Hierarchical Correctness Proofs for Distributed Algorithms

Nancy A. Lynch and Mark R. Tuttle

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract: We introduce the *input-output automaton*, a simple but powerful model of computation in asynchronous distributed networks. With this model we are able to construct modular, hierarchical correctness proofs for distributed algorithms. We define this model, and give an interesting example of how it can be used to construct such proofs.

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 the intuition underlying their algorithms, it is often difficult to make this intuition formal and precise. When this intuition *is* formalized, the result is typically an analysis performed at a very low level of abstraction, involving messages and local process variables. Reasoning precisely about the interaction between these messages and process 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 designers *are* able to convey an informal understanding of the key ideas behind their algorithms. The distributed minimum spanning tree algorithm of [GHS83], for example, can be understood 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 might *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 might then be proven correct using rigorous versions of the high-level, intuitive arguments given by the algorithm's designers. Successive refinements of this algorithm might then be defined at successively lower levels of detail, and each rigorously shown 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 describe the algorithm would become part of a rigorous proof of the full algorithm's correctness.

Some time ago, we began to consider this approach of proof by refinement for a simple resource allocation algorithm, an arbiter for a resource in an asynchronous network, originally suggested by Schönhage in [Sch80]. Correctness conditions for this resource arbitration problem include both safety and liveness conditions: the *mutual exclusion* condition that at most one user is holding 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 eventually grants the resource to every requesting user. The key idea be-

---

The full version of this paper is available as MIT Technical Report *MIT/LCS/TR*-387.

This work was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, by the National Science Foundation under Grants DCR-83-02391 and CCR-8611442, and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

This paper appeared in *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, ACM, August 1987.

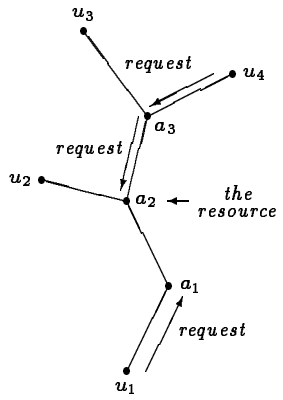

Figure 1: A state of Schönhage's arbiter.

hind the algorithm can be understood as follows. The algorithm assumes that the network forms an acyclic graph G as illustrated in Figure 1, with the users at the leaves of the graph, and the remaining nodes forming the arbiter itself. Initially, the resource is held by some arbiter node in the graph. A user desiring the resource sends a request for the resource to the adjacent arbiter node, and this node forwards this request in the direction of the node holding the resource. At any given time, there is a subtree of the graph rooted at the resource consisting of edges over which a request has been sent. The resource is allocated among the requesting users simply by causing the resource to traverse this tree of requests.

We found it convenient to view this algorithm at three levels of abstraction. At the highest level is a simple, set-theoretic statement of the correctness conditions required of the arbiter, this statement itself described as an algorithm. At the second level is a graph-theoretic description of the arbiter, similar to the one outlined above. At the third and lowest level is a distributed implementation of the arbiter, describing the protocol individual processors implementing the arbiter must follow in terms of messages and local process variables.

The first problem we face is how to express these algorithms describing the arbiter at the various levels of abstraction. Two of the most popular languages are Milner's CCS (see [Mil80]) and Hoare's CSP (see [Hoa85]), but certain aspects of the algorithms under consideration make it clear that these languages are not appropriate. In particular, the interaction between the users and the arbiter makes a clear distinction between those events internal and external to the arbiter. 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 the resource) occurs only under the control of the arbiter. This distinction between internal and external events is extremely important if reasoning about the system is to be decomposed into reasoning about system components in isolation, as was recognized by Barringer, Kuiper, and Pnueli in [BKP84]. Furthermore, it is clear that satisfaction of liveness conditions such as 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 distinction between internal and external events, on the ability to determine which process controls the performance of an action.

Unfortunately, neither CCS nor CSP makes such a distinction, and hence neither is able to express this notion of control. As a result, in the case of CCS, for example, notions of fairness considered are typically variants of *weak* or *strong fairness* (see [Fra86]). Weak fairness requires that an action π be performed infinitely often if it is continuously enabled, while strong fairness 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 such as the networks we consider. In such a system, a process is always able to accept interrupts, but should not be required to interrupt itself unless some external source requests the interrupt. Since there is no notion in CCS of an interface between processes from which we can deduce the internal and external actions of a process, variants of weak and strong fairness are essentially the only forms of fairness that can be expressed in CCS. Furthermore, as a side remark, we note that while the notion of a process state is not necessary for Milner's intended use of CCS, we find the notion a convenient descriptive tool, and useful when relating models of an algorithm at different levels of abstraction.

Similar comments can also be made for CSP with respect to fairness. 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 (fair) behavior of a process from its finite behaviors, CSP precludes the study of properties such as fairness without extending the semantics of a CSP process. We note, however, that the semantics of a CSP process is already quite complex due to the complexity of the compositions in CSP. Recall, for instance, that if P and Q are two processes, then $P \sqcap Q$ is a process that nondeterministically (itself) chooses to behave either like P or Q, while $P \Box Q$ is a process that allows the *environment*

to determine whether it 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. As a result of this "silent," internal choice between P and Q made by $P \sqcap Q$, it is possible to place $P \sqcap Q$ and $P \Box Q$ in an environment (offering an action π as input) in which $P \sqcap Q$ deadlocks at its first step while $P \Box Q$ does not. Reading between the lines of Hoare's book [Hoa85], it seems that Hoare would prefer to retain for nondeterministic processes the automata-theoretic (trace-theoretic) semantics he develops for deterministic processes. The processes $P \sqcap Q$ and $P \Box Q$, however, force 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. For our purposes, the fact that a process is able to accept input at all times should remove the entire problem resulting from the internal versus external nondeterminism illustrated above by $P \sqcap Q$ and $P \Box Q$. Furthermore, the complexity of operations allowed in CSP (such as blocking communication) do not seem appropriate when describing the loosely-coupled networks we have in mind. The semantic simplification gained by the elimination of such powerful operations should therefore more than make up for the resulting loss of expressive power. With hope, the result would be a general model of computation in which asynchronous distributed algorithms can be expressed without abandoning clean, automata-theoretic semantics.

We were therefore led to a new model of asynchronous distributed computation, 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; in other words, a process must be able to accept any input at any time. As a result, a very strong distinction is made between those actions locally-controlled by the system itself (output and internal actions) and those actions controlled by the system's external environment (input actions), and our model has the event-driven flavor characteristic of many asynchronous distributed algorithms.

Rather than the complex compositions allowed in CSP, we restrict ourselves to a very simple composition. Roughly speaking, the composition of a collection of automata is the Cartesian product of the automata, where automata are required to synchronize the performance of common (shared) actions. If π is an input action of A and an output action of B, then the simultaneous performance of π by both automata models the receipt of input at A caused by output generated at B. Since processes cannot be expected to synchronize the generation of output in asynchronous systems, we require that the output actions of the composed automata be disjoint. Similarly, since internal actions model externally undetectable actions, we require that the internal actions of each automaton be disjoint from the actions of the other automata in the composition. These restrictions on the composition of automata, together with the fact that the input actions of an automaton are enabled from every state, guarantee that locally-controlled actions of a composition are controlled by precisely one component of the composition.

As previously noted, the notion of fair computation plays a fundamental role in our work. Informally, a computation of a system is said to be fair if every system component is always eventually given the chance to take a computational step. Since one 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 the locally-controlled actions of each system component. We therefore associate with every automaton a partition of its locally-controlled 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.

It is clear that most verification methods, such as the Hoare-logic of Owicki and Gries in [OG76], the use of invariant assertions advocated by Lamport and Schneider in [LS84b], the temporal logic of Manna and Pnueli in [MP81b] and [MP81a], and the method of deriving proof obligations of Alpern and Schneider in [AS87], can be used to verify the correctness of algorithms expressed in terms of input-output automata. We do not fix on a particular methodology for reasoning about the behavior of individual automata. Instead, we study the problem of hierarchical decomposition, the problem of relating the algorithms describing the arbiter at different levels of abstraction. Lam and Shankar have successfully used notions of abstraction in their verification work [LS84a]. Their notion of abstraction involves the projection of an algorithm with several clearly distinguishable functions onto each function individually, abstracting away the details of the remaining functions. The result is a collection of smaller, simpler algorithms to analyze, allowing each function to be analyzed inde-

pendently. Since, however, we are describing algorithms at entirely different levels of conceptual abstraction, and not just ignoring certain aspects of the algorithms' behaviors, these techniques are not appropriate for our work. The use of abstraction in Harel's statecharts [Har87] is similar, in the sense that system states are grouped together to form superstates. Lamport has also advocated the use of abstraction in the specification of program modules [Lam83]. Lamport's specifications consists of a collection of *state functions* mapping program states into sets of values, a collection of *initial values* essentially defining the set of states in which the system may begin computation, and a collection of *properties* describing the safety and liveness conditions required. The intention of a state function is to extract some relevant information about the entity being implemented from the program state. For example, the specification of a queue might include a state function mapping each program state to the value in that state of the queue being implemented. Notice, however, that incorrectly chosen state functions can extract highly implementation-dependent information from the program state, and hence constrain the implementation of the specification. While a careful writer of specifications would never use such state functions, it illustrates how tightly a notion of correctness involving state functions can couple a specification to its implementation. We prefer a simpler notion of correctness, independent of program states, that allows us to construct independently descriptions of an algorithm at different levels of abstraction, and then relate these descriptions to each other.

Loosely speaking, we consider one automaton A to simulate a second automaton B if every behavior exhibited by A is a possible behavior of B. The automaton A simulates B in the sense that any correctness condition satisfied by the behaviors of B is satisfied by every behavior of A. As previously mentioned, however, fair computation is generally crucial to the satisfaction of most interesting liveness conditions. We therefore require only that the *fair* behaviors of A be contained in the *fair* behaviors of B. The simplicity of such correctness conditions lends a uniform structure to correctness proofs of algorithms. The problem of proving that our low-level description of the arbiter is a correct implementation of its high-level specification is simply the problem of proving that each description of the arbiter simulates the description of the arbiter at the previous (higher) level of abstraction. As an aid in doing so, we introduce the notion of a *possibilities mapping* between automata, relating the states of one automaton to the states of another. The notion of a possibilities mapping was first introduced by Lynch in [Lyn83] for process algebras, and we adapt these mappings for our own purposes. Possibilities mappings are similar in spirit to Lamport's state functions, but automata describing an algorithm at different levels of abstraction are independent of the possibilities mappings relating them. We remark that Stark has greatly generalized the notion of a possibilities mapping in [Sta84]. His model is much more general that ours, but we find ours simpler and easier to use, and expressive enough to describe most systems of interest.

The remainder of this paper consists of three parts. In the first part, we define the input-output automaton model. In the second part, we demonstrate how this model can be used to construct a modular, hierarchical correctness proof of Schönhage's distributed arbiter. Finally, we end with some concluding remarks.

2 The Model

We now formalize the notions introduced in the introduction. We begin with our model of computation.

2.1 Input-Output Automata

An *action signature* S is a collection of disjoint sets of input, output, and internal actions. We denote these sets by $in(S)$, $out(S)$, and $int(S)$, respectively, and their union by $acts(S)$. Since $int(S)$ is the set of internal actions, it is natural to refer to the actions of $in(S)$ and $out(S)$ as the set of *external actions*, denoted by $ext(S)$. Finally, we denote the set of locally-controlled actions, the actions of $int(S)$ and $out(S)$, by $local(S)$.

An *input-output automaton* A consists of

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 input action π and state a there is a transition of the form (a, π, a'); and
5. an equivalence relation $part(A)$ on $local(sig(A))$.

The equivalence relation $part(A)$ is the partition of the locally-controlled actions referred to in the introduction.

An element (a, π, a') of $steps(A)$ is referred to as a *step* of A. If (a, π, a') is a step of A, we say that the

action π is *enabled* from a. Since every input action is enabled from every state, we say that an automaton is *input-enabled*. An *execution* of A is a finite sequence $a_0\pi_1a_1\ldots\pi_ka_k$ or infinite sequence $a_0\pi_1a_1\pi_2a_2\ldots$ of alternating states and actions beginning with a start state such that each triple $(a_i, \pi_{i+1}, a_{i+1})$ is a step of A. A state is said to be *reachable* if it is the final state of a finite execution. The *schedule* of an execution e is the subsequence of actions appearing in e, denoted by $sched(e)$. We denote the sets of executions and schedules of A by $execs(A)$ and $scheds(A)$, respectively.

Since certain subsets of executions and schedules are of particular interest to us (such as the set of fair executions, for example), we are led to define the notions of execution modules and schedule modules, essentially sets of executions and 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. 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 signatures, and the executions of E are executions of A. We denote by $Execs(A)$ the execution module of A having as its set of executions the executions of 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 actions of S. Given an execution module E, there is a natural schedule module associated with E consisting of the action signature of E together with the schedules of the executions of E. We denote this schedule module by $Scheds(E)$, and write $Scheds(A)$ as a shorthand for $Scheds(Execs(A))$.

An *external action signature* is an action signature with no internal actions. As a special case of a schedule module, we define an *external schedule module* to be a schedule module with an external action signature. Given a schedule module S, we define the external action signature of S to be the action signature obtained by removing the internal actions from the action signature of S, and the external schedule module $External(S)$ to be the schedule module with the external action signature of S and the schedules obtained by removing from every schedule of S the internal actions of S. We write $External(E)$ for $External(Scheds(E))$ for an execution module E, and similarly for an automaton 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.

2.2 Composition

We now define the composition of automata, execution modules, and schedule modules.

Recall from the introduction that whether the composition of a collection of automata is defined depends on their action signatures. We say that the action signatures $\{S_i : i \in I\}$ are *compatible* if $out(S_i) \cap out(S_j) = \emptyset$ and $int(S_i) \cap acts(S_j) = \emptyset$ for every $i, j \in I$. 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 is defined to be the action signature with

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

Notice that the output and internal actions of the components become the output and internal actions of the composition, respectively, and that the remaining actions become the input actions of the composition.

The composition $A = \prod_{i\in I} A_i$ of compatible automata 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 $a_i = a_i'$.

In the case that I is the finite set $\{1, \ldots, n\}$, we denote the composition A by $A_1 \cdot \ldots \cdot A_n$. It is convenient to denote the execution of A_i induced by an execution e of the composition A by $e|A_i$. More formally, if $a = \{a_i\}$ is a state of the composition A, define $a|A_i = a_i$. If $e = a_0\pi_1a_1\ldots$ is an execution of A, define $e|A_i$ to be the sequence obtained by deleting π_ja_j if π_j is not

an action of A_i, and by replacing the remaining a_j by $a_j|A_i$.

The composition $E = \prod_{i \in I} E_i$ of compatible execution modules is defined as follows. Informally, if each E_i is an execution module of an automaton A_i, then E is the execution module of the composition $\prod_i A_i$ with executions e such that $e|A_i$ is an execution of E_i for every i. More formally, the states of E are $\prod_{i \in I} states(E_i)$ and the action signature of E is $\prod_{i \in I} sig(E_i)$. If $a = \{a_i\}$ is a state of E, define $a|E_i = a_i$. If $e = a_0 \pi_1 a_1 \ldots$ is a sequence of states and actions of E, define $e|E_i$ to be the sequence obtained by deleting $\pi_j a_j$ if π_j is not an action of E_i, and replacing the remaining a_j by $a_j|E_i$. The executions of E are those sequences $e = a_0 \pi_1 a_1 \ldots$ of states and actions of E such that $e|E_i$ is an execution of E_i for every i, and $a_{j-1}|E_i = a_j|E_i$ whenever π_j is not an action of E_i.

The composition $S = \prod_{i \in I} S_i$ of compatible schedule modules is defined as follows. The action signature of S is $\prod_{i \in I} sig(S_i)$. If s is a sequence of actions of S, define $s|S_i$ to be the subsequence of s consisting of actions of S_i. The schedules of S are those sequences s of actions of S such that $s|S_i$ is a schedule of S_i for every i.

These compositions are clearly related. For example, the execution module of a composition of automata is the composition of the execution modules of the automata. Notice that actions shared by several objects are not hidden by these compositions. In the full paper [LT87] we define a simple operation to hide such actions, merely relabeling a set of actions as internal actions. Notice also that the compatibility of a collection of objects, and hence whether their composition is defined, depends solely on their action signatures. In the full paper we define a simple operation to rename the actions of an object and thereby avoid incompatibility due to naming conflicts.

2.3 Fairness

Informally, computation in a system of processes is said to be fair if every system component is allowed to make computational progress infinitely often. Recall that associated with an automaton A is a partition $part(A)$ of its locally-controlled actions, where each class is interpreted as the set of locally-controlled actions of one component in the system modeled by A. An execution e of A is said to be *fair* if the following conditions hold for each class C of $part(A)$:

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

These conditions may be interpreted as follows. If e is finite, then computation in the system has halted since no process is able to take another step. If e is infinite, then every process has been given an infinite number of chances to take a step, although it may be that some processes were unable to take steps every time the chance was offered. This notion of fairness is similar to *weak fairness* (see [Fra86]), except that the performance of input actions is never required.

We denote the set of fair executions of an automaton A by $fair(A)$, and the execution module of A have $fair(A)$ as its set of executions by $Fair(A)$. An important property of this definition of fairness is the fact that the fair executions of a composition are the composition of the fair executions of the components: that is, $Fair(\prod_i A_i) = \prod_i Fair(A_i)$. In the full paper [LT87], we explore several consequences of our definition of fair computation not directly related to algorithm verification, including an interesting notion of process equivalence induced by fair computations.

2.4 Correctness

As mentioned in the introduction, we consider an automaton A to simulate an automaton B if the fair behavior of A is contained in the fair behavior of B. More formally, we define the schedule module $Fbeh(A) = External(Fair(A))$ to be the (externally observable) fair behavior of A, and denote the schedules of $Fbeh(A)$ by $fbeh(A)$. Viewing the automaton B as a specification satisfied by the automaton A, we say that A *satisfies* B if A and B have the same external action signature and $fbeh(A) \subseteq fbeh(B)$. A satisfies B in the sense that every correctness condition satisfied by the fair behavior of B is satisfied by the fair behavior of A. Notice that since automata are input-enabled, the trivial satisfaction of a specification by an automaton exhibiting no behavior is not possible. We extend these definitions to objects of arbitrary type by defining $Fbeh(O) = External(O)$ for execution modules and schedule modules O. Notice that since execution modules need not be nonempty, it is possible for an execution module with no executions to satisfy every execution module with the same external action signature. Therefore, we say that an object O is *implementable* if it is satisfied by an automaton A. The object O is implementable in the sense that there is a system (modeled by the automaton A) satisfying the specification represented by the

object O. We say that an object O *solves* (the problem specified by) an object O' if O is an implementable object satisfying O'. Notice that if an automaton A satisfies an automaton B, then A certainly solves B.

Clearly, the notion of satisfaction is the basis of the definitions stated above. In the remainder of this section we exhibit a sufficient condition for one automaton to satisfy another. The key to this sufficient condition is the notion of a possibilities mapping. 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 π is an action of B, then there is a step (b, π, b') of B such that $b' \in h(a')$.
   (b) If π is not an action of B, then $b \in h(a')$.

Such a mapping, reminiscent of bisimulation from CCS [Mil80], enables us to relate executions of A to executions of B as follows. If e and f are two finite executions of A and B, respectively, we say that f *finitely corresponds* to e under h if $sched(f) = sched(e)|B$ and the final state of f is contained in the image of the final state of e under h. In general, if e and f are two executions of A and B, we say that f *corresponds* to e under h if for every finite prefix $e_i = a_0\pi_1a_1 \ldots a_i$ of e there is a finite prefix f_i of f finitely corresponding to e_i under h such that f is the limit of the f_i. It is easy to show that if h is a possibilities mapping from A to B and e is an execution of A, then there is an execution f of B corresponding to e under h. The existence of a possibilities mapping is a useful relationship between the automata A and B since it allows us to relate the states appearing in executions of A to the states appearing in corresponding executions of B. We now show how this relationship can be used to prove that A satisfies B.

Let S and T be two sets of states, and let Π be a set of actions. Given an execution $e = a_0\pi_1a_1 \ldots$, the execution e satisfies the condition $S \hookrightarrow (T, \Pi)$ if whenever the execution passes through a state of S, eventually either the execution passes through a state of T or an action from Π is performed.[1] More formally, e satisfies the condition $S \hookrightarrow (T, \Pi)$ if, whenever $a_i \in S$, either $a_j \in T$ for some $j \geq i$ or $\pi_j \in \Pi$ for some $j > i$. As a notational convenience, we will denote the condition $S \hookrightarrow (T, \Pi)$ by $S \hookrightarrow \Pi$ when the set T is empty. Notice, however, that the fair executions of an automaton A are precisely those executions satisfying the conditions $states(A) \hookrightarrow (disabled(C), C)$ where C is a class of $part(A)$ and $disabled(C)$ is the set of states from which no action of C is enabled. It is straightforward to prove the following.

Lemma 1: Let h be a possibilities mapping from A to A'. Let e be an execution of A, and let e' be an execution of A' corresponding to e under h.

1. Let $S \supseteq h^{-1}(S')$, $h(T) \subseteq T'$, and $\Pi \subseteq \Pi'$. If e satisfies $S \hookrightarrow (T, \Pi)$, then e' satisfies $S' \hookrightarrow (T', \Pi')$.
2. Let $h(S) \subseteq S'$, $T \supseteq h^{-1}(T')$, and $\Pi \supseteq \Pi'$. If e' satisfies $S' \hookrightarrow (T', \Pi')$, then e satisfies $S \hookrightarrow (T, \Pi)$.

As a result, a possibilities mappings can be used as part of a sufficient condition for an automaton A to satisfy an automaton B, as claimed.

3 A Distributed Arbiter

In this section, we sketch how the ideas introduced in the previous section can be used to construct a modular, hierarchical correctness proof for Schönhage's distributed arbiter.

3.1 A High-Level Model

In our high-level model of the arbiter, the automaton A_1, we refer to the arbiter itself as a, and to the users of the arbiter as $u_1, \ldots, u_n$. A state of A_1 consists of a set *requesters* of requesting users, together with the identity *holder* of 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 itself. The actions of A_1 are given in Figure 2.[2] A user u requests the resource with the input action $request(u)$, which simply places u in the set *requesters* of requesting users. The user u returns the resource to the arbiter with the input action $return(u)$. If the user is actually holding the resource

[1] This condition may also be expressed in the temporal logic of [BKP84].

[2] We define the transition relation of an automaton by defining the preconditions and effects of every action. The triple (a, π, a') is a transition of the automaton if the state a satisfies the precondition of π, and the state a' can be obtained from a by modifying a as specified by the effects of π. The precondition for an action is omitted if it is *true*, as is the case for input actions.

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 2: The actions of A_1.

when it tries to return the resource, this action makes the arbiter the new holder of the resource. If the user is not actually holding the resource, this "return" is ignored. The arbiter grants the resource to a requesting user u with the output action *grant*(u). This action merely removes u from the set of requesting users and makes u the new holder of the resource.

Notice that since at most one user is holding the resource at any time, every execution of A_1 satisfies the mutual exclusion condition. The satisfaction of the no lockout condition, however, clearly requires some cooperation from the users. Let u be a user node, and let us define the following sets of states and actions.[3]

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

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

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

$$NoLockout_1 = RtnRes_1 \supset GrRes_1$$

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 $NoLockout_1$. The execution module E_1 serves as our specification of the arbiter.

3.2 An Intermediate-Level Model

Our second model of the arbiter is essentially the global description given in the introduction (see Figure 1). In this model, the arbiter and its users are modeled by an undirected, 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 $a_1, \ldots, a_m$. Arrows are placed on edges of the graph to indicate either a request for the resource or the granting of the resource. The (directed) edge of G from the node v to w is denoted by $\langle v, w \rangle$. With every edge $\langle v, w \rangle$ we associate a set *arrows*(v, w) containing the arrows on the edge $\langle v, w \rangle$. The states of A_2 are determined by the sets *arrows*(v, w). The start states of A_2 are chosen from those states in which all arrow sets are empty, except that one arrow set *arrows*(v, a_i) contains a *grant* arrow for some arbiter node a_i. 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. Therefore, the initial states are chosen from those states in which no requests are pending and an arbiter node is the root of the graph. The particular set of start states chosen is of no importance at the moment, so we will defer the choice until the next section.

A user u_i requests the resource with the input action *request*(u_i, a_j), placing a *request* arrow on the edge $\langle u_i, a_j \rangle$ from itself to the adjacent arbiter node a_j. The arbiter grants the resource to u_i with the output action *grant*(a_j, u_i), removing this *request* arrow from $\langle u_i, a_j \rangle$ and placing a *grant* arrow on $\langle a_j, u_i \rangle$. The user returns the resource with the input action *grant*(u_i, a_j), moving the *grant* arrow from the edge $\langle a_j, u_i \rangle$ to the edge $\langle u_i, a_j \rangle$. In general, if an arbiter node a_j finds itself at the head of a *request* arrow, its response depends on whether it is holding the resource or not. If the arbiter holds the resource, then it must be at the head of a *grant* arrow, and so there must be

[3] 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 define these conditions will be superscripted with the letters s and a, respectively.

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: The actions of A_2.

a *grant* arrow on some edge $\langle w, a_j \rangle$. The arbiter node selects the first node v in some fixed ordering of its adjacent nodes having a *request* arrow on $\langle v, a_j \rangle$. The arbiter then grants the resource to this node with the action $grant(a_j, v)$, removing the *request* arrow and moving the *grant* arrow to the edge $\langle a_j, v \rangle$. If the arbiter node a_j does not hold the resource, then the arbiter forwards the request in the direction of the node holding the resource with the action $request(a_j, v)$, placing a *request* on the edge $\langle a_j, v \rangle$ pointing toward a root (that is, the edge $\langle a_j, v \rangle$ in the path from a_j to the root). The actions of A_2 are formally defined in Figure 3. Here we fix for each node an ordering of its adjacent nodes. We denote by (v, w) the set of nodes strictly between v and w in this ordering, and by $(v, w]$ the set of nodes (v, w) together with w. The external actions of A_2 are the actions $request(u_i, a_j)$, $grant(u_i, a_j)$, and $grant(a_j, u_i)$; and the remaining actions are internal actions. For technical convenience, we remove all potentially unreachable states from A_2 so that all states are reachable.

Straightforward inductive arguments show that the automaton A_2 satisfies the following invariants:

Lemma 2: Every state of A_2 has precisely one root.

Lemma 3: 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.

The first invariant, Lemma 2, shows that A_2 satisfies the mutual exclusion condition. However, in order to ensure that the arbiter satisfies the no lockout condition, it is clearly important that 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.

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

$$\begin{aligned} FwdGr_2^s(a,v,w) = \; & \{s \in states(A_2) : \\ & request \in arrows(v,a) \text{ and} \\ & grant \in arrows(w,a) \text{ in } s\} \\ FwdGr_2^a(a,v,w) = \; & \{grant(a,y) : y \in (w,v]\} \end{aligned}$$

The first arbiter correctness condition

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

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

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

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 define E_2 to be the execution module of A_2 with those executions of A_2 satisfying the condition C_2.

While Lemma 2 states that executions of E_2 satisfy the mutual exclusion condition, 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 every execution of E_2 satisfies the no lockout condition. 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.

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

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

The condition

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

says that 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 that the arbiter always satisfies requesting users. The condition

$$NoLockout_2 = RtnRes_2 \supset GrRes_2$$

says that if users return the resource, then the arbiter satisfies all requests. It is fairly simple to show that

Lemma 4: Every execution of E_2 satisfies the condition $NoLockout_2$.

With this, we are now ready to show that the execution module E_2 satisfies the execution module E_1, the specification of the arbiter. Recall that one requirement for E_1 to satisfy E_2 is that both execution modules have the same external action signature. For the sake of exposition, however, we have given the actions of A_2 names reflecting their level of abstraction, rather than using the names of the actions of A_1. It is a simple matter to rename the (external) actions of A_2 and E_2 (yielding A_2' and E_2') to be consistent with those of A_1 and E_1 by renaming $request(u,a)$ as $request(u)$, $grant(u,a)$ as $return(u)$, and $grant(a,u)$ as $grant(u)$. Having done so, we construct the mapping h_1 mapping the state s of A_2' to the state t of A_1 such that

$u \in requesters$ in t iff
$\quad request \in arrows(u,a)$ in s
$holder = u$ in t iff
$\quad grant \in arrows(a,u)$ in s
$holder = a$ in t iff
$\quad grant \notin arrows(a,u)$ for every user u in s

It is a routine matter to prove that

Lemma 5: The mapping h_1 is a possibilities mapping from A_2' to A_1.

Furthermore, using Lemma 1 and the possibilities mapping h_1, it is easy to prove that

Lemma 6: E_2' satisfies E_1.

3.3 A Low-Level Model

Previous models have given global descriptions of the arbiter's behavior. In the description of the arbiter given above, the arbiter nodes are intended to represent processes in a distributed network implementing the arbiter. In this model we actually distribute the arbiter by modeling each process with a separate automaton. These automata describe the low-level protocol followed by each process implementing the arbiter. 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 system as an independent automaton. By composing the automata modeling arbiter processes together with the automaton modeling the message system, we obtain a global model of the arbiter.

In order to model asynchronous message delivery at the intermediate level of abstraction (with the automaton A_2), 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 sending of a message from a to a' corresponds to placing an arrow on the edge $\langle a, b_{a,a'} \rangle$, and the delivery of the message by the message system corresponds to 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*, and not arbiter nodes. We denote by $\mathcal{G}$ the graph obtained from G by the addition of such buffer nodes. Two user or arbiter nodes (processes) are said to be *neighbors* in $\mathcal{G}$ if they are separated by at most a buffer node. 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. With this added structure of $\mathcal{G}$, we can prove the following invariant concerning buffer nodes during executions of A_2.

Lemma 7: Let a and a' be adjacent arbiter nodes, and let s be a state of A_2. If $request \in arrows(b_{a,a'}, a')$ or $grant \in arrows(a', b_{a,a'})$, then $request \in arrows(a, b_{a,a'})$.

That is, if the arbiter node a has not sent a *request* to the arbiter node a', then there will be no *request* in transit from a to a' and no *grant* in transit from a' to a.

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 grant 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 *requesting* of processes from which it has received a request for the resource, the link *lastforward* over which the resource was last sent, a flag *holding* indicating whether the process is holding the resource, and a flag *requesting* indicating whether the process has sent a request in the direction of the resource. Initially, some arbiter process is designated as the initial holder of the resource, and is known to all processes in the network. For each arbiter process a, each arbiter (nonbuffer) node of $\mathcal{G}$, we construct an automaton A_a modeling the process a. The actions of A_a are given in Figure 4. Here, v is a neighbor of 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. More formally, the state of the message system consists of a set *messages* of triples of the form $(a, a', request)$ and $(a, a', grant)$ denoting *request* and *grant* messages, respectively, to be delivered from a to a'. We construct an automaton M to model the asynchronous message system. The actions of M are given in Figure 5. Here, a and a' are neighboring arbiter nodes.

The global model A_3 of the arbiter at this low level of abstraction is constructed by composing the automata A_a modeling the arbiter processes together with the automaton M modeling the message system, and hiding actions inherently internal to the global system (that is, the actions of the message system M).

As mentioned in the introduction to this model, an

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 4: The Actions of A_a.

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 neighbors of the arbiter process a, and let us define the following sets of states and actions.

$$\begin{aligned} FwdReq_a^s(v) = \; & \{s \in states(A_a) : requesting \neq \emptyset, \\ & \quad requested = false, \\ & \quad holding = false, \text{ and} \\ & \quad lastforward = v \text{ in } s\} \\ FwdReq_a^a(v) = \; & \{receivegrant(v, a), \\ & \quad sendrequest(a, v)\} \end{aligned}$$

$$\begin{aligned} FwdGr_a^s(v, w) = \; & \{s \in states(A_a) : v \in requesting \\ & \quad holding = true, \text{ and} \\ & \quad lastforward = w \text{ in } s\} \\ FwdGr_a^a(v, w) = \; & \{sendgrant(a, y) : y \in (w, v]\} \end{aligned}$$

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 for-

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 5: The actions of M.

ward 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 8: Every fair execution of A_a satisfies C_a.

We let the execution module $E_a = Fair(A_a)$. The execution module E_a is clearly an implementable execution module.

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

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

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

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. We note that since we are implementing the arbiter protocol and not the message system, we have not constructed a *particular* automaton whose fair executions satisfy the condition C_M. In the full paper, however, we do prove that E_M is implementable, being satisfied by a message system delivering messages in the order they are received.

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. Since the component execution modules are implementable, so is E_3.

Again, having named the actions of A_3 for the convenience of exposition, we must rename the actions of A_3 to correspond to the actions of A_2 at the higher level of abstraction before proving that the execution module E_3 satisfies the execution module E_2. After renaming the actions of A_3 and E_3 (yielding A_3' and E_3') to be consistent with the names of actions of A_2 and E_2, we construct a possibilities mapping h_2 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 $requesting_a$. 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 $U1 - U3$ and $A1 - A3$ 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; 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 to v; and that a has forwarded a request for the resource toward the root in t iff a has sent a request in the direction it last forwarded the resource in s. $U4$ says that the user u has the resource in t iff in s the node a last forwarded the resource to u and has not received the resource since. $A4$ 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. Conditions $I1$ and $I2$ are invariants that must be preserved by the mapping. $I1$ says that if a *request* message is in transit in s, then it must not have been received in t. $I2$ says that the value of *lastforward* correctly records the direction of the resource in the network.

We first prove that

Lemma 9: The mapping h_2 is a possibilities mapping from A_3' to A_2.

Using Lemma 1, we then prove that

Lemma 10: E_3' satisfies E_2.

We note that the proofs Lemmas 9 and 10 make heavy use of invariants such as Lemmas 2, 3, and 7 proven at the intermediate level of abstraction, in addition to the invariants $I1$ and $I2$ satisfied by the mapping h_2. As a result, properties proven at a higher level of abstraction are actually used in the correctness proof of the low-level implementation.

Renaming the actions of E_3' to agree with those of E_1 to obtain E_3'', we use Lemmas 6 and 10 together with the fact that E_3'' is implementable to show that

Theorem 11: E_3'' solves E_1.

With this, the proof of the correctness of a fully-detailed implementation of Schönhage's resource allocation is complete.

4 Conclusions

In this work we have introduced a simple, powerful model of computation in asynchronous distributed networks. We have shown how this model can be used to construct modular, hierarchical correctness proofs of distributed algorithms. The technique demonstrated in this paper shows that the high-level, intuitive understanding of an algorithm's designer need not be sacrificed for the sake of rigorous correctness. To the contrary, this technique makes use of the intuition, incorporating it into a proof of the detailed algorithm's correctness. We feel that the strongest evidence for the usefulness of the input-output automaton model, however, is the fact that it has already been used successfully by others modeling a variety of distributed algorithms. Examples of the use of input-output automata differing from the arbiter example discussed in this work include concurrency control algorithms (see [LM86], [HLMW87], [FLMW87], and [GL87]), mutual exclusion algorithms (see [Wel87]), hardware register algorithms (see [Blo87]), and dataflow computation (see [Lyn86]). In many of these papers the model has been found to be especially helpful when attempting to identify the interface between system components, and discovering a system's natural decomposition. While our work has described one method for proving the correctness of algorithms within the input-output automaton model, others are being examined. Work in progress ([FLS87], for example) continues to explore ways in which this model can be used to decompose systems and their proofs of correctness.

One important question related to this work is that of how much of the correctness proofs we generate can be checked by machine. In general, the use of correctly chosen possibilities mappings to prove that one object satisfies another is quite mechanical and should be checkable by machine. In the case of the arbiter examined in this paper, for example, each description of the arbiter is essentially a finite state description, with the exception of the message system described at the lowest level of abstraction. It is fairly simple, however, to characterize the behavior of the message system in terms of temporal logic. One interesting question is whether it is possible to use a temporal logic characterization of the message system together with finite state descriptions of the remainder of the

arbiter to mechanically verify the satisfaction of one description of the arbiter by another.

While this work 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 communications algorithms. Furthermore, the analysis of an algorithm's complexity requires some notion of bounds on processor step times and message delivery times. We give in the full paper [LT87] some *ad hoc* techniques for reasoning about the time complexity of the arbiter discussed in this paper. We analyze the complexity of the arbiter at the intermediate-level of abstraction, and 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. The problem of incorporating time more naturally into our model and of investigating techniques for reasoning about time in our model certainly deserve further study.

Acknowledgments

Much of the preliminary work for this paper (including the conception of the input-output automaton) was done concurrently with the work of the first author and Michael Merritt in [LM86], and that work has greatly influenced this paper. Conversations with and the experience of Alan Fekete, Ken Goldman, and Jennifer Welch as they have used input-output automata in their work have also been very helpful. The presentation of this work was improved by comments from Leslie Lamport and Liuba Shrira.

References

[AS87] Bowen L. Alpern and Fred B. Schneider. Proving boolean combinations of deterministic properties. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, June 1987.

[BKP84] Howard Barringer, Ruurd Kuiper, and Amir Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th ACM Symposium on Theory of Computing*, pages 51–63. ACM, April 1984.

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

[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.

[FLS87] Alan Fekete, Nancy A. Lynch, and Liuba Shrira. A modular proof of correctness for a network synchronizer. In *Proceedings of the 2nd International Workshop on Distributed Algorithms*, July 1987.

[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.

[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.

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

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

[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.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR–387, MIT Laboratory for Computer Science, April 1987.

[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.

[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.

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

[Sch80] Arnold Schonhage. Personal Communication, 1980.

[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.