Thus, h is a possibilities mapping from A to B. □

2.3.2 Execution Module Satisfaction

As previously mentioned, when constructing the correctness proof of an algorithm, we first construct automata $A_1, \ldots, A_n$ describing the algorithm at several levels of abstraction. If the algorithm is required to satisfy certain liveness conditions, we also construct execution modules E_i of A_i describing these liveness conditions. The remainder of the

correctness proof consists of proving that each E_i satisfies E_{i-1}. We now show how possibilities mappings can be used to prove that certain execution modules satisfy other execution modules.

We remark that one correctness condition common to many system specifications is a condition of the form "if condition P holds, then eventually condition Q holds." Lamport denotes this temporal logic statement $\Box(P \supset \Diamond Q)$ by $P \rightsquigarrow Q$ in [Lam77], read "P leads to Q." Given an automaton A, a set of states S, and a set of actions T, a simple correctness condition common to specifications in our model (see Chapter 3, for instance) is the condition "if the current state of A is contained in S, then eventually an action of T will be performed." With Lamport's notation in mind, we denote this condition by $S \hookrightarrow T$.[1] Given two execution modules E and F satisfying a collection of such conditions, we now show how a possibilities mapping can be used to show that E satisfies F. We begin with a result relating individual executions.

Lemma 32: Let h be a possibilities mapping from A to B. Let x be an execution of A, and let y be an execution of B corresponding to x under h.

1. If y satisfies $U \hookrightarrow V$, and if $h(S) \subseteq U$ and $T \supseteq V$, then x satisfies $S \hookrightarrow T$.
2. If x satisfies $S \hookrightarrow T$, and if $S \supseteq h^{-1}(U)$ and $T \subseteq V$, then y satisfies $U \hookrightarrow V$.

Proof: Let $x = a_0\pi_1a_1\ldots$, and let $y = b_0\varphi_1b_1\ldots$. For each $i \in I$, let $x_i = a_0\pi_1a_1\ldots a_i$, and let y_i be the prefix of y finitely corresponding to x_i under h.

Suppose y satisfies $U \hookrightarrow V$, and let us show that x satisfies $S \hookrightarrow T$. It is enough to show that if $a_k \in S$, then $\pi_\ell \in T$ for some $\ell > k$. Since y_k finitely corresponds to x_k under h, we have $y_k = b_1\varphi_1b_1\ldots b_m$ with $b_m \in h(a_k)$ for some m. Since $a_k \in S$ and $h(S) \subseteq U$, we have $b_m \in U$. Since y satisfies $U \hookrightarrow V$, we have $\varphi_n \in V$ for some $n > m$. Since $V \subseteq T$, for some $\ell > k$ we have $sched(x_\ell)|B = sched(y_n)$ where $sched(x_\ell)|B$ and $sched(y_n)$ both end with φ_n. Therefore, for some $\ell > k$ we have $\pi_\ell = \varphi_m \in T$, as desired.

Conversely, suppose x satisfies $S \hookrightarrow T$, and let us show that $U \hookrightarrow V$ is satisfied by y. It is enough to show that if $b_k \in U$, then $\varphi_\ell \in V$ for some $\ell > k$. Since $y_m = b_0\varphi_1\ldots b_k$ finitely corresponds to $x_m = a_0\pi_1\ldots a_m$ for some m, we have $b_k \in h(a_m)$. Since $b_k \in U$ and $h^{-1}(U) \subseteq S$, we have $a_m \in S$. Since x satisfies $S \hookrightarrow T$, for some $n > m$ we have $\pi_n \in T$. Since $sched(x_n)|B = sched(y_n)$ and $T \subseteq V \subseteq acts(B)$, we see that the final action of y_n is π_n. If $y_n = b_0\ldots\varphi_\ell b_\ell$, then $\varphi_\ell = \pi_n \in V$ for some $\ell > k$ as desired. $\Box$

With this result, we are now able to give the following sufficient condition for the satisfaction of one execution module by another.

[1]The statement $S \hookrightarrow T$ is essentially a statement in temporal logic, as is $\Box(P \supset \Diamond Q)$. The fact that executions are sequences of states and actions, instead of simply infinite sequences of states, means the standard model for temporal logic must be slightly modified if the condition $S \hookrightarrow T$ is to be expressed in temporal logic.

Lemma 33: Let h be a possibilities mapping from A to B. Let E be the execution module of A with the executions of A satisfying the conditions $S_i \hookrightarrow T_i$ for every $i \in I$, and let F be the execution module of B with the executions of B satisfying the conditions $U_i \hookrightarrow V_i$ for every $i \in I$. If for every $i \in I$ we have that $S_i \supseteq h^{-1}(U_i)$ and $T_i \subseteq V_i$, then E satisfies F.

Proof: Since h is a possibilities mapping from A to B, these automata (and hence the execution modules E and F) have the same external action signature. Let x be an execution of E, and let y be an execution of B corresponding to x under h. Since x satisfies $S_i \hookrightarrow T_i$ for every i, Lemma 32 implies that y satisfies $U_i \hookrightarrow V_i$ for every i. It follows that y is an execution of F. Therefore, $\mathit{fbeh}(E) \subseteq \mathit{fbeh}(F)$, and E satisfies F. □

We conclude with a simple result relating conditions of the form $S \hookrightarrow T$ satisfied by executions of a composition of automata to conditions of the form $S' \hookrightarrow T'$ satisfied by executions of an individual component.

Lemma 34: Let $A = \mathit{Hide}_{\Sigma}(\prod_i A_i)$. Let $S \subseteq \mathit{states}(A)$, and let $S_j = \{s|A_j \ : \ s \in S\}$. If x is an execution of A, then x satisfies the $S \hookrightarrow T$ iff $x|A_j$ satisfies $S_j \hookrightarrow T$.

Chapter 3

An Example

As an example of the hierarchical organization of correctness proofs proposed in the preceding chapter, in this chapter we prove the correctness of Schönhage's distributed resource allocation algorithm described in the introduction. The problem is to design an arbiter allocating a resource among a collection of users that guarantees the *mutual exclusion* condition that at most one user is using the resource at any given time; and the *no lockout* condition that if users holding the resource eventually return the resource, then the arbiter will eventually satisfy every requesting user. The distributed system in which this arbiter is to be used is completely asynchronous: processor speeds may be independent; messages may take an arbitrary, finite amount of time to be delivered; and messages may be delivered in any order.

The arbiter itself is described in parallel with the proof of its correctness. We begin with a high-level model serving as a simple specification of the problem the arbiter is to solve. We then give a graph-theoretic description of the algorithm's global behavior. Finally, the arbiter is distributed and described in terms of a low-level protocol to be followed by the processors comprising the arbiter. We show that this low-level model solves the high-level problem specification, and hence that the given protocol is a correct solution to the arbiter's problem specification.

3.1 The Automaton A_1

Our high-level model of the arbiter, the automaton A_1, is a very simple specification of the arbiter's correctness conditions. We refer to the arbiter itself as a, and to the users of the arbiter as $u_1, \ldots, u_n$.[1]

[1] In general, we will denote entities associated with the arbiter by the letter a, and entities associated with the users by letter u. Letters near the end of the alphabet such as v and w will be used to denote entities associated with either the arbiter or the users.

Input Actions:
    $request(u)$
        effects:
            $requesters \leftarrow requesters \cup \{u\}$
    $return(u)$
        effects:
            if $holder = u$ then
                $holder \leftarrow a$
Output Actions:
    $grant(u)$
        preconditions:
            $u \in requesters$
            $holder = a$
        effects:
            $requesters \leftarrow requesters - \{u\}$
            $holder \leftarrow u$

Figure 3.1: The actions of A_1.

3.1.1 The States of A_1

A state of A_1 consists of a set $requesters \subseteq \{u_1, \ldots, u_n\}$ of requesting processes, together with a value $holder \in \{u_1, \ldots, u_n, a\}$ indicating the entity currently holding the resource (either a user or the arbiter itself). The start state of A_1 is the state in which the set *requesters* of requesting users is empty, and the initial *holder* is the arbiter a itself. We note that all states of A_1 are reachable, as will become clear when the actions of A_1 have been introduced.

3.1.2 The Actions of A_1

The actions of A_1 are given in Figure 3.1. We specify the transition relation of an automaton by giving for each action a list of preconditions and effects. An action is enabled from any state s satisfying the action's preconditions, and the action takes s to the state t if t can be obtained by modifying s as indicated by the action's effects. Since input actions are enabled from every state, we omit the preconditions of input actions.

The input actions of A_1 are of the form $request(u)$ and $return(u)$, where u is a user. The action $request(u)$ simply places the user u in the set *requesters* of requesting users. Since automata are input-enabled, a user is able to request the resource at any time, even when it is currently holding the resource. The effect of a user's requesting the resource while holding the resource is that the request is recorded for later use (later servicing of the user). The action $return(u)$ returns the resource to the arbiter by making the

arbiter the new holder of the resource. Notice that if a (faulty) user tries to return the resource when it does not actually hold it, the arbiter simply ignores the "return." The automaton A_1 has no internal actions. The output actions of A_1 are of the form $grant(u)$, where u is again a user. The arbiter grants the resource to u with the action $grant(u)$, which removes u from the set of requesting users and makes u the new holder of the resource. Notice that the arbiter grants the resource *only* when the arbiter actually holds the resource. Consequently, at most one user is using the resource at any time.

3.1.3 The Execution Module E_1

While the executions of A_1 satisfy the mutual exclusion condition that at most one user is using the resource at any given time, we must still ensure the no lockout condition is satisfied by the arbiter: If users using the resource eventually return the resource to the arbiter, then the arbiter eventually satisfies every request for the resource. Let u be a user node, and let us define the following sets of states and actions.[2]

$$\begin{aligned} RtnRes_1^s(u) &= \{s \in states(A_1) : holder = u \text{ in } s\} \\ RtnRes_1^a(u) &= \{return(u)\} \end{aligned}$$

$$\begin{aligned} GrRes_1^s(u) &= \{s \in states(A_1) : u \in requesters \text{ in } s\} \\ GrRes_1^a(u) &= \{grant(u)\} \end{aligned}$$

The condition

$$RtnRes_1 = \bigwedge_u RtnRes_1^s(u) \hookrightarrow RtnRes_1^a(u)$$

says that any user holding the resource will eventually return the resource to the arbiter. The condition

$$GrRes_1 = \bigwedge_u GrRes_1^s(u) \hookrightarrow GrRes_1^a(u)$$

says that any user requesting the resource will eventually be granted the resource. The correctness condition

$$C_1 = RtnRes_1 \supset GrRes_1$$

says that if users holding the resource always return the resource, then users requesting the resource will always be granted the resource. This is precisely the no lockout condition we require the arbiter to satisfy. We denote by E_1 the execution module of A_1 with the executions of A_1 satisfying the condition C_1. The execution module E_1 serves as the specification of the arbiter.

[2] We will be defining several correctness conditions for each of the models we study. We will subscript these conditions to indicate the level of abstraction with which they are associated. Furthermore, the sets of states and actions used to construct these conditions will be superscripted with the letters s or a, respectively.

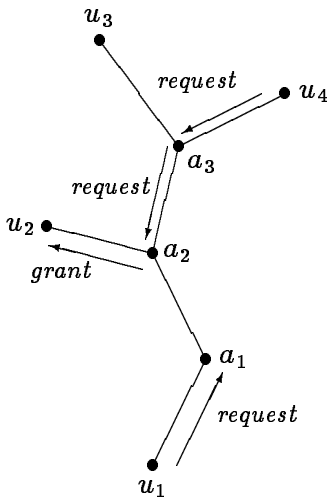


Figure 3.2: One state of the arbiter modeled by A_2.

3.2 The Automaton A_2

Our next model reveals the distributed structure of the arbiter, but still at a high level of abstraction, a level at which one might describe the algorithm at the blackboard. In this model, illustrated in Figure 3.2, the arbiter and its environment are modeled by a connected, acyclic graph G. The leaves of G are *user nodes* representing the users, labeled $u_1, \ldots, u_n$. The arbiter itself consists of the remaining *arbiter nodes*, labeled $a_1, \ldots, a_m$. The (directed) edge of G from the node v to w is denoted by $\langle v, w \rangle$. An edge $\langle v, w \rangle$ is said to *point* toward a node x if $\langle v, w \rangle$ is an edge in the path from v to x. Arrows are placed on edges of the graph to indicate either a request for the resource or the granting of the resource. In general, the resource is considered to be held by a node at the head of a *grant* arrow. Such a node is called a *root* of the graph. A user u requests the resource by placing a *request* arrow on the edge $\langle u, a \rangle$ from itself to the adjacent arbiter node a. The arbiter grants the resource to u by removing this arrow and placing a *grant* arrow on $\langle a, u \rangle$. The user then returns the resource by moving the *grant* arrow from the edge $\langle a, u \rangle$ to the edge $\langle u, a \rangle$. The arbiter itself, however, is an acyclic graph of arbiter nodes. When the head of a *request* arrow is placed at an arbiter node a, the arbiter node's response depends on whether it is holding the resource. If the arbiter node a holds the resource, then it must be at the head of a *grant* arrow, and so there must be a *grant* arrow on some edge $\langle v, a \rangle$. The arbiter selects the first node w in some fixed ordering of its adjacent nodes having a *request* arrow on $\langle w, a \rangle$. The arbiter then grants the resource to this node by removing the *request* arrow and moving the *grant* arrow to the edge $\langle a, w \rangle$. In this case we say that the resource has been *forwarded* by a to w. If the arbiter node a does not hold the resource, then the arbiter *forwards* the request in the direction of a

node holding the resource by placing a *request* on the edge pointing toward a root. The work in this section holds for arbitrary connected, acyclic graphs. When we consider the model A_3 in the following section, however, we will restrict our attention to graphs with a particular structure.

3.2.1 The States of A_2

In order to refer conveniently to the arrows on an edge of the graph, we associate with each edge $\langle v, w \rangle$ an *arrow set*, $arrows(v, w)$, containing all of the arrows on the edge $\langle v, w \rangle$. A state of A_2 therefore consists of one arrow set, $arrows(v, w)$, for every edge $\langle v, w \rangle$ of the graph G. The start states of A_2 are taken from the set of states in which a single arrow set $arrows(v, a)$ contains only a *grant* arrow, and all other arrow sets are empty, where a is an arbiter node of the graph G. In such a state, the arbiter holds the resource and no requests for the resource are pending. We will soon restrict our attention to a particular set of such start states in the next section, but the work of this section is independent of the particular set chosen. We note that some states of A_2 are unreachable. For technical convenience, we remove these states from A_2 so that all states of A_2 are reachable.

3.2.2 The Actions of A_2

Fix for each node of G an (arbitrary) ordering of its adjacent nodes. Let (v, w) denote the set of nodes properly between the nodes v and w in this ordering, and let $(v, w]$ denote the set nodes properly between v and w together with the node w. The actions of A_2 are given in Figure 3.3. The input actions are of the form $request(u, a)$ and $grant(u, a)$, and the output actions are of the form $grant(a, u)$, where u is a user node and a is an adjacent arbiter node. The internal actions are of the form $request(a, u)$ where u is a user node and a is an adjacent arbiter node; and of the form $request(a, a')$ and $grant(a, a')$ where a and a' are adjacent arbiter nodes. As in the previous model, users may request or grant the ticket at any time, but grants by users not actually holding the ticket are effectively ignored. Note we have added internal actions with which the arbiter may request that the user return the resource. The arbiter had no such ability in the previous model. These actions have been added for the sake of symmetry. Having been added as internal actions, they have no effect on the arbiter's interface with its users.

The next few results state certain invariants that hold during executions of A_2. The first guarantees that every state contains at most one root, and hence that at most one user is using the resource at any time.

Lemma 35: If s is a state of A_2, there is exactly one root in s.

Input Actions:

$request(u, a)$

effects:

$arrows(u, a) \leftarrow arrows(u, a) \cup \{request\}$

$grant(u, a)$

effects:

if $grant \in arrows(a, u)$ then

$arrows(a, u) \leftarrow arrows(a, u) - \{request\}$

$arrows(a, u) \leftarrow arrows(a, u) - \{grant\}$

$arrows(u, a) \leftarrow arrows(u, a) \cup \{grant\}$

Internal and Output Actions:

$request(a, v)$

preconditions:

$request \in arrows(w, a)$ for some w

$\langle a, v \rangle$ points toward a root

$request \notin arrows(a, v)$

effects:

$arrows(a, v) \leftarrow arrows(a, v) \cup \{request\}$

$grant(a, v)$

preconditions:

$request \in arrows(v, a)$

$grant \in arrows(w, a)$ for some w

$request \notin arrows(y, a)$ for $y \in (w, v)$

effects:

$arrows(v, a) \leftarrow arrows(v, a) - \{request\}$

$arrows(w, a) \leftarrow arrows(w, a) - \{grant\}$

$arrows(a, v) \leftarrow arrows(a, v) \cup \{grant\}$

Figure 3.3: The actions of A_2.

Proof: In every start states of A_2, precisely one arrow set contains a *grant* arrow. Furthermore, every action that adds a *grant* arrow to an arrow set also removes a *grant* arrow from an arrow set. The result follows by a simple inductive argument, since all states of A_2 are reachable. □

The second invariant states that every *request* arrow placed on the graph by the arbiter points toward the root of the graph. In other words, the arbiter correctly forwards requests in the direction of the resource.

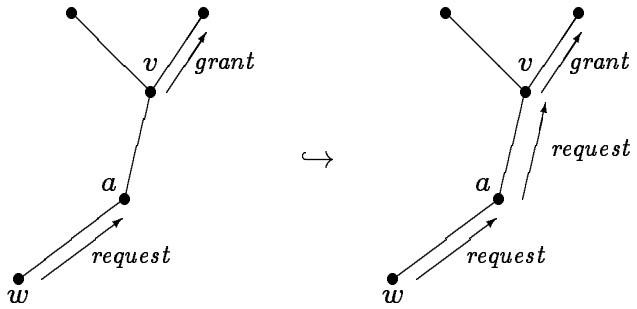
Lemma 36: Let s be a state of A_2, and let a be an arbiter node of G. If $arrows(a, v)$ contains a *request* arrow, then $\langle a, v\rangle$ points toward the root of G.

Proof: No arrow set of any start state contains a *request* arrow, so the start states of A_2 certainly satisfy the hypothesis. Suppose s is a state of A_2 satisfying the hypothesis, and suppose that $s \xrightarrow{\pi} t$ is a step of A_2. We claim that t satisfies the hypothesis as well. Suppose π is of the form $request(x, y)$. Notice that π does not modify the position of the *grant* arrow, and that π adds a *request* arrow to $arrows(a, v)$ only if $\langle a, v\rangle$ points toward the root in s, and hence in t. It follows that t must satisfy the hypothesis. Suppose $\pi = grant(v, a)$. In this case, π removes any *request* arrow from $arrows(a, v)$, and so t must satisfy the hypothesis. Finally, suppose $\pi = grant(x, y) \neq grant(v, a)$. Since π does not add or remove a *request* arrow from $arrows(a, v)$, if the set $arrows(a, v)$ contains a *request* arrow in t, the same is true in s. The fact that π is enabled from s implies that x is the root in s. The hypothesis implies that the edge $\langle a, v\rangle$ must point toward the root x in s. Since π forwards the resource from x to y (and since $y \neq a$) the edge $\langle a, v\rangle$ must point toward the root y in t. Therefore, t must satisfy the hypothesis. The lemma now follows by a simple inductive argument, since all states of A_2 are reachable. □

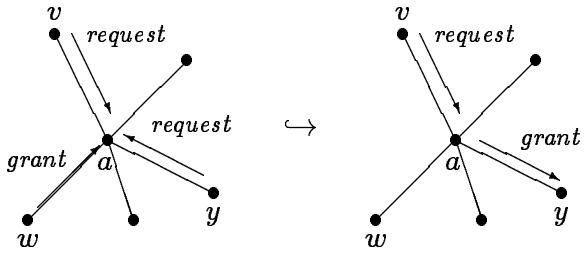
3.2.3 The Execution Module E_2

To ensure that the arbiter satisfies all user requests, it is obviously important that the internal arbiter nodes forward all requests in the direction of the root, and that arbiter nodes holding the resource eventually grant the resource to adjacent requesting nodes. Let a be an arbiter node adjacent to nodes v and w, and let us define the following sets of states and actions.

$$\begin{aligned} FwdReq_2^s(a, v) &= \{s \in states(A_2) : request \in arrows(w, a) \text{ for some } w, \\ &\qquad \langle a, v\rangle \text{ points toward the root, and} \\ &\qquad request \notin arrows(a, v) \text{ in } s\} \\ FwdReq_2^a(a, v) &= \{grant(v, a), request(a, v)\} \\ \\ FwdGr_2^s(a, v, w) &= \{s \in states(A_2) : request \in arrows(v, a) \text{ and} \end{aligned}$$



The correctness condition $FwdReq_2$.



The correctness condition $FwdGr_2$.

Figure 3.4: Arbiter correctness conditions.

$$grant \in arrows(w, a) \text{ in } s\}$$

$$FwdGr_2^a(a, v, w) \;=\; \{grant(a, y) : y \in (w, v]\}$$

The first arbiter correctness condition

$$FwdReq_2 = \bigwedge_{a,v} FwdReq_2^s(a, v) \hookrightarrow FwdReq_2^a(a, v),$$

illustrated at the top of Figure 3.4, states that if an arbiter node a is at the head of a *request* arrow and has not forwarded the request in the direction of the root, then either a becomes the root (possibly because v is a user node, and v has placed a *grant* arrow on $\langle v, a\rangle$), or a eventually forwards the request in the direction of the root. The second arbiter correctness condition

$$FwdGr_2 = \bigwedge_{a,v,w} FwdGr_2^s(a, v, w) \hookrightarrow FwdGr_2^a(a, v, w),$$

illustrated at the bottom of Figure 3.4, states that if an arbiter node a is a root at the head of a *request* arrow, then it eventually forwards the resource to an adjacent requesting

node. The correctness condition

$$C_2 \;=\; FwdReq_2 \wedge FwdGr_2$$

ensures that arbiter nodes always forward requests in the direction of the root; and that arbiter nodes holding the resource always grant it to adjacent requesting nodes. We let E_2 be the execution module of A_2 with the executions of A_2 satisfying the condition C_2.

While Lemma 35 states that at most one user is using the resource at any given time, and while condition C_2 ensures that arbiter nodes holding the resource always grant the resource to requesting nodes, we have not yet shown that the arbiter always satisfies user requests. As before, this requires cooperation on the part of the users. Let u be a user node adjacent to the arbiter node a, and let us define the following sets of states and actions.

$$\begin{aligned} RtnRes_2^s(u) &= \{s \in states(A_2) : grant \in arrows(a,u) \text{ in } s\} \\ RtnRes_2^a(u) &= \{grant(u,a)\} \end{aligned}$$

$$\begin{aligned} GrRes_2^s(u) &= \{s \in states(A_2) : request \in arrows(u,a) \text{ in } s\} \\ GrRes_2^a(u) &= \{grant(a,u)\} \end{aligned}$$

The condition

$$RtnRes_2 = \bigwedge_u RtnRes_2^s(u) \hookrightarrow RtnRes_2^a(u)$$

says user nodes holding the resource always return the resource, and the condition

$$GrRes_2 = \bigwedge_u GrRes_2^s(u) \hookrightarrow GrRes_2^a(u)$$

says the arbiter eventually satisfies requesting users. The condition $RtnRes_2 \supset GrRes_2$ says that if users return the resource, then the arbiter satisfies all requests. We now show that every execution of E_2 satisfies the condition $RtnRes_2 \supset GrRes_2$. First, however, we prove the following result, the inductive statement in the argument that E_2 satisfies the condition $RtnRes_2 \supset GrRes_2$.

Lemma 37: Let s be a state of A_2 having a *request* arrow in $arrows(v,w)$. Let x be an execution fragment of A_2 from s satisfying the condition $C_2 \wedge RtnRes_2$. Then the action $grant(w,v)$ must appear in x.

Proof: If the graph G is viewed as a tree rooted at v, then w can be viewed as the root of a subtree of v. We proceed by induction on the height h of the subtree of v rooted at w.

Suppose $h = 0$. In this case, w must be a leaf of G, and therefore w must be a user node and v an arbiter node. Since v is an arbiter node and $arrows(v, w)$ contains a *request* arrow, Lemma 36 implies the edge $\langle v, w\rangle$ points toward the root. Therefore, $arrows(v, w)$ must contain a *grant* arrow. Since x satisfies $RtnRes_2$, the user w must eventually return the resource to the arbiter, and hence $grant(w, v)$ must appear in x.

Suppose $h > 0$ and the inductive hypothesis holds for $h - 1$. We first show that x can be written as $\alpha x'$ where x' is an execution fragment satisfying $C_2 \wedge RtnRes_2$ in whose initial state $request \in arrows(v, w)$ and w is the root (that is, $grant \in arrows(w', w)$ for some node w'). We consider two cases. First, suppose $\langle v, w\rangle$ does not point toward the root in s. Since $arrows(v, w)$ contains a *request* arrow, Lemma 36 implies that v must be a user node. Since user nodes are leaves, and since $\langle v, w\rangle$ does not point toward the root, the root must be at v; that is, $arrows(w, v)$ must contain a *grant* arrow. Since x satisfies $RtnRes_2$, the user v must eventually return the resource to the arbiter, so $grant(v, w)$ must appear in x. Therefore, $x = \beta\, grant(v, w)x'$ as desired. Now, suppose $\langle v, w\rangle$ *does* point toward the root. If w itself is the root, then setting $x' = x$ we are done, so suppose w is not the root. If for some node w' the set $arrows(w, w')$ contains a *request* arrow, then since the height of the subtree of w rooted at w' must be less than h, the inductive hypothesis for $h-1$ implies that $grant(w', w)$ appears in x. Therefore, $x = \beta\, grant(w', w)x'$ as desired. On the other hand, suppose no arrow set $arrows(w, w')$ contains a *request* arrow. Note that the fact that $h > 0$ implies that w is not a leaf, and hence that w is an arbiter node. Since x satisfies C_2, we see that for some node w' either $grant(w', w)$ or $request(w, w')$ appears in x. If $grant(w', w)$ appears in x, then $x = \beta\, grant(w', w)x'$ as desired. If $request(w, w')$ appears in x, then a *request* arrow is placed in $arrows(w, w')$, and again the inductive hypothesis for $h - 1$ implies that $x = \beta\, grant(w', w)x'$ as above.

We now show that if x' is an execution fragment satisfying $C_2 \wedge RtnRes_2$ in whose initial state $request \in arrows(v, w)$ and $grant \in arrows(w', w)$ for some node w', then $grant(w, v)$ appears in x'. From this it will follow that $grant(w, v)$ appears in x as well. We proceed by induction on d, the distance from w' to v in the ordering of the nodes adjacent to w in G. Suppose $d = 1$. Since $request \in arrows(v, w)$ and $grant \in arrows(w', w)$, condition C_2 implies that $grant(w, y)$ must appear in x' for some $y \in (w', v] = \{v\}$. Thus, $grant(w, v)$ must appear in x'. Suppose $d > 1$ and the inductive hypothesis holds for $d-1$. Suppose the inductive hypothesis does *not* hold for x': Suppose that $grant(w, v)$ does not appear in x', and hence that $request \in arrows(v, w)$ in every state appearing in x'. As in the case of $d = 1$, the action $grant(w, y)$ must appear in x' for some $y \in (w', v]$. If $y = v$ then we are done, so suppose $y \neq v$. If $arrows(w, y)$ contains a *request*, then the inductive hypothesis for $h-1$ implies that $grant(w, y)$ appears in x', and the inductive hypothesis for $d-1$ implies that $grant(w, v)$ must also appear in x'. On the other hand, suppose $arrows(w, y)$ does not contain a *request* arrow. Condition C_2 implies that either $grant(y, w)$ or $request(w, y)$ appears in x'. If $grant(y, w)$ appears in x', then a *grant* arrow is placed in $arrows(y, w)$, and the inductive hypothesis for $d-1$ implies that $grant(w, v)$ appears in x'. If $request(w, y)$ appears in x', then a *request* arrow is placed

in $arrows(w, y)$, and $grant(w, v)$ must appear in x' as we have seen above. □

An immediate corollary of Lemma 37 is the following.

Corollary 38: Every execution of E_2 satisfies the condition $RtnRes_2 \supset GrRes_2$.

3.2.4 The Execution Module E_2'

For the sake of exposition, we have given the actions of A_2 names suitable to its level of abstraction, rather than using names from A_1. It is therefore necessary to rename these actions before showing that E_2 solves E_1. The action mapping f_1 from A_2 to A_1 is defined to map

$$\begin{array}{rcl} request(u, a) & \text{to} & request(u), \\ grant(u, a) & \text{to} & return(u), \\ grant(a, u) & \text{to} & grant(u), \end{array}$$

and all remaining (internal) actions to themselves. We will denote by A_2' the automaton $f_1(A_2)$, and in general we will denote by affixing a prime to its name the entity obtained by renaming its actions according to f_1.

3.2.5 The Satisfaction of E_1 by E_2'

We begin the proof that E_2' satisfies E_1 by exhibiting a possibilities mapping from A_2' to A_1. The mapping h_1 maps the state s of A_2' to the state t of A_1 such that

$$\begin{array}{rcl} u \in requesters \text{ in } t & \text{iff} & request \in arrows(u, a) \text{ in } s \\ holder = u \text{ in } t & \text{iff} & grant \in arrows(a, u) \text{ in } s \\ holder = a \text{ in } t & \text{iff} & grant \notin arrows(a, u) \text{ for every user } u \text{ in } s \end{array}$$

These conditions ensure that a user is a requesting user in t iff it is in s, and that a user is holding the resource in t iff it is in s. Since all states of A_2' are reachable, and since in all reachable states of A_2' there is exactly one root, this mapping takes each state of A_2' to a singleton set of states of A_1.

Lemma 39: The mapping h_1 is a possibilities mapping from A_2' to A_1.

Proof: The automata A_2' and A_1 clearly have the same external action signature. If s is a start state of A_2, then a single arrow set $arrows(v, a)$ contains a *grant* arrow and all other arrow sets are empty. In particular, no arrow set $arrows(u, a)$ contains a *request* arrow, and no arrow set $arrows(a, u)$ contains a *grant* arrow. Therefore, in every state

of $h_1(s)$ the set *requesters* of requesting users is empty, and $holder = a$. Since this is the start state of A_1, we see that if s is a start state of A_2', then a start state of A_1 is contained in $h_1(s)$.

Consider the action $\pi = request(u)$ of A_2', originally the action $request(u, a)$ of A_2. Suppose s and t are reachable states of A_2' and A_1, respectively, such that $t \in h_1(s)$. The action π is an input action of both automata, and hence is enabled from both s and t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. Since π adds a *request* arrow to $arrows(u, a)$ in s', and adds u to *requesters* of requesting users in t', we see that $t' \in h_1(s')$.

Consider the action $\pi = return(u)$ of A_2', originally the action $return(u, a)$ of A_2. Suppose s and t are reachable states of A_2' and A_1, respectively, such that $t \in h_1(s)$. Again, π is an input action of both automata, and hence is enabled from both s and t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. The definition of h_1 implies that $grant \in arrows(a, u)$ in s iff $holder = u$ in t. If both conditions are false, then π has no effect on either s or t, so $t \in h_1(s)$ implies $t' \in h_1(s')$. Suppose both conditions are true. Notice that u is the unique root in s. The action π moves the *grant* arrow from $arrows(a, u)$ to $arrows(u, a)$ in s', and π sets *holder* to a in t'. Thus, $t' \in h_1(s')$.

Consider the action $\pi = grant(u)$ of A_2', originally the action $grant(a, u)$ of A_2. Suppose s and t are reachable states of A_2' and A_1, respectively, such that $t \in h_1(s)$. If π is enabled from s, then $request \in arrows(u, a)$ and $grant \in arrows(w, a)$ for some node w. Since $request \in arrows(u, a)$ in s, the set *requesters* of requesting users contains u in t. Since a is the unique root in s, $holder = a$ in t. Thus, π is enabled from t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. The action π removes the *request* arrow from $arrows(u, a)$ and moves the *grant* arrow to $arrows(a, u)$ in s', and π removes u from the set *requesters* of requesting users and sets *holder* to u in t'. Therefore, $t' \in h_1(s')$.

Finally, the remaining actions $request(a, u)$, $request(a, a')$, and $grant(a, a')$ of A_2' are not actions of A_1. These actions do not affect *request* arrows in the arrow sets $arrows(u, a)$ or *grant* arrows in the arrow sets $arrows(a, u)$. Therefore, suppose s and t are reachable states of A_2' and A_1 such that $t \in h_1(s)$. If $s \xrightarrow{\pi} s'$ is a step of A_2', then $t \in h_1(s')$. It follows that h_1 is indeed a possibilities mapping from A_2' to A_1. □

We can now show that E_2' satisfies E_1.

Lemma 40: E_2' satisfies E_1.

Proof: Let x be an execution of E_2', and let y be an execution of A_1 corresponding to y under h_1. First, we claim that (i) if y satisfies $RtnRes_1^s(u) \hookrightarrow RtnRes_1^a(u)$, then x satisfies $RtnRes_2^s(u) \hookrightarrow RtnRes_2^a(u)'$. Suppose s is a state of $RtnRes_2^s(u)$. Since $grant \in arrows(a, u)$ in s, we see that $holder = u$ in every state of $h_1(s)$, and hence that $h_1(RtnRes_2^s(u)) \subseteq RtnRes_1^s(u)$. Since, in addition, $RtnRes_1^a(u) \subseteq RtnRes_2^a(u)'$, the claim follows by Lemma 32. Second, we claim that (ii) if x satisfies $GrRes_2^s(u) \hookrightarrow GrRes_2^a(u)'$, then y satisfies $GrRes_1^s(u) \hookrightarrow GrRes_1^a(u)$. Suppose $t \in h_1(s)$ is a state of $GrRes_1^s(u)$.

Since $u \in \mathit{requesters}$ in t, we see that $\mathit{request} \in \mathit{arrows}(u, a)$ in s, and hence that $h_1^{-1}(GrRes_1^s(u)) \subseteq GrRes_2^s(u)$. Since, in addition, $GrRes_2^a(u)' \subseteq GrRes_1^a(u)$, the claim follows by Lemma 32. From observations (i) and (ii) it follows that if y satisfies $RtnRes_1$, then x satisfies $RtnRes_2$; and that if x satisfies $GrRes_2$, then y satisfies $GrRes_1$. Since x satisfies $RtnRes_2 \supset GrRes_2$, it follows that y satisfies $RtnRes_1 \supset GrRes_1$, and hence that y is an execution of E_1. Since $\mathit{sched}(x)|A_1 = \mathit{sched}(y)$, and since E_2' and E_1 have the same external action signature, it follows that $\mathit{fbeh}(E_2') \subseteq \mathit{fbeh}(E_1)$, and hence that E_2' will satisfy E_1. □

3.3 The Automaton A_3

In the description of the arbiter given by the previous model, the arbiter nodes are intended to represent processes in a distributed network implementing the arbiter. Previous models have given global descriptions of the arbiter's behavior. In this model we actually distribute the arbiter by modeling each process as a separate automaton. These automata describe the low-level protocol followed by each process in the arbiter's implementation. Notice that while previous models have acknowledged the asynchrony of processor step times, they have essentially ignored the asynchrony of the network's message system by assuming instantaneous message delivery. We now introduce this asynchrony into the model, modeling the message delivery system as an independent automaton. By composing the automata modeling arbiter processes with the automaton modeling the message delivery system, we obtain a global model of the arbiter.

In order to model asynchronous message delivery, it is convenient to add to the graph G an extra arbiter node $b_{a,a'}$ (or $b_{a',a}$) between every pair of adjacent arbiter nodes a and a'. The node $b_{a,a'}$ acts as a message buffer between a and a': The node a sends a message to a' by placing an arrow on the edge $\langle a, b_{a,a'} \rangle$, and the message system delivers the message to a' by placing an arrow on the edge $\langle b_{a,a'}, a' \rangle$. Since they function as message buffers, we will hereafter refer to the nodes $b_{a,a'}$ as *buffer nodes*. We denote by $\mathcal{G}$ the graph obtained from G by the addition of such buffer nodes. Two nodes (processes) are said to be *adjacent* in $\mathcal{G}$ if they are separated by at most a buffer node; that is, if they are user or arbiter nodes adjacent in the graph G. Since the results of the previous section hold for arbitrary connected, acyclic graphs, and since $\mathcal{G}$ is such a graph, these results hold for the graph $\mathcal{G}$. We therefore fix $\mathcal{G}$ as the graph underlying the model A_2. Furthermore, we fix as the set of start states of A_2 those start states in which no buffer node is a root. In such states, the arbiter holds the resource, and no undelivered messages are pending. We note that with the added structure of $\mathcal{G}$, we can prove the following result about buffer nodes during executions of A_2.

Lemma 41: Let a and a' be adjacent arbiter nodes, and let s be a state of A_2. If $\mathit{request} \in \mathit{arrows}(b_{a,a'}, a')$ or $\mathit{grant} \in \mathit{arrows}(a', b_{a,a'})$, then $\mathit{request} \in \mathit{arrows}(a, b_{a,a'})$.

Proof: The sets $arrows(b_{a,a'}, a')$ and $arrows(a', b_{a,a'})$ do not contain *request* or *grant* arrows, respectively, in any start state of A_2, and hence every start state satisfies the hypothesis. Suppose s is a reachable state of A_2 satisfying the hypothesis, and suppose $s \xrightarrow{\pi} t$ is a step of A_2. We claim that t satisfies the hypothesis was well. If $\pi = request(x, y)$, then π places a *request* arrow in $arrows(x, y)$. The only case we need consider is the case of $\langle x, y \rangle = \langle b_{a,a'}, a' \rangle$. In this case, π is enabled only if $\langle b_{a,a'}, a' \rangle$ points toward the root, and there is a *request* in $arrows(v, b_{a,a'})$ for some v. If $v = a'$, then Lemma 36 implies that the edge $\langle a', b_{a,a'} \rangle$ also points toward the root. Since Lemma 35 states that there is only one root, this is clearly impossible. Therefore, we must have $v = a$, and hence that t satisfies the hypothesis. If $\pi = grant(x, y)$, then π places a *grant* arrow in $arrows(x, y)$. The only case we need consider is the case of $\langle x, y \rangle = \langle a', b_{a,a'} \rangle$. In this case, π is enabled only if there is a *request* arrow in $arrows(b_{a,a'}, a')$ in s. By hypothesis, there must be a *request* arrow in $arrows(a, b_{a,a'})$ in s, and hence in t. Therefore, t must satisfy the hypothesis. The lemma follows by a simple inductive argument. □

Note that we do not model any message asynchrony between users and the arbiter: User nodes are to be interpreted as ports to the arbiter through which the users communicate with the arbiter, and not the user processes themselves. If the arbiter is to be used in a larger system, then the responsibility of modeling the message delivery between the arbiter and the rest of the system falls on the model of the larger system's message delivery.

The previous models have given some indication of the behavior required of arbiter processes. In the first place, arbiter processes must always forward a request for the resource in the direction of the resource. Since the network is acyclic, the process is able to determine the direction of the resource by remembering the direction in which it last forwarded the resource. Furthermore, arbiter processes holding the resource must forward the resource to a requesting process. In particular, if arbiter process a receives the resource from process v, then a must grant the resource to the first requesting process after v in a fixed ordering of its neighbors. Therefore, the state of an arbiter process is determined by a set of processes from which it has received a request, the link over which the resource was last sent, whether or not the process is holding the resource, and whether or not a request has been forwarded in the direction of the resource. For each arbiter process a (each arbiter node of the graph G), we construct an automaton A_a modeling the process a.

The behavior required of the message system is very simple. The system must be able to accept messages for delivery, and ensure that every message sent is eventually delivered. The state of the message system is simply a collection of undelivered messages, together with their destinations. We construct an automaton M to model the asynchronous message communication system.

3.3.1 The States of A_a and M

As mentioned above, a state of A_a is determined by a set *requesting* of requesting processes adjacent to a, a variable *lastforward* indicating the adjacent process to which a last forwarded the resource, a binary flag *holding* indicating whether or not a is holding the resource, and a binary flag *requested* indicated whether or not a has requested the resource since last holding the resource. To define the start state of A_a, we designate one of the arbiter processes and the *initial holder* of the resource. The start state of A_a is a state in which the set *requesting* of requesting processes is empty; the variable *lastforward* is set to the process adjacent to a on the path from a to the process currently holding the resource, or to any adjacent process if a is the initial holder; the flag *holding* is set depending on whether a is the initial holder; and the flag *requested* is set to false. Notice that there are several possible initial states for the initial holder since *lastforward* may be set to any of its adjacent processes, but that the initial state of the remaining processes is unique.

As indicated above, the state of M is determined by a set *messages* of messages to deliver (either request or grant messages) together with the identity of the sender and receiver of the message. More formally, *messages* is a set of triples of the form $(v, w, \mathit{request})$ or (v, w, grant) denoting messages to be delivered from v to w. The initial state of M is the state in which *messages* is empty, the state in which no messages are undelivered.

3.3.2 The Actions of A_a and M

The actions of A_a are given in Figure 3.5. The input actions are those actions of the form $\mathit{receiverequest}(v, a)$ and $\mathit{receivegrant}(v, a)$, and the output actions are of the form $\mathit{sendrequest}(a, v)$ and $\mathit{sendgrant}(a, v)$, where v is a node (process) adjacent to a in the graph $\mathcal{G}$. These actions behave just as described above. There are no internal actions of A_a.

The actions of M are given in Figure 3.6. The input actions are those actions of the form $\mathit{sendrequest}(a, a')$ and $\mathit{sendgrant}(a, a')$, and the output actions are of the form $\mathit{receiverequest}(a, a')$ and $\mathit{receivegrant}(a, a')$, where a and a' are adjacent arbiter nodes of $\mathcal{G}$. These actions accept messages to be delivered by placing them in the message buffer *messages*, and deliver them by removing them from the buffer. There are no internal actions of M.

3.3.3 The Automaton A_3

The composition of the automata A_a modeling the arbiter processes together with the automaton M modeling the message system yields a global model of the arbiter. However,

Input Actions:

receiverequest(v, a)

effects:

$requesting \leftarrow requesting \cup \{v\}$

receivegrant(v, a)

effects:

if $holding = false$ and $lastforward = v$ then

$holding \leftarrow true$

$requested \leftarrow false$

Output Actions:

sendrequest(a, v)

preconditions:

$requesting \neq \emptyset$

$requested = false$

$holding = false$

$lastforward = v$

effects:

$requested \leftarrow true$

sendgrant(a, v)

preconditions:

$v \in requesting$

$holding = true$

$lastforward = w$

$y \notin requesting$ for all $y \in (w, v)$

effects:

$requesting \leftarrow requesting - \{v\}$

$lastforward = v$

$holding \leftarrow false$

Figure 3.5: The Actions of A_a.

Input Actions:

$sendrequest(a, a')$

effects:

$messages \leftarrow messages \cup \{(a, a', request)\}$

$sendgrant(a, a')$

effects:

$messages \leftarrow messages \cup \{(a, a', grant)\}$

Output Actions:

$receiverequest(a, a')$

preconditions:

$(a, a', request) \in messages$

effects:

$messages \leftarrow messages - \{(a, a', request)\}$

$receivegrant(a, a')$

preconditions:

$(a, a', grant) \in messages$

effects:

$messages \leftarrow messages - \{(a, a', grant)\}$

Figure 3.6: The actions of M.

we must hide actions that are inherently internal to the arbiter. Therefore, we define the automaton A_3 to be the composition of the automata A_a together with the automaton M, after hiding all output actions of the composition except those of the form $sendgrant(a,u)$ (where a and u are adjacent arbiter and user nodes, respectively).

3.3.4 The Execution Module E_3

As mentioned in the introduction to this model, an arbiter process a is required to forward all requests, and to grant the resource to a requesting process if the arbiter process holds the resource. Let v and w be two nodes adjacent to the arbiter node a, and let us define the following sets of states and actions.

$$\begin{aligned}
FwdReq_a^s(v) &= \{s \in states(A_a) : requesting \neq \emptyset, \\
&\qquad\qquad requested = false, \\
&\qquad\qquad holding = false, \text{ and} \\
&\qquad\qquad lastforward = v \text{ in } s\} \\
FwdReq_a^a(v) &= \{receivegrant(v,a), sendrequest(a,v)\} \\
\\
FwdGr_a^s(v,w) &= \{s \in states(A_a) : v \in requesting \\
&\qquad\qquad holding = true, \text{ and} \\
&\qquad\qquad lastforward = w \text{ in } s\} \\
FwdGr_a^a(v,w) &= \{sendgrant(a,y) : y \in (w,v]\}
\end{aligned}$$

The condition

$$FwdReq_a = \bigwedge_v FwdReq_a^s(v) \hookrightarrow FwdReq_a^a(v)$$

says that the arbiter process a having received a request and not holding the resource will either forward a request for the resource or receive the resource (without having requested it, perhaps from a user). The condition

$$FwdGr_a = \bigwedge_v FwdGr_a^s(v) \hookrightarrow FwdGr_a^a(v)$$

says that the arbiter process a holding the resource and having received a request will eventually forward the resource to a requesting process. The condition

$$C_a = FwdReq_a \wedge FwdGr_a$$

is the desired correctness condition for the arbiter process a. We note the following.

Lemma 42: Every fair execution of A_a satisfies C_a.

Proof: Let s be a state of $FwdReq_a^s(v)$ and let x be an execution fragment of A_a from s. If neither $receivegrant(v,a)$ nor $sendrequest(a,v)$ appear in x, then $sendrequest(a,v)$ is enabled from every state appearing in x. Therefore, every fair execution of A_a satisfies $FwdReq_a$. Similarly, let s be a state of $FwdGr_a^s(v,w)$ and let x be an execution fragment of A_a from s. If no action of $FwdGr_a^a(v,w)$ appears in x, then again an action from this set is enabled from every state appearing in x. Therefore, every fair execution of A_a satisfies $FwdGr_a$. It follows that every fair execution of A_a satisfies C_a. □

We let the execution module $E_a = Fair(A_a)$. Recall that an object O solves (the problem specified by) an object O' only if it is implementable. Since E_a is part of our solution to the arbiter's problem specification, it is necessary to show that E_a (as well as every other execution module defined at this low level of abstraction) is implementable.

Lemma 43: E_a is implementable.

We must also require that the message system deliver all messages sent. Let a and a' be two adjacent arbiter processes, and let us define the following sets of states and actions.

$$\begin{aligned} DelReq_M^s(a,a') &= \{s \in states(M) : (a,a',request) \in messages \text{ in } s\} \\ DelReq_M^a(a,a') &= \{receiverequest(a,a')\} \end{aligned}$$

$$\begin{aligned} DelGr_M^s(a,a') &= \{s \in states(M) : (a,a',grant) \in messages \text{ in } s\} \\ DelGr_M^a(a,a') &= \{receivegrant(a,a')\} \end{aligned}$$

If we let

$$DelReq_M = \bigwedge_{a,a'} DelReq_M^a(a,a') \hookrightarrow DelReq_M^s(a,a')$$

and

$$DelGr_M = \bigwedge_{a,a'} DelGr_M^a(a,a') \hookrightarrow DelGr_M^s(a,a'),$$

then the condition

$$C_M = DelReq_M \wedge DelGr_M$$

says that messages sent are always delivered. We denote by E_M the execution module of M with the executions satisfying C_M.

Lemma 44: E_M is implementable.

Proof: It is easy to construct an automaton M' with the action signature of E_M whose fair executions are executions of E_M: The automaton M' keeps messages to be delivered in a FIFO buffer, and delivers them in the order in which they are received for delivery. □

Finally, we define E_3 to be the composition of the execution modules E_a and E_M after hiding the internal actions of A_3. As a result of Lemma 26, we have the following.

Lemma 45: E_3 is implementable.

3.3.5 The Execution Module E_3'

As with the execution module E_2, it is necessary to rename the actions of E_3 to be consistent with the names of E_2. As mentioned when we defined the buffer nodes $b_{a,a'}$, the arbiter node a sends a message to the arbiter node a' by placing an arrow on the edge $\langle a, b_{a,a'} \rangle$ between a and the buffer node $b_{a,a'}$, and the message system delivers the message by placing an arrow on the edge $\langle b_{a,a'}, a' \rangle$ between the buffer node and a'. An arbiter node and user node communicate by placing an arrow on the edge between them. Therefore, if a is an arbiter node and a' and u are arbiter and user nodes, respectively, adjacent to a in $\mathcal{G}$, we define the action mapping f_2 to map

$$
\begin{array}{rcl}
\mathit{receiverequest}(u,a) & \text{to} & \mathit{request}(u,a) \\
\mathit{receivegrant}(u,a) & \text{to} & \mathit{grant}(u,a) \\
\mathit{sendrequest}(a,u) & \text{to} & \mathit{request}(a,u) \\
\mathit{sendgrant}(a,u) & \text{to} & \mathit{grant}(a,u)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{receiverequest}(a',a) & \text{to} & \mathit{request}(b_{a',a},a) \\
\mathit{receivegrant}(a',a) & \text{to} & \mathit{grant}(b_{a',a},a) \\
\mathit{sendrequest}(a,a') & \text{to} & \mathit{request}(a,b_{a,a'}) \\
\mathit{sendgrant}(a,a') & \text{to} & \mathit{grant}(a,b_{a,a'})
\end{array}
$$

We will denote by A_3' the automaton $f_2(A_3)$, and in general we will denote by affixing a prime to its name the entity obtained by renaming its actions according to f_2.

3.3.6 The Solution of E_2 by E_3'

We begin the proof that E_3' satisfies E_2 by exhibiting a possibilities mapping from A_3' to A_2. In order to define this mapping, it will be necessary to refer to state variables from each of the components of A_3'. While the name of the state variable *messages* of M' is unique to M', the remaining components share variable names. In order to avoid ambiguity, we will indicate the component to which a state variable belongs by subscripting the variable with an appropriate identifier. For example, the set *requesting* of requesting processes in A_a' will be denoted by $\mathit{requesting}_a$. The mapping h_2 maps the state s of A_3' to the set of states t of A_2 satisfying the following conditions:

$U1$	$request \in arrows(u, a)$	iff	$u \in requesting_a$
$U2$	$grant \in arrows(u, a)$	iff	$holding_a = true$ and $lastforward_a = u$
$U3$	$request \in arrows(a, u)$	iff	$requested_a = true$ and $lastforward_a = u$
$U4$	$grant \in arrows(a, u)$	iff	$holding_a = false$ and $lastforward_a = u$
$A1$	$request \in arrows(b_{a',a}, a)$	iff	$a' \in requesting_a$
$A2$	$grant \in arrows(b_{a',a}, a)$	iff	$holding_a = true$ and $lastforward_a = a'$
$A3$	$request \in arrows(a, b_{a,a'})$	iff	$requested_a = true$ and $lastforward_a = a'$
$A4$	$grant \in arrows(a, b_{a,a'})$	iff	$(a, a', grant) \in messages$
$I1$	$request \in arrows(a, b_{a,a'})$, $request \notin arrows(b_{a,a'}, a')$, and $grant \notin arrows(a', b_{a,a'})$	iff	$(a, a', request) \in messages$
$I2$	$\langle a, b_{a,a'} \rangle$ points toward the root	iff	$holding_a = false$ and $lastforward_a = a'$

The conditions $U1 - U2$ and $A1 - A4$ are straightforward. They say that the arbiter process a has received a request from a process v in t iff v is in a's set *requesting* of requesting processes in s, and that a has received the resource from v in t iff a holds the resource in s and last sent (and hence received) the resource from v. Similarly, a has forwarded a request for the resource in t iff a has sent a request in the direction it last forwarded the resource in s. $A4$ says that the resource is in transit between a and a' in t iff there is a *grant* message from a to a' in the message buffer *messages* in s. $U4$ says that the user u has the resource in t if in s the node a last forwarded the resource to u and has not received the resource since. Conditions $I1$ and $I2$ are invariants that must be preserved by the mapping. $I1$ says that a state with a request in transit must map only to states satisfying Lemma 41. $I2$ says that the value of *lastforward* correctly records the direction of the resource in the network. We now have the following.

Lemma 46: The mapping h_2 is a possibilities mapping from A'_3 to A_2.

Proof: The action mapping f_2 has renamed the actions of A_3 so that A'_3 and A_2 have the same external action signature. Let s be a start state of A'_3. For every arbiter process a in s, the set $requesting_a$ of requesting processes is empty, and $requested_a$ is set to *false*. It follows by $U1$, $U3$, $A1$, and $A3$ that no arrow set of any state in $h_2(s)$ contains a *request* arrow. Furthermore, the initial holder a in s has set its flag $holding_a$ to *true*; all other processes a' have set $holding_{a'}$ to *false*, and $lastforward_{a'}$ to the node in the direction of the resource; and no *grant* message is pending in the message buffer *messages*. It follows by $U2$, $U4$, $A2$, and $A4$ that there is precisely one root in every state of $h_2(s)$. Therefore, $h_2(s)$ contains a start state of A_2 as desired.

Consider the action $\pi = request(u, a)$ of A'_3, originally the action $receiverequest(u, a)$ of A_3. Suppose s and t are reachable states of A'_3 and A_2, respectively, such that $t \in h_2(s)$. The action π is an input action of both automata, and hence is enabled from both s

and t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. To show that $t' \in h_2(s')$, we must show that $U1$ holds. However, π adds u to the set $requesting_a$ of requesting processes is s', and adds a *request* arrow to the set $arrows(u,a)$ in t', and hence $U1$ holds. Therefore, $t' \in h_2(s')$.

Consider the action $\pi = grant(u,a)$ of A_3', originally the action $receivegrant(u,a)$ of A_3. Suppose s and t are reachable states of A_3' and A_2, respectively, such that $t \in h_2(s)$. Since π is an input action in both automata, π is enabled from both s and t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. We see by $U4$ that there is a *grant* arrow in the set $arrows(a,u)$ of t iff $holding_a = false$ and $lastforward_a = u$ in s. If both conditions are false, then π has no effect on either state, and hence $t \in h_2(s)$ implies $t' \in h_2(s')$. On the other hand, suppose both conditions are true. To show $t' \in h_2(s')$, we must show that $U2$, $U3$, and $U4$ hold. Notice that $lastforward_a = u$ in s'. First, π sets $holding_a$ to *true* in s', and adds a *grant* arrow to $arrows(u,a)$ in t', so $U2$ holds. Second, π sets $requested_a$ to *false* in s', and removes any *request* arrow from the set $arrows(a,u)$ in t', so $U3$ holds. Finally, since π sets $holding_a$ to *true* in s', the fact that π moves the *grant* arrow from $arrows(a,u)$ to $arrows(u,a)$ implies that $U4$ holds. Therefore, $t' \in h_2(s')$.

Consider the action $\pi = request(a,u)$ of A_3', originally the action $sendrequest(a,u)$ of A_3. Suppose s and t are reachable states of A_3' and A_2, respectively, such that $t \in h_2(s)$. If π is enabled from s, then the set $requesting_a$ of requesting processes is nonempty in s, so $U1$ and $A1$ implies that some set $arrows(w,a)$ contains a *request* arrow in t. Furthermore, since $holding_a = false$ and $lastforward_a = u$ in s, we have by $U4$ that $arrows(a,u)$ contains a *grant* arrow in t, and hence that the edge $\langle a,u \rangle$ points toward the root in t. Finally, since $requested_a = false$ in s, by $U3$ we have that $arrows(a,u)$ does not contain a *request* arrow. Therefore, π is enabled from t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. To see that $t' \in h_s(s')$, we must show that $U3$ holds. Notice that π sets $requested_a$ to *true* in s', and that $lastforward_a = u$ in s'. Since π adds a *request* arrow to $arrows(a,u)$ in t', we see that $U3$ holds. Therefore, $t' \in h_2(s')$.

Consider the action $\pi = grant(a,u)$ of A_3', originally the action $sendgrant(a,u)$ of A_3. Suppose s and t are reachable states of A_3' and A_2, respectively, such that $t \in h_2(s)$. If π is enabled from s, then u is contained in the set $requesting_a$ of requesting processes in s, and $U1$ implies that $arrows(u,a)$ contains a *request* arrow. Furthermore, $holding_a = true$ and $lastforward_a = w$ in s, so $U2$ and $A2$ imply that $arrows(b_{w,a},a)$ (or $arrows(w,a)$ if w is a user node) contains a *grant* arrow in t. In addition, since $y \notin requesting_a$ for all $y \in (w,u)$ in s, $U1$ and $A1$ imply that in t no set $arrows(b_{y,a},a)$ (or $arrows(y,a)$ if y is a user node) contains a *request* arrow for any $y \in (w,u)$. Therefore, π is enabled from t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. To show that $t' \in h_2(s')$, we must show that $U1$, $U2$ and $A2$, $U3$ and $A3$, and $U4$ hold. First, the action π removes u from $requesting_a$ in s', and removes a *request* arrow from $arrows(v,a)$ in t', so $U1$ holds. Second, since $holding_a$ is set to *false* in s', and since a is not a root in t', $U2$ and $A2$ hold. Third, since $holding_a = true$ in s, we see that $requested_a = false$ in s and hence in s', so $U3$ and $A3$ hold. Finally, since π sets $holding_a$ to *false* and $lastforward_a$ to u in s', and since π adds a *grant* arrow to $arrows(a,u)$ in t', we see that $U4$ holds. Therefore, $t' \in h_2(s')$.

Consider the action $\pi = request(b_{a',a}, a)$ of A_3', originally $receiverequest(a', a)$ of A_3. Suppose s and t are reachable states of A_3' and A_2, respectively, such that $t \in h_2(s)$. If π is enabled from s, the set *messages* of undelivered messages in s must contain a *request* message from a' to a. It follows by $I1$ that in t the set $arrows(a', b_{a',a})$ contains a *request* arrow, the set $arrows(b_{a',a}, a)$ does not contain a *request* arrow, and the set $arrows(a, b_{a,a'})$ does not contain a *grant* arrow. Since $arrows(a', b_{a',a})$ contains a *request* arrow, Lemma 36 implies that $\langle a', b_{a',a} \rangle$ points toward a root. This together with the fact that $arrows(a, b_{a',a})$ does not contain a *grant* arrow implies that $\langle b_{a',a}, a \rangle$ points toward the root as well. Therefore, the action π is enabled from t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. In order to see that $t' \in h_2(s')$, we must show that $A1$ and $I1$ hold. First, π adds a' to the set $requesting_a$ of requesting processes in s', and π adds a *request* arrow to $arrows(b_{a',a}, a)$ in t', so $A1$ holds. Second, π removes a *request* message from a' to a from the set *messages* of undelivered messages in s', and π adds a *request* arrow to $arrows(b_{a',a}, a)$, so $I1$ holds. Therefore, $t' \in h_2(s')$.

Consider the action $\pi = grant(b_{a',a}, a)$ of A_3', originally the action $receivegrant(a', a)$ of A_3. Suppose s and t are reachable states of A_3' and A_2, respectively, such that $t \in h_2(s)$. If π is enabled from s, the set *messages* of undelivered messages in s must contain a *grant* message from a' to a. By $A4$ we see that the set $arrows(a', b_{a',a})$ contains a *grant* arrow in t. Lemma 41 implies that the set $arrows(a, b_{a,a'})$ must contain a *request* arrow. Since the degree of the buffer node $b_{a,a'}$ is 2, we see that π is enabled from t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. Since the set $arrows(a, b_{a,a'})$ contains a *request* arrow in t, Lemma 36 implies that the edge $\langle a, b_{a,a'} \rangle$ points toward the root. By $I2$ we see that $holding_a = false$ and $lastforward_a = a'$ in s. Therefore, to see that $t' \in h_2(s')$, we must show that $A2$, $A3$, $A4$, and $I2$ hold. First, π sets $holding_a$ to *true* in s'. Notice that $lastforward_a = a'$ in s, and therefore in s' as well. Since π adds a *grant* arrow to $arrows(b_{a',a}, a)$ in t', we see that $A2$ holds. Second, π sets $requested_a$ to *false* in s', and π removes a *request* arrow from $arrows(a, b_{a,a'})$ in t', so $A3$ holds. Third, π removes a *grant* message from a' to a from the set *messages* of undelivered messages in s', and π removes a *grant* arrow from $arrows(a', b_{a',a})$ in t', so $A4$ holds. Finally, since $holding_a$ is set to *true* in s', it is easy to see that $I2$ holds. Therefore, $t' \in h_2(s')$.

Consider the action $\pi = request(a, b_{a,a'})$ of A_3', originally the action $sendrequest(a, a')$ of A_3. Suppose s and t are reachable states of A_3' and A_2, respectively, such that $t \in h_2(s)$. If π is enabled from s, then the set $requesting_a$ of requesting processes is nonempty in s, and hence by $U1$ and $A1$ some set $arrows(w, a)$ of t contains a *request* arrow. Furthermore, since $holding_a = false$ and $lastforward_a = a'$ in s, by $I2$ we see that the edge $\langle a, b_{a,a'} \rangle$ points toward the root in t. Finally, since $requesting_a = false$ in s, by $A3$ we see that there is no *request* arrow in $arrows(a, b_{a,a'})$ in t. Therefore, π is enabled from t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. To see that $t' \in h_2(s')$, we must show that $A3$ and $I1$ hold. Notice that π sets $requested_a$ to *true* in s', and places a *request* arrow in $arrows(a, b_{a,a'})$ in t'. Since $lastforward_a = a'$ in s and hence in s', we see that $A3$ holds. Notice that $requested_a = false$ in s. Since $lastforward_a = a'$ in s, $A3$ implies that

$arrows(a, b_{a,a'})$ does not contain a *request* arrow in t. Lemma 41 implies that there is no *request* arrow in $arrows(b_{a,a'}, a')$ and no *grant* arrow in $arrows(a', b_{a,a'})$ in t, and hence the same is true in t'. Since π adds a *request* arrow to $arrows(a, b_{a,a'})$ in t', and adds a *request* message from a to a' to the set *messages* of undelivered messages in s', we see that $I1$ holds. Therefore, $t' \in h_2(s')$.

Finally, consider the action $\pi = grant(a, b_{a,a'})$ of A'_3, originally $sendgrant(a, a')$ of A_3. Suppose s and t are reachable states of A'_3 and A_2, respectively, such that $t \in h_2(s)$. If π is enabled from s, then since $a' \in requesting_a$ in s, we see by $A1$ that $arrows(b_{a',a}, a)$ contains a *request* arrow in t. Since $holding_a = true$ and $lastforward_a = w$ in s, we see by $U2$ and $A2$ that a *grant* arrow must be contained in $arrows(b_{w,a}, a)$ (or $arrows(w, a)$ if w is a user node) in t. Furthermore, since $y \notin requesting_a$ for all $y \in (w, a')$ in s, we see by $U3$ and $A3$ that no *request* arrow is contained in $arrows(b_{y,a}, a)$ (or $arrows(y, a)$ if y is a user node) in t. Therefore, π is enabled from t. Suppose $s \xrightarrow{\pi} s'$ and $t \xrightarrow{\pi} t'$. To see that $t' \in h_2(s')$, we must show that $A1$, $A2$ and $I2$, $A4$, $I1$, and $I2$ hold. All except $I1$ are straightforward, so we show $I1$. Notice that $arrows(b_{a',a}, a)$ contains a *request* arrow in t. By $I1$, there is no undelivered *request* message from a' to a in the set *messages* of s, and hence in s'. However, π puts a *grant* arrow in $arrows(a, b_{a,a'})$, so $I1$ holds. Therefore, $t' \in h_2(s')$. □

Having exhibited a possibilities mapping h_2 from A'_3 to A_2, we now use this mapping together with Lemma 33 to show that E'_3 satisfies E_2. Before using Lemma 33, however, we must translate the local correctness conditions C'_a and C_M for E'_a and E'_M, respectively, into a global correctness condition for E'_3. We use Lemma 34 to recharacterize E'_3 in this way. Let a and a' be adjacent arbiter nodes, and let v be an arbitrary (user or arbiter) node adjacent to a in $\mathcal{G}$. Let

$$FwdReq^s_a(v)' = \{a \in states(A'_3) \ : \ a | A'_a \in FwdReq^s_a(v)\}$$

$$FwdGr^s_a(v)' = \{a \in states(A'_3) \ : \ a | A'_a \in FwdGr^s_a(v)\}$$

$$DelReq^s_M(a, a')' = \{a \in states(A'_3) \ : \ a | M' \in DelReq^s_M(a, a')\}$$

$$DelGr^s_M(a, a')' = \{a \in states(A'_3) \ : \ a | M' \in DelGr^s_M(a, a')\} .$$

Furthermore, let

$$FwdReq'_a = \bigwedge_v FwdReq^s_a(v)' \hookrightarrow FwdReq^a_a(v)'$$

$$FwdGr'_a = \bigwedge_v FwdGr^s_a(v)' \hookrightarrow FwdGr^a_a(v)'.$$

$$DelReq'_M = \bigwedge_{a,a'} DelReq^s_M(a, a')' \hookrightarrow DelReq^a_M(a, a')'$$

$$DelGr'_M = \bigwedge_{a,a'} DelGr^s_M(a, a')' \hookrightarrow DelGr^a_M(a, a')'.$$

Finally, let

$$C'_a = FwdReq'_a \wedge FwdGr'_a$$
$$C'_M = DelReq'_M \wedge DelGr'_M.$$

If

$$C'_3 = \bigwedge_a C'_a \ \wedge \ C'_M$$

then the following is an immediate result of Lemma 34.

Lemma 47: E'_3 is the execution module of A'_3 with the executions of A'_3 satisfying C'_3.

Having made this transformation from local to global correctness conditions, we now use Lemma 33 to show that E'_3 satisfies E_2.

Lemma 48: E'_3 satisfies E_2.

Proof: Let a and a' be adjacent arbiter nodes, and let v and w be arbitrary nodes adjacent to a. If v is an arbiter node, then let v' be the buffer node $b_{a,v}$ between a and v; and let v' be the node v itself if v is a user node. The node v' is simply the node of $\mathcal{G}$ adjacent to a such that the edge $\langle a, v' \rangle$ points toward v. Let w' be the analogous node with respect to w. We will show that

1. $h_2^{-1}(FwdReq_2^s(a, v')) \subseteq FwdReq_a^s(v)'$
2. $h_2^{-1}(FwdReq_2^s(b_{a,a'}, a')) \subseteq DelReq_M^s(a, a')'$
3. $h_2^{-1}(FwdGr_2^s(a, v', w')) \subseteq FwdGr_a^s(v, w)'$, and
4. $h_2^{-1}(FwdGr_2^s(b_{a,a'}, a, a')) \subseteq DelGr_M^s(a', a)'$

Since it is easy to see from the definition of f_2 and the following sets that

1. $FwdGr_a^a(v)' \subseteq FwdReq_2^a(a, v')$,
2. $DelReq_M^a(a'a)' \subseteq FwdReq_2^a(b_{a',a}, a)$,
3. $FwdGr_a^a(v, w)' \subseteq FwdGr_2^a(a, v', w')$, and
4. $DelGr_M^a(a', a)' \subseteq FwdGr_2^a(b_{a',a}, a)$,

it will follow by Lemma 33 that E_3' satisfies E_2.

First, suppose $t \in h_2(s)$ is a state of $FwdReq_2^s(a, v')$, and let us show that s is a state of $FwdReq_a^s(v)'$. Since some set $arrows(w, a)$ of t contains a *request*, we see by $U1$ and $A1$ that the set $requesting_a$ of requesting processes is nonempty. Since $\langle a, v' \rangle$ points toward the root in t, we see by $U4$ and $I2$ that $holding_a = false$ and $lastforward_a = v$ in s. Since the set $arrows(a, v')$ does not contain a *request* arrow in t, the fact that $lastforward_a = v$ together with $U3$ and $A3$ imply that $requested_a = false$. Therefore, $s \in FwdReq_a^s(v)'$.

Second, suppose $t \in h_2(s)$ is a state of $FwdReq_2^s(b_{a,a'}, a')$, and let us show that s is a state of $DelReq_M^s(a, a')'$. Since in t there is a *request* arrow in $arrows(w, b_{a,a'})$ for some w, the edge $\langle w, b_{a,a'} \rangle$ must point toward the root . Since $\langle b_{a,a'}, a' \rangle$ also points toward the root in t, and since this root is unique, this *request* arrow must be in $arrows(a, b_{a,a'})$. Furthermore, since $\langle b_{a,a'}, a' \rangle$ points toward the root, we see that there can be no *grant* arrow in $arrows(a', b_{a,a'})$ and no *request* arrow in $arrows(b_{a,a'}, a')$. It follows by $I1$ that there is a *request* message from a to a' in the set *messages* of undelivered messages in s. Therefore, $s \in DelReq_M^s(a, a')'$.

Third, suppose $t \in h_2(s)$ is a state of $FwdGr_2^s(a, v', w')$, and let us show that s is a state of $FwdGr_a^s(v, w)'$. Since there is a *request* arrow in $arrows(v', a)$ in t, $U1$ and $A1$ imply that v is contained in the set $requesting_a$ of requesting processes. Since there is a *grant* arrow in $arrows(w', a)$ in t, $U2$ and $A2$ imply that $holding_a = true$ and $lastforward_a = w$ in s. Therefore, $s \in FwdGr_a^s(v, w)'$.

Finally, suppose $t \in h_2(s)$ is a state of $FwdGr_2^s(b_{a,a'}, a, a')$, and let us show that s is a state of $DelGr_M^s(a', a)$. Since there is a *grant* arrow in $arrows(a', b_{a,a'})$ in t, $A4$ implies that there is a *grant* message from a' to a in the set *messages* of undelivered messages in s. Therefore, $s \in DelGr_M^s(a', a)'$. □

Finally, combining the work of the last few section, we have the following result. Let E_3'' be the execution module obtained by renaming the actions of E_3 according to the action mapping $f_1 f_2$.

Theorem 49: E_3'' solves E_1.

Proof: Since E_3' satisfies E_2, it follows by Lemma 27 that E_3'' satisfies E_2'. Since E_2' satisfies E_1, it follows by Lemma 26 that E_3'' satisfies E_1. Since E_3' is implementable, Lemma 27 implies that E_3'' is implementable. Therefore, E_3'' solves E_1. □

With this we have proven the correctness of a fully-detailed protocol for resource allocation in an asynchronous network.

3.4 Time Complexity

The primary concern motivating Schönhage's arbiter is its time performance. For example, Lynch and Fischer consider two simple resource arbiters in [LF81], allocating a

resource among n users. One arbiter is a process that simple polls each user in round-robin order, granting the resource to each requesting user in turn. Given that each user uses the resource for a bounded amount of time, the response time for this arbiter (the maximum time a user must wait for the resource) is $O(n)$ regardless of the number of users requesting the resource. A second arbiter is a binary tree (a tournament tree) with the users at the leaves of the tree. Each internal node of the tree repeatedly polls its children until one of its children requests the resource, at which point it stops and passes the name of the child up to the internal node's parent. The root of the tree actually determines which user is granted the resource. When only one user is requesting the resource at a time, this arbiter's response time is only $O(\log n)$. In the worst case, however, (when every user is requesting the resource) this arbiter's response time is $O(n \log n)$. Schönhage's algorithm, in contrast, combines favorable aspects of both these arbiters. In particular, (in the case that the graph G is a binary tree) the arbiter's response time is $O(\log n)$ if only one user requests the resource at a time, and $O(n)$ in the worst case. In this section we perform the complexity analysis needed to make these claims precise.

For convenience, we perform our complexity analysis at the middle level of abstraction, with the automaton A_2. We have not yet introduced the notion of time into our model. While we have not yet decided on how time should be incorporated into our model, one alternative is to assign times to states (or equivalently to actions) denoting the time at which an automaton transition causes the automaton to enter this state. Let us refer to such an execution as a *timed* execution. In order to perform any time analysis, it is necessary to place bounds on the time between automaton transitions. Recall that all liveness conditions required of the automaton A_2 in the construction of E_2 are of the form $S \hookrightarrow T$, meaning that if A_2 enters a state of S, then eventually an action of T is performed. Let us denote by $S \overset{b}{\hookrightarrow} T$ the condition that if A_2 enters a state s of S, the within time b an action π of T will be performed. That is, following state s in a timed execution satisfying $S \overset{b}{\hookrightarrow} T$ there is a π-step to a state s' such that the difference in times assigned to s and s' is at most b. Let

$$BndedFwdReq_2 = \bigwedge_{a,v} FwdReq_2^s(a,v) \overset{b}{\hookrightarrow} FwdReq_2^a(a,v)$$

$$BndedFwdGr_2 = \bigwedge_{a,v,w} FwdGr_2^s(a,v,w) \overset{b}{\hookrightarrow} FwdGr_2^a(a,v,w)$$

$$BndedRtnRes_2 = \bigwedge_{u} RtnRes_2^s(u) \overset{b}{\hookrightarrow} RtnRes_2^a(u)$$

Let us say that a timed execution of A_2 is *b-bounded* if it satisfies the conditions $BndedFwdReq_2$, $BndedFwdGr_2$, and $BndedRtnRes_2$. We define the *response time* in a b-bounded execution x of A_2 to be a time r such that for all states s with $request \in arrows(u,a)$ (where u is a user node) appearing in x, the difference in times assigned to s and the first state with $grant \in arrows(a,u)$ appearing after s in x is less than r.

Suppose the graph G has diameter d. It is easy to see that the response time for b-bounded executions of A_2 is $2bd$ when only one user request the resource at a time: The request must travel the diameter of the graph to the root, and the root must be moved the diameter of the graph to the user. Thus, we have the following.

Theorem 50: If the diameter of the graph G is d, then the response time in b-bounded executions of A_2 in which at most one user requests the resource at a time is $2bd$.

Conversely, suppose the graph G has e edges. We now show that the worst-case response time (when the arbiter is heavily loaded) is $3be - b$. We begin with the following preliminary lemma, the inductive statement in the proof that the arbiter's response time is $3be - b$. Given an edge $\langle v, w \rangle$, we define $e(v, w)$ to be the number of edges in the subtree of v rooted at w.

Lemma 51: Let s be a state of A_2 in which $request \in arrows(v, w)$ and the edge $\langle v, w \rangle$ points toward the root. In any b-bounded execution fragment of A_2 from s, $grant \in arrows(w, v)$ within time $3be(v, w) + b$.

Proof: We proceed by induction on $e = e(v, w)$. Suppose $e = 0$. In this case, w must be a leaf, and hence a user node. Since the edge $\langle v, w \rangle$ points toward the root, $grant \in arrows(v, w)$. Since w is a user node, condition $BndedRtnRes_2$ implies that $grant \in arrows(w, v)$ within time $b = 3be + b$.

Suppose $e > 0$ and the inductive hypothesis holds for numbers of edges less that e. By assumption, the edge $\langle v, w \rangle$ points toward the root. If w itself is the root, since $request \in arrows(v, w)$, condition $BndedFwdGr_2$ implies that within time b we have $grant \in arrows(w, x)$ for some node x. Notice that if $x = v$, then we are done, so let us assume that $x \neq v$. Then in either case, regardless of whether w itself is the root, the edge $\langle w, x \rangle$ points toward the root within time b for some node x other than v. Let $x = x_i, \ldots, x_1, v$ be the nodes between x and v in the ordering of nodes adjacent to w. Let $e_j = e(w, x_j)$, and notice that $e \geq \sum_{j=1}^{i}(e_j + 1)$. We proceed by induction on i to show that if $request \in arrows(v, w)$ and $\langle w, x_i \rangle$ points toward the root, the $grant \in arrows(w, v)$ within time $\sum_{j=1}^{i} 3b(e_j + 1)$. It will follow that $grant \in arrows(w, v)$ within time $b + \sum_{j=1}^{i} 3b(e_j + 1) \leq 3be + b$ of the time $request \in arrows(v, w)$. The case of $i = 0$ is vacuously true. Suppose $i > 0$ and the inductive hypothesis holds for $i - 1$. Since $request \in arrows(v, w)$, the edge $\langle w, x_i \rangle$ points toward the root, and $request \notin arrows(w, x_i)$, condition $BndedFwdReq_2$ implies that either $request \in arrows(w, x_i)$ or $grant \in arrows(x_i, w)$ within time b. In the case that $request \in arrows(v, w)$, since the edge $\langle w, x_i \rangle$ points toward the root, the inductive hypothesis for $e-1$ implies that $grant \in arrows(x_i, w)$ within time $3be_i + b$. In either case, $grant \in arrows(x_i, w)$ within time $3be_i + 2b$. Since $request \in arrows(v, w)$ and $grant \in arrows(x_i, w)$, condition $BndedFwdGr_2$ implies that $grant \in arrows(w, x_j)$ within time b for some $x_j \in \{x_{i-1}, \ldots, x_1, v\}$. The

inductive hypothesis for $i-1$ implies that $grant \in arrows(w,v)$ within time $\sum_{j=1}^{i-1} 3b(e_j + 1)$, for a total of time $\sum_{j=1}^{i} 3b(e_j + 1)$ as desired. □

Finally, we have the following.

Theorem 52: If the graph G has e edges, then the response time in any b-bounded execution of A_2 is $3be - b$.

Proof: Let s be a state of A_2 in which $request \in arrows(u,a)$ for some user node u. Either $grant \in arrows(a,u)$ or the edge $\langle u,a \rangle$ points toward the root. In the case that $grant \in arrows(a,u)$, the condition $BndedRtnRes_2$ implies that $grant \in arrows(u,a)$ within time b. In either case, $request \in arrows(u,a)$ and the edge $\langle u,a \rangle$ points toward the root within time b. Lemma 51 implies that $grant \in arrows(a,u)$ within time $3be(u,a) + b = 3be - 2b$ for a total of time $3be - b$. □

Thus, as claimed, the response time in b-bounded executions is linear in the diameter of the network when the load on the arbiter is light, and linear in the size of the network when the load is heavy. We note that when an arbiter node grants the resource to an adjacent node, if it has received a request for the resource, it later forwards a request in the direction of the resource. As a result, three messages are sent over the edge to the adjacent node: the *grant* and *request* messages sent by the arbiter node, and a *grant* message sent to the arbiter node when the node receives the resource. Hence, the worst case response time of about $3be$. If, however, the arbiter node were to combine the *grant* and *request* messages sent to the adjacent node, then only two messages would traverse the edge between them. We note that in this case the worst case response time is $2be$. We have chosen to separate the messages in order to make the algorithm easier to describe.

Chapter 4

Conclusions

In this thesis we have introduced a new model of distributed computation in asynchronous systems. We find this model to be quite expressive, and find that the transparent, automata-theoretic semantics make reasoning about system behavior relatively simple. We have shown how the strong distinction between input and output actions captures the game-theoretic interplay between a system and its environment. This distinction has been found to be useful when describing the interface between system components, and when decomposing a system into modular components (see [Blo87]). We have found that the clarity of the interface between system components described by automata allows us to express the notion of fair computation quite simply and naturally. Finally, we have seen that automata may be used to construct hierarchical correctness proofs for distributed algorithms, allowing intuitive reasoning about key high-level ideas behind an algorithm's behavior to be incorporated into a formal proof of its correctness. While the framework developed in this thesis has proven to be quite useful, there are a number of ways in which it could be enhanced. We now consider a few of these enhancements.

First of all, it would be nice to find a more compact notation, a programming language, for defining automata than the precondition/effects style of presentation used in this thesis. In particular, since our work is in several ways similar to CCS, it would be nice to develop a CCS-like calculus having input-output automata as its underlying operational semantics. We note that one aspect of CCS that has not been developed for input-output automata is a powerful theory of equational reasoning. We do not know if such a theory can be associated with our model. Any results in this direction will certainly be valuable, for they will allow us to combine the transparent operational semantics of input-output automata with powerful semantic techniques for reasoning about system behavior.

As of yet, we have not attempted to characterize the expressive power of input-output automata. Our feeling that our model is generally quite powerful is the result of experience, and our feeling that certain aspects of the model (such as the requirement that an automaton be input-enabled) capture important aspects of asynchronous distributed computation. Bloom has made some initial attempts at characterizing the expressive

power of our model in [Blo86]. In particular, he has characterized the languages that can be expressed as the set of schedules of an automaton (resulting from arbitrary executions). Left uncharacterized are the languages that can be expressed as the set of schedules resulting from *fair* executions. Another possible characterization of interest is the relationship between the expressive power of temporal logic and our model. Wolper, Vardi, and Sistla show in [WVS83] that given a formula in a particular extension of temporal logic, it is possible to construct a Büchi automaton accepting precisely those sequences satisfying the given formula. It might be possible that these techniques can be adapted to prove a similar result for input-output automata.

We note that our model includes a single, simple notion of automaton composition. In particular, our composition requires that automata sharing an action π perform π simultaneously whenever π is performed by their composition. The intention is that if π is an output action of A and an input action of B, then the simultaneous performance of π models communication from A to B. We think of the performance of π as a computational step of A causing B to be notified of the arrival of input. However, since two processes in an asynchronous system cannot be expected to perform an action simultaneously, rather than complicating our notion of composition, we have chosen to require that the output actions of automata in a composition be disjoint. This has a number of effects on how systems are modeled with automata. For instance, to use Hoare's example of a vending machine (see [Hoa85]), suppose that we construct automata modeling humans, and an automaton modeling a vending machine. Humans can insert coins into the vending machine (output from humans and input to the vending machine). Since we require that the output actions of automata in a composition be disjoint, if we compose a collection of humans with the vending machine, each human's output action of inserting a coin must be tagged with an identifier. Thus, the vending machine is effectively able to determine which human is inserting a coin, which is not necessarily a realistic model of this simple interaction. It might be interesting to study other notions of composition that would avoid this problem. One such composition might require all automata having π as an input action to synchronize with precisely one automaton (the same for all) having π as an output action. While this is a natural notion of composition, the semantics of this composition complicate our model quite a bit. We feel that one virtue of our composition is that, as a consequence of Corollary 3, reasoning about the enabling of an action in a composition can be carried out by reasoning about the state of a single component. This has been found to be very convenient in [LM86].

While fair computation important to us, we have not made an explicit study of the nature of fairness in our model. In fact, we have defined only one of several possible notions of fairness (see [Fra86]). We feel that it should be possible to express many other notions of fairness in our model, and the study of these definitions in our model are of interest to us.

However, since the primary emphasis of this thesis has been the decomposition of correctness proofs, both hierarchically and modularly, we are naturally interested in

continuing the study of how automata can be used in new techniques of decomposition. We have already mentioned the work of [LS84a] and [LLW87]. The authors of these papers seem to be using a horizontal decomposition different from any considered in our work. In our work we have attempted to decompose systems into modular units that can be composed to yield the desired system. Once this decomposition has been made, each component can be examined in isolation, simplifying the verification process. In some systems, however, the system components are so heavily interdependent that no clean decomposition appears possible. [LS84a] and [LLW87] use the technique of "projecting" onto one system component (or algorithm component), abstracting the remaining system components to a high-level black box, and reasoning about the remaining "images." Notice that these images cannot be composed to yield a model of the system since each *is* a model of the complete system. The work of [LLW87] concerns how correctness proofs for each image can be combined into a correctness proof for the entire system. This work appears to be quite promising.

Finally, while this thesis has essentially ignored the notion of time, time is a very important part of modern distributed systems. Timeouts, for instance, are a crucial part of the fault-tolerance of many communication algorithms. Furthermore, complexity analysis of algorithms requires some notion of bounds on processor step times and message delivery times. We have shown, using rather *ad hoc* techniques, how rigorous reasoning about time complexity can be performed n our model. A very important problem is that of incorporating time into our model more naturally, and investigating useful properties about time that can be used to reason about time complexity of algorithms in our model. For instance, what does it mean to compose the timed equivalent of execution modules? Another important problem is that of relating complexity results obtained at different levels of abstraction. In our example, we analyzed the complexity of Schönhage's arbiter at a level of abstraction higher than the fully-detailed protocol, but it is not hard to see how this complexity result translates down to the lower level of abstraction. In general, however, relating time complexities at different levels of abstraction is a difficult problem. Such problems certainly deserve further study.

Bibliography

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Apt84] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part II: Nondeterminism. *Theoretical Computer Science*, 28(1,2):83–109, January 1984.

[AS85] Bowen Alpern and Fred B. Schneider. Verifying temporal properties without using temporal logic. Technical Report TR 85-723, Department of Computer Science, Cornell University, December 1985.

[AS86] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. Technical Report TR 86-727, Department of Computer Science, Cornell University, January 1986.

[Blo86] Bard Bloom. Unpublished notes, 1986.

[Blo87] Bard Bloom. Constructing two-writer atomic registers. In progress, 1987.

[CS84] Gerardo Costa and Colin Stirling. A fair calculus of communicating systems. *Acta Informatica*, 21(5):417–441, December 1984.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[EM72] Murray Eisenberg and Michael McGuire. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11):999, November 1972.

[FLMW87] Alan Fekete, Nancy Lynch, Michael Merritt, and William Weihl. Nested transactions and read/write locking. In *Proceedings of the Symposium on Principles of Database Systems*, 1987. To appear.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics, volume 19*, pages 19–32. American Mathematical Society, 1967.

[Fra86] Nissim Francez. *Fairness*. Springer-Verlag, Berlin, 1986.

[GHS83] Robert Gallagher, Pierre Humblet, and Phillip Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[GL87] Kenneth Goldman and Nancy Lynch. Quorum consensus in nested transaction systems. In progress, 1987.

[Gri77] David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, December 1977.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. To appear in *Science of Computer Programming*, 1987.

[HLMW87] Maurice Herlihy, Nancy Lynch, Michael Merritt, and William Weihl. On the correctness of orphan elimination algorithms. In progress, 1987.

[HO80] Brent Hailpern and Susan Owicki. Verifying network protocols using temporal logic. In *Proceedings Trends and Applications 1980: Computer Network Protocols*, pages 18–28. IEEE Computer Society, May 1980. Also appeared as Technical Report 192, Computation Systems Laboratory, Stanford University.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[KdR83] R. Kuiper and W. P. de Roever. Fairness assumptions for CSP in a temporal logic framework. In Dines Bjorner, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts–II*, pages 159–170. North-Holland Publishing Company, Amsterdam, 1983.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Lam80] Leslie Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14(3):21–37, September 1980.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[LF81] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13(1):17–43, January 1981.

[LG81] Gary Marc Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, June 1981.

[LGH+78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10(1):1–26, August 1978.

[LLW87] Leslie Lamport, Nancy A. Lynch, and Jennifer L. Welch. A hierarchical proof of a distributed algorithm. In progress, 1987.

[LM86] Nancy Lynch and Michael Merritt. Introduction to the theory of nested transactions. Technical Report MIT/LCS/TR-367, Laboratory for Computer Science, Massachusetts Institute of Technology, 1986.

[LS84a] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[LS84b] Leslie Lamport and Fred Schneider. The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[Lyn83] Nancy A. Lynch. Concurrency control for resilient nested transactions. Technical Report MIT/LCS/TR-285, Laboratory for Computer Science, Massachusetts Institute of Technology, February 1983.

[Lyn86] Nancy Lynch. Unpublished notes, 1986.

[MCS82] Jayadev Misra, K. Mani Chandy, and Todd Smith. Proving safety and liveness of communicating processes with examples. In *Proceedings of the 1st Annual ACM Symposium on Principles of Distributed Computing*, pages 201–208, August 1982.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer-Verlag, Berlin, 1980.

[MP81a] Zohar Manna and Amir Pnueli. Verification of concurrent programs: Temporal proof principles. In Dexter Kozen, editor, *Logic of Programs, Lecture Notes in Computer Science 131*, pages 200–252. Springer-Verlag, Berlin, 1981.

[MP81b] Zohar Manna and Amir Pnueli. Verification of concurrent programs: The temporal framework. In Robert S. Boyer and J. Strother Moore, editors, *The Correctness Problem in Computer Science*, International Lecture Series in Computer Science, pages 215–273. Academic Press, London, 1981.

[MP84] Zohar Manna and Amir Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4(3):257–289, December 1984.

[NGO85] Van Nguyen, David Gries, and Susan Owicki. A model and temporal proof system for networks of processes. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 121–131, January 1985.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, August 1976.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Par85] Joachim Parrow. *Fairness Properties in Process Algebra with Applications in Communication Protocol Verification.* PhD thesis, Department of Computer Systems, Uppsala University, Uppsala, Sweden, 1985.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*, pages 46–57, October 1977.

[Rei84] Wolfgang Reisig. Partial order semantics versus interleaving semantics for CSP-like languages and its impact on fairness. In Jan Paredaens, editor, *11th Colloquium on Automata, Langauges and Programming, Lecture Notes in Computer Science 172*, pages 403–413. Springer-Verlag, Berlin, July 1984.

[Sch80] Arnold Schonhage. Personal Communication, 1980.

[SMS81] Richard L. Schwartz and P. Michael Melliar-Smith. Temporal logic specification of distributed systems. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 446–454. IEEE Computer Society Press, April 1981.

[SS84] Richard D. Schlichting and Fred B. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, July 1984.

[Sta84] Eugene W. Stark. *Foundations of a Theory of Specification for Distributed Systems.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1984. Available as Technical Report MIT/LCS/TR-342.

[Wel87] Jennifer L. Welch. A synthesis of efficient mutual exclusion algorithms. In progress, 1987.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 185–194, November 1983.