An Introduction to Input/Output Automata

Nancy A. Lynch and Mark R. Tuttle
Massachusetts Institute of Technology
Cambridge, Mass. 02139

November 18, 1988

1 Introduction

The *input/output automaton* model has recently been defined, in [LT1, LT2], as a tool for modeling concurrent and distributed discrete event systems of the sorts arising in computer science. Since its introduction, the model has been used for describing and reasoning about several different types of systems, including network resource allocation algorithms, communication algorithms, concurrent database systems, shared atomic objects, and dataflow architectures.

This paper is intended to introduce researchers to the model. It is organized as follows. Section 2 contains an overview of the model. Section 3 defines the model formally and examines several illustrative examples concerning candy vending machines. Section 4 contains a second example, a leader election algorithm. Finally, Section 5 contains a survey of some of the uses that have so far been made of the model.

2 Overview of the Model

I/O automata provide an appropriate model for discrete event systems consisting of concurrently-operating components. Such systems are often characterized by the fact that, instead of simply computing some function of their input and

This research was supported in part by the National Science Foundation under Grant CCR-86-11442, by the Office of Naval Research under Contract N00014-85-K-0168 and by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125. The second author was also supported by a GTE Graduate Fellowship and an IBM Graduate Fellowship.

This article appeared in *CWI Quarterly*, 2(3):219–246, September 1989, and is also available as MIT Technical Memo MIT/LCS/TM-373.

halting, they continuously receive input from and react to their environment. Although I/O automata can be used to model synchronous systems, they are best suited for modeling systems in which the components operate asynchronously.

Each system component is modeled as an I/O automaton, which is essentially a (possibly infinite state) automaton with an action labeling each transition. A fundamental property of our model is that we make a very clear distinction between those actions whose performance is under the control of the automaton and those actions whose performance is under the control of its environment. An automaton's actions are classified as either 'input', 'output', or 'internal'. An automaton generates output and internal actions autonomously, and transmits output instantaneously to its environment. In contrast, the automaton's input is generated by the environment and transmitted instantaneously to the automaton. Our distinction between input and other actions is fundamental, based on who determines when the action is performed: an automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action.[1]

The fact that our automata are unable to block inputs distinguishes our model from CSP (Communicating Sequential Processes) [Ho]. There, input blocking is used for two purposes: as a way of blocking the activity of the environment and as a way of eliminating undesirable inputs. Our model does not allow an automaton to block its environment or eliminate undesirable inputs. Suppose, however, that we only wish to guarantee that an automaton exhibits some behavior when the environment observes certain restrictions on the production of inputs. Instead of allowing the automaton to block the bad inputs, we permit these inputs to occur, but permit the automaton to exhibit arbitrary behavior when they do. Our correctness conditions are often of the form 'if the environment behaves correctly, then the automaton behaves correctly.' Alternatively, our correctness condition may require the automaton to detect bad inputs and respond to them with error messages. In either case, we have simple ways of dealing with input restrictions without including input-blocking in the model.

I/O automata may be nondeterministic, and indeed the nondeterminism is an important part of the model's descriptive power. Describing algorithms as nondeterministically as possible tends to make results about the algorithms quite general, since many results about nondeterministic algorithms apply a fortiori to all algorithms obtained by restricting the nondeterministic choices. Moreover, the use of nondeterminism helps to avoid cluttering algorithm descriptions and proofs with inessential details.

I/O automata can be composed to yield other I/O automata. When we compose a collection of automata, we identify the same-named actions of the different automata. Our composition guarantees that if one automaton has π

[1] The shared-memory model described in [LF] has had a strong influence on the present work. In particular, the inability to block inputs appears as the 'read-anything' property in [LF].

as an output action, then π is an input action of all remaining automata having π as an action. As a result, an automaton generating an output action does so autonomously, and this output is transmitted instantaneously to all other automata having the same action as an input. All such components are passive recipients of the input, and take steps simultaneously with the output step. As in CSP, we use simultaneous performance of actions to synchronize components, but we permit only one component to determine when the action occurs.

When I/O automata are run, they generate 'executions' (alternating sequences of states and actions). Among all the executions of an automaton, we are primarily interested in the 'fair' executions — those that permit each of the automaton's primitive components to have infinitely many chances to perform output or internal actions. The fair executions of an automaton give rise to the 'fair behaviors' of the automaton — the subsequences of the fair executions that consist of external (that is, input and output) actions. It is this set of sequences that we believe embodies the interesting behavior of an I/O automaton; thus, our semantics is a 'trace' semantics. The set of fair behaviors of an I/O automaton can consist of both finite and infinite sequences of actions, and is not necessarily closed under the operation of taking prefixes.

A 'problem' to be solved by an I/O automaton is formalized essentially as an arbitrary set of (finite and infinite) sequences of external actions. Our notion of what it means for an automaton to 'solve' a problem is particularly simple: essentially, an automaton is said to 'solve' a problem P provided that its set of fair behaviors is a subset of P. It might not be obvious to the reader that this definition is nontrivial; for example, if an automaton had no fair behaviors, then our definition would say that it is a solution to every problem. However, this anomaly does not arise, since our definitions imply that every automaton has a nonempty set of fair behaviors. Since an automaton cannot block its input, for every possible pattern of inputs that might arrive from the environment, the automaton is required to provide some response such that the resulting sequence of actions is in the problem set P. That is, the automaton is required to respond appropriately to every possible input pattern.

The model permits description of algorithms and systems at different levels of abstraction. Abstraction mappings are defined, mapping automata that include implementation detail to more abstract automata that suppress some of the detail. Such mappings can be used as aids in correctness proofs for algorithms: if automaton A is an image of B under an appropriate abstraction mapping and A solves problem P, then B also solves P.

The model allows very careful and readable descriptions of particular concurrent algorithms. We have developed a simple language for describing automata, based on 'precondition' and 'effect' specifications for actions. This notation, similar to Dijkstra's 'guarded commands,' has proved sufficient for describing all algorithms we have attempted so far. However, the model does not depend on this manner of describing automata; for example, the model is general enough to serve as a formal basis for languages that include more elaborate constructs for sequential flow of control.

Our model also allows precise statements of the problems that are to be solved by modules in concurrent systems. As described above, such problems are formulated as sets of finite and infinite sequences of external actions. We have not so far developed any particular language or notation for describing such sets, but have used a variety of notations (e.g., temporal logic or generating automata) as they have seemed convenient. Again, our model is general enough to serve as an operational model for many different languages describing sets of action sequences.

The model can be used as a formal basis for algorithm correctness proofs — proofs that particular algorithms solve particular problems in the sense described above. In fact, a current major thrust of our research involves producing correctness proofs for substantial-sized and complex concurrent algorithms. We use a variety of techniques for such proofs, primarily based on notions of composition and abstraction. In every case, we try to utilize the modularity that is suggested by informal descriptions of the algorithm in our formal correctness proofs. So far, our proofs have been done by hand, but it appears that machine-checking of some of our proofs might be possible using current automatic proof technology.

The model can also be used for carrying out complexity analysis, proving upper and lower bounds on the complexity of solving particular problems, and proving impossibility results.

3 The Input/Output Automaton Model

In this section we formally define our model of computation, show how it can be used to model a system, how it can be used to construct a problem specification, and how it can be used to prove that a system satisfies a specification.

3.1 Input/Output Automata

We begin with the definition of an automaton. As previously mentioned, an automaton's actions are partitioned into sets of input, output, and internal actions. This set of actions and its partition determines an interface between the automaton and its environment. We refer to this interface as the action signature of the automaton. Formally, an *action signature* S is a partition of a set $acts(S)$ of *actions* into three disjoint sets $in(S)$, $out(S)$, and $int(S)$ of *input actions*, *output actions*, and *internal actions*, respectively. We denote by $ext(S) = in(S) \cup out(S)$ the set of *external actions*, those actions visible to the environment of any automaton have S as its action signature. An *external action signature* is an action signature S with no internal actions; that is, $int(S) = \emptyset$ or $acts(S) = ext(S)$. Given an action signature S, we define $extsig(S)$ to be the external action signature S' with $in(S') = in(S)$ and $out(S') = out(S)$.

We denote by $local(S) = out(S) \cup int(S)$ the set of *locally-controlled actions*, those actions under the local control of any automaton having S as its action signature. Given an automaton A with action signature S, we will frequently abuse notation and denote $in(S)$ by $in(A)$, etc.

An *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of five components:

- an action signature $sig(A)$,
- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*,
- a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$, and
- an equivalence relation $part(A)$ partitioning the set $local(A)$ into at most a countable number of equivalence classes.

Since the equivalence relation $part(A)$ is used only in the definition of fair computation in Section 3.3, we will ignore it for now. It is used to identify the primitive components of the system being modeled by the automaton: each class is thought of as the set of actions under the local control of one system component.

Each element of an automaton's transition relation represents a possible step in the computation of the system the automaton models. We refer to an element (s', π, s) of $steps(A)$ as a *step* of A. If (s', π, s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled*. This means that the automaton is unable to block its input, which is one of the fundamental assumptions made in our model (the other being that the performance of an action is controlled by at most one system component).

When an automaton 'runs,' it generates a string representing an execution of the system the automaton models. An *execution fragment* of A is a finite sequence $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of A for every i. An *execution* is an execution fragment beginning with a start state. We denote the set of executions of A by $execs(A)$, and the set of finite executions of A by it finexecs(A). We say that a state is *reachable* if it is the final state of a finite execution.

While an execution represents a system computation, we are often interested only in the sequence of actions performed during the course of the computation, and not in the states through which the computation passes. The *schedule* of an execution fragment α is the subsequence of α consisting of the actions appearing in α, and is denoted by $sched(\alpha)$. We say that β is a *schedule* of an automaton

A if β is the schedule of an execution of A. We denote the set of schedules of A by *scheds*(A), and the set of finite schedules of A by *finscheds*(A). The *behavior* of an execution or schedule α of A is the subsequence of α consisting of *external* actions, and is denoted by *beh*(α). Intuitively, *beh*(α) is the externally observable portion of α , the sequence of actions the external environment might observe during α. We say that β is a *behavior* of A if β is the behavior of an execution of A. We denote the set of behaviors of A by *behs*(A) and the set of finite behaviors of A by *finbehs*(A).

We remark that since the same action may occur several times in an execution or a schedule, it is sometimes convenient to distinguish the different occurrences. On these occasions we refer to a particular occurrence of an action as an *event*.

We will be illustrating many of our definitions using simple examples of candy machines and their customers. We hope that, since this class of examples is so popular in the CSP literature, they will provide an interesting comparison of the models. In the remainder of this section, we define automata modeling these candy machines and customers.

Our three candy machines CM-1, CM-2, and CM-3 differ only in their transition relations. We begin with the definition of CM-1. This candy machine has the following action signature.

Input actions:	PUSH1, PUSH2
Output actions:	SKYBAR, HEATHBAR, ALMONDJOY
Internal actions:	none

We will sometimes abbreviate the two push actions as 1 and 2, and the three dispensation actions as S, H and A. The partition *part*(CM-1) places all three output actions S, H, and A in the same equivalence class. The state of CM-1 consists of one variable 'button_pushed,' which takes on values 0, 1 and 2. In the initial state, 'button_pushed' is set to 0. We describe the transition relation for CM-1 by giving a *precondition* and an *effect* for every action π : the triple (s', π, s) is a step of CM-1 exactly if the precondition of π is satisfied by s' and s is the result of transforming s' as determined by the effects of π . We omit the precondition for an action when this precondition is *true*. The transition relation for CM-1 is as follows:

PUSH1		
	Effect:	button_pushed $\leftarrow$ 1
PUSH2		
	Effect:	button_pushed $\leftarrow$ 2
SKYBAR		
	Precondition:	button_pushed = 1
	Effect:	button_pushed $\leftarrow$ 0
HEATHBAR		
	Precondition:	button_pushed = 2
	Effect:	button_pushed $\leftarrow$ 0
ALMONDJOY		
	Precondition:	button_pushed = 2
	Effect:	button_pushed $\leftarrow$ 0

When the customer pushes button 1, CM-1 can dispense a SKYBAR. When the customer pushes button 2, CM-1 can dispense either a HEATHBAR or an ALMONDJOY, but not both. The choice between H and A is made nondeterministically by CM-1.

Candy machine CM-2 is identical to CM-1, except that its HEATHBAR action has 'false' as its precondition. This candy machine never dispenses HEATHBARs, but is able to dispense SKYBARs and ALMONDJOYs.

Candy machine CM-3 is identical to CM-1 except that all three candy dispensation actions have 'false' as their precondition. It never dispenses candy, which must disappoint a number of its customers.

Like our candy machines, our three customers CUST-1, CUST-2, and CUST-3 are also quite similar. Customer CUST-1 continues to request candy bars ad infinitum, nondeterministically choosing which button to push. Its action signature is the 'complement' of the candy machines':

Input actions:	SKYBAR, HEATHBAR, ALMONDJOY
Output actions:	PUSH1, PUSH2
Internal actions:	none

The state of CUST-1 consists of one variable 'waiting', which takes on values 'yes' and 'no'. In the initial state, 'waiting' is set to 'no'. CUST-1's actions are as follows.

SKYBAR		
	Effect:	waiting $\leftarrow$ no
HEATHBAR		
	Effect:	waiting $\leftarrow$ no
ALMONDJOY		
	Effect:	waiting $\leftarrow$ no
PUSH1		
	Precondition:	waiting = no
	Effect:	waiting $\leftarrow$ yes
PUSH2		
	Precondition:	waiting = no
	Effect:	waiting $\leftarrow$ yes

This customer is very patient: after pushing a button, it waits for a candy bar before pushing a button a second time. The partition *part*(CUST-1) of this customer's locally-controlled actions puts PUSH1 and PUSH2 together in one equivalence class.

Customer CUST-2 is somewhat more selective than CUST-1. It pushes button 2 repeatedly just until the machine dispenses a HEATHBAR, and then pushes button 1 forever. Formally, CUST-2 has another variable 'heathbar_received' in its state in addition to 'waiting'. This variable takes on values 'yes' and 'no', initially 'no'. The actions of CUST-2 that differ from those of CUST-1 are as follows:

HEATHBAR		
	Effect:	waiting $\leftarrow$ no; heathbar_received $\leftarrow$ yes
PUSH1		
	Precondition:	waiting = no; heathbar_received = yes
	Effect:	waiting $\leftarrow$ yes
PUSH2		
	Precondition:	waiting = no; heathbar_received = no
	Effect:	waiting $\leftarrow$ yes

Customer CUST-3 is similar to CUST-1 except that it may make a transition to a 'satiated' state from which it no longer requests any candy bars. Formally, CUST-3's state has an additional 'satiated' variable besides the 'waiting' variable of CUST-1. It takes on values 'yes' or 'no', initially 'no'. CUST-3 has an additional internal action BECOME_SATIATED, defined as follows.

BECOME_SATIATED		
	Precondition:	satiated = no; waiting = no
	Effect:	satiated $\leftarrow$ yes

Also, each of PUSH1 and PUSH2 has the additional precondition 'satiated = no'. Again, *part*(CUST-3) puts all three locally-controlled actions PUSH1, PUSH2, and BECOME_SATIATED in the same equivalence class.

3.2 Composition

We can construct an automaton modeling a complex system by composing automata modeling the simpler system components. The essence of this composition is quite simple: when we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π , all automata having π as an action perform π simultaneously (automata not having π as an action do nothing). For example, in the composition of CM-1 and CUST-1, we identify the output action PUSH1 of the customer with the input action PUSH1 of the candy machine. The occurrence of PUSH1 causes both the candy machine and the customer to perform PUSH1, causing button_pushed to be set to 1 in the candy machine's local state, and waiting to be set to 'yes' in the customer's local state. This synchronization models a form of communication from the customer to the candy machine.

We impose certain restrictions on the composition of automata. Since internal actions of an automaton A are intended to be unobservable by any other automaton B, we cannot allow A to be composed with B unless the internal actions of A are disjoint from the actions of B, since otherwise one of A's internal actions could force B to take a step. Furthermore, in keeping with our philosophy that at most one system component controls the performance of any given action, we cannot allow A and B to be composed unless the output actions of A and B form disjoint sets. Finally, since we do not preclude the possibility of composing a countable collection of automata, each action of a composition must be an action of only finitely many of the composition's components. One motivation for this restriction is Milner's motivation for ruling out infinite products in CCS [M]: if each automaton in an infinite product has π as an action, then an infinite amount of work is performed by a single action π , which we consider unreasonable. Since we do not have a recursion operation as CCS does, however, we require infinite products in order to model systems that can create processes dynamically.

Since the action signature of a composition (the composition's interface with its environment) is determined uniquely by the action signatures of its components, it is convenient to define a composition of action signatures before defining the composition of automata. The preceding discussion motivates the following definition. A countable collection $\{S_i\}_{i \in I}$ of action signatures is said to be *strongly compatible*[2] if for all $i, j \in I$ satisfying $i \neq j$ we have

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

[2] Such a collection is said to be *compatible* if it satisfies the first two of the three properties listed. Some of the results below follow simply from compatibility, while others require strong compatibility. Here, we simplify matters by considering the stronger definition only. The consequences of the two definitions are described more carefully in [LT1].

3. no action is contained in infinitely many sets $acts(S_i)$.

We say that a collection of automata are *strongly compatible* if their action signatures are strongly compatible. CM-1 and CUST-1, for example, are strongly compatible.

When we compose a collection of automata, internal actions of the components become internal actions of the composition, output actions become output actions, and all other actions (each of which can only an input action of a component) become input actions. For example, all actions become output actions in the composition of CM-1 and CUST-1. Notice that this composition does not hide actions such as PUSH1 representing communication between components CM-1 and CUST-1 by making them internal actions of the composition CM-1·CUST-1. As motivation for this decision, consider one automaton A having π as an output actions and two automata B_1 and B_2 having π as an input action. Notice that π is essentially a broadcast from A to B_1 and B_2 in the composition $A \cdot B_1 \cdot B_2$ of the three automata. Notice, however, that if we hide communication, then the composition $(A \cdot B_1) \cdot B_2$ would not be the same as the composition $A \cdot B_1 \cdot B_2$ since π would be made internal to $A \cdot B_1$ before composing with B_2, and hence π would no longer be a broadcast to both B_1 and B_2. This is problematic if we want to reason about the system $A \cdot B_1 \cdot B_2$ in a modular way by first reasoning about $A \cdot B_1$ and then reasoning about $A \cdot B_1 \cdot B_2$. We will define another operation to hide such communication actions explicitly.

The preceding discussion motivates the following definitions. The *composition* $S = \prod_{i \in I} S_i$ of a countable collection of strongly compatible action signatures $\{S_i\}_{i \in I}$ is defined to be the action signature with

- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$,
- $out(S) = \cup_{i \in I} out(S_i)$, and
- $int(S) = \cup_{i \in I} int(S_i)$.

The *composition* $A = \prod_{i \in I} A_i$ of a countable collection of strongly compatible automata $\{A_i\}_{i \in I}$ is the automaton defined as follows:[3]

- $sig(A) = \prod_{i \in I} sig(A_i)$,
- $states(A) = \prod_{i \in I} states(A_i)$,
- $start(A) = \prod_{i \in I} start(A_i)$,
- $steps(A)$ is the set of triples $(\vec{s_1}, \pi, \vec{s_2})$ such that, for all $i \in I$, if $\pi \in acts(A_i)$ then $(\vec{s_1}[i], \pi, \vec{s_2}[i]) \in steps(A_i)$, and if $\pi \notin acts(A_i)$ then $\vec{s_1}[i] = \vec{s_2}[i]$, and

[3] Here $start(A)$ and $states(A)$ are defined in terms of the ordinary Cartesian product, while $sig(A)$ is defined in terms of the composition of actions signatures just defined. Also, we use the notation $\vec{s}[i]$ to denote the ith component of the state vector $\vec{s}$.

- $part(A) = \cup_{i \in I} part(A_i)$.

When I is the finite set $\{1, ..., n\}$, we often denote $\prod_{i \in I} A_i$ by $A_1 \cdot \cdots \cdot A_n$.

Notice that since the automata A_i are input-enabled, so is their composition. The partition of the composition's locally-controlled actions is formed by taking the union of the components' partitions (that is, each equivalence class of each component becomes an equivalence class of the composition). For example, since CM-1's partition has one class $\{S, H, A\}$ and CUST-1's partition has one class $\{1, 2\}$, the partition of CM-1·CUST-1 has two classes $\{S, H, A\}$ and $\{1, 2\}$. This corresponds to our intuition that this partition identifies the primitive components (e.g., CM-1 and CUST-1) of the system modeled by an automaton. Again, we ignore this partition until we define fair computation in the next section.

Three basic results relate the executions, schedules, and behaviors of a composition to those of the composition's components. The first says, for example, that an execution of a composition induces executions of the component automata. Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1}\ldots$ of A, let $\alpha|A_i$ be the sequence obtained by deleting $\pi_j\vec{s_j}$ when π_j is not an action of A_i and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$.

Proposition 1: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. If $\alpha \in execs(A)$ then $\alpha|A_i \in execs(A_i)$ for every $i \in I$. Moreover, the same result holds if $execs()$ is replaced by it finexecs(,) $scheds()$, *finscheds*(), $behs()$, or *finbehs*().

Certain converses of the preceding proposition are also true. The following proposition says that executions of component automata can often be pasted together to form an execution of the composition.

Proposition 2: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $acts(A)$ such that $\beta|A_i = sched(\alpha_i)$ for every $i \in I$. Then there is an execution α of A such that $\beta = sched(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$. Moreover, the same result holds when $acts()$ and $sched()$ are replaced by $ext()$ and $beh()$, respectively.

As a corollary, schedules and behaviors of component automata can also be pasted together to form schedules and behaviors of the composition.

Proposition 3: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. Let β be a sequence of actions in $acts(A)$. If $\beta|A_i \in scheds(A_i)$ for every $i \in I$, then $\beta \in scheds(A)$. Moreover, the same result holds when $acts()$ and $scheds()$ are replaced by $ext()$ and $behs()$, respectively.

As promised, we now define an operation that 'hides' actions of an automaton by converting them to internal actions. We begin with a hiding operation for action signatures: if S is an action signature and $\Sigma \subseteq acts(S)$, then $hide_{\Sigma} S = S'$ where $in(S') = in(S) - \Sigma$, $out(S') = out(S) - \Sigma$ and $int(S') = int(S) \cup \Sigma$. We now define a hiding operation for automata: if A is an automaton and $\Sigma \subseteq acts(A)$, then $hide_{\Sigma} A$ is the automaton A' obtained from A by replacing $sig(A)$ with $sig(A') = hide_{\Sigma} sig(()A)$.

3.3 Fairness

Consider CUST-4, a particularly greedy version of CUST-1 in which all actions have the precondition 'true;' that is, the customer does not wait for a candy bar before pressing a button again. One behavior of the composition CM-1·CUST-4 is the infinite sequence 1111... in which the customer repeatedly pushes button 1 without giving the candy machine a chance to dispense a candy bar. Clearly the only time the candy machine can do its job is when it is treated fairly; that is, when it is given a chance to respond to its input. For this reason, we are in general only interested in the executions of a composition in which all components are treated fairly. While what it means for a component to be treated fairly may vary from context to context, it seems that any reasonable definition should have the property that infinitely often the component is given the opportunity to perform one of its locally-controlled actions (cf. [F]). In this section we define such a notion of fairness.

As we have mentioned, the partition of an automaton's locally-controlled actions is intended to capture some of the structure of the system the automaton is modeling. Each class of actions is intended to represent the set of locally-controlled actions of some system component. Notice that the locally-controlled actions of CM-1 and CUST-4 are $\{S, H, A\}$ and $\{1, 2\}$, respectively, and that the partition of the locally-controlled actions of CM-1·CUST-4 has two equivalence classes $\{S, H, A\}$ and $\{1, 2\}$. The definition of automaton composition guarantees that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the *essential* structure of the system's primitive components.[4] In our model, therefore, being fair to each component means being fair to each equivalence class of locally-controlled actions. This motivates the following definition.

A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of *part*(A):

1. If α is finite, then no action of C is enabled in the final state of α.

2. If α is infinite, then either α contains infinitely many events from C, or α contains infinitely many occurrences of states in which no action of C is enabled.

[4] It might be argued that retaining this partition is a bad thing to do since it destroys some aspects of abstraction. Notice, however, that any reasonable definition of fairness must lead to some breakdown of abstraction since being fair means being fair to the primitive components which must somehow be modeled.

This says that a fair execution gives fair turns to each class C of *part*(A), and therefore to each component of the system being modeled. Infinitely often the automaton attempts to perform an action from the class C. On each attempt, either an action of C is performed, or no action from C *can* be performed since no action from C is enabled. For example, we may view a finite fair execution as an execution at the end of which the automaton repeatedly cycles through the classes in round-robin order attempting to perform an action from each class, but failing each time since no action is enabled from the final state. Returning to the composition CM-1·CUST-4, we see that 111... is not a fair behavior since the output action S of CM-1 is enabled in every state (except the first) and yet never performed. On the other hand, 11S11S... *is* a fair behavior of the composition since infinitely often an output action of CM-1 is performed and infinitely often an output action of CUST-4 is performed. Considering the composition CM-1·CUST-3, notice that any finite execution ending with the action BECOME_SATIATED is a fair execution since from the state following this action no action of the composition is enabled. (In fact, these are precisely the fair finite executions of this composition.)

We denote the set of fair executions of A by *fairexecs*(A). We say that β is a *fair schedule* of A if β is the schedule of a fair execution of A, and we denote the set of fair schedules of A by *fairscheds*(A). We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A, and we denote the set of fair behaviors of A by *fairbehs*(A). For example, the schedule consisting of the single internal action BECOME_SATIATED is a fair schedule of CM-1·CUST-3, and hence the empty schedule consisting of no actions is a fair behavior of this composition.

We can prove the following analogues to Propositions 1-3 in the preceding section:

Proposition 4: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. If $\alpha \in \mathit{fairexecs}(A)$ then $\alpha|A_i \in \mathit{fairexecs}(A_i)$ for every $i \in I$. Moreover, the same result holds if *fairexecs*() is replaced by *fairscheds*() or *fairbehs*().

Proposition 5: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Suppose α_i is a fair execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $\mathit{acts}(A)$ such that $\beta|A_i = \mathit{sched}(\alpha_i)$ for every $i \in I$. Then there is a fair execution α of A such that $\beta = \mathit{sched}(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$. Moreover, the same result holds when *acts*() and *sched*() are replaced by *ext*() and *beh*(), respectively.

Proposition 6: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Let β be a sequence of actions in $\mathit{acts}(A)$. If $\beta|A_i \in \mathit{fairscheds}(A_i)$ for every $i \in I$, then $\beta \in \mathit{fairscheds}(A)$. Moreover, the same result holds when *acts*() and *fairscheds*() are replaced by *ext*() and *fairbehs*(), respectively.

We state these results because analogous results often do not hold in other models. As we will see in the following section, the fact that the fair behavior of a composition is uniquely determined by the fair behavior of the components makes it possible to reason about the fair behavior of a system in a modular way. The proofs of these propositions are nearly identical to the proofs of Propositions 1-3. The one additional key fact needed is the fact that a component automaton determines by itself when one of its locally-controlled actions may be performed.

3.4 Problem Specification

We want to say that a problem specification is simply a set of allowable 'behaviors,' and that an automaton solves the specification if each of its 'behaviors' is contained in this set. The automaton solves the problem in the sense that every 'behavior' it exhibits is a 'behavior' allowed by the problem specification (but notice that there is no single 'behavior' the automaton is *required* to exhibit). The appropriate notion of 'behavior' (e.g., finite behavior, infinite behavior, fair behavior, etc.) used in such a definition depends to some extent on the nature of the problem specification.

It is often useful to differentiate between two types of specifications since different techniques are usually used to prove that such specifications are satisfied [La1]. *Safety properties* are informally characterized by the fact that they specify a property that must hold in every state of a computation. Since an infinite computation satisfies a safety property if and only if every finite prefix of the computation does so, the notion of 'behavior' most useful in this context seems to be finite behaviors. *Liveness properties* are informally characterized by the fact that they specify events that must eventually be performed. A reliable candy machine, for example, should satisfy the liveness condition that if a button is pushed, then a candy bar (of the correct type) is eventually dispensed. Clearly this is a property of infinite behaviors, and not finite behaviors. In fact, this is a property that can only be satisfied by fair behaviors, since a candy machine cannot dispense the required candy bar if it is not given the chance to do so. The notion of 'behavior' most useful in this context, therefore, seems to be fair behaviors.

Consequently, we would like to say that a specification is a set of allowable behaviors, and that an automaton solves the specification if all finite or fair behaviors (depending on the context) of the automaton are contained in the set. In addition to a set of allowable behaviors, however, a problem specification must specify the required interface between a solution and its environment. That is, we want a problem specification to be a set of behaviors together with an action signature.

We therefore define a *schedule module* H to consist of two components:

- an action signature $sig(H)$, and
- a set $scheds(H)$ of *schedules*.

Each schedule in $scheds(H)$ is a finite or infinite sequence of actions of H. We denote by *finscheds*(H) the set of finite schedules of H. The *behavior* of a schedule β of H is the subsequence of β consisting of external actions, and is denoted by $beh(\beta)$. We say that β is a *behavior* of H if β is the behavior of a schedule of H. We denote the set of behaviors of H by $behs(H)$ and the set of finite behaviors of H by *finbehs*(H). We extend the definitions of fair schedules and fair behaviors to schedule modules in a trivial way, letting $fairscheds(H) = scheds(H)$ and $fairbehs(H) = behs(H)$. We will use the term *module* to refer to either an automaton or a schedule module.

There are several natural schedule modules that we often wish to associate with an automaton. They correspond to the automaton's schedules, finite schedules, fair schedules, behaviors, finite behaviors and fair behaviors. For each automaton A, let $Scheds(A)$, $Finscheds(A)$ and $Fairscheds(A)$ be the schedule modules having action signature $sig(A)$ and having schedules $scheds(A)$, *finscheds*(A) and $fairscheds(A)$, respectively. Also, for each module M (either an automaton or schedule module), let $Behs(M)$, $Finbehs(M)$ and $Fairbehs(M)$ be the schedule modules having the external action signature $extsig(M)$ and having schedules $behs(M)$, *finbehs*(M) and $fairbehs(M)$, respectively. (Here and elsewhere, we follow the convention of denoting sets of schedules with lower case names and corresponding schedule modules with corresponding upper case names.)

It is convenient to define two operations for schedule modules. Corresponding to our composition operation for automata, we define the composition of a countable collection of strongly compatible schedule modules $\{H_i\}_{i\in I}$ to be the schedule module $H = \prod_{i\in I} H_i$ where:

- $sig(H) = \prod_{i\in I} sig(H_i)$,
- $scheds(H)$ is the set of sequences β of actions of H such that $\beta|H_i$ is a schedule of H_i for every $i \in I$.

The following proposition shows how composition of schedule modules corresponds to composition of automata.

Proposition 7: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Then $Scheds(A) = \prod_{i\in I} Scheds(A_i)$, $Fairscheds(A) = \prod_{i\in I} Fairscheds(A_i)$, $Behs(A) = \prod_{i\in I} Behs(A_i)$ and $Fairbehs(A) = \prod_{i\in I} Fairbehs(A_i)$.

Corresponding to our hiding operation for automata, we define hide $hide_\Sigma H$ to be the schedule module H' obtained from H by replacing $sig(H)$ with $sig(H') = hide_\Sigma sig(()H)$.

Finally, we are ready to define a problem specification and what it means for an automaton to satisfy a specification. A *problem* is simply a schedule module P. An automaton A *solves*[5] a problem P if A and P have the same external

[5]This concept is called *satisfying* in [LT1].

action signature and *fairbehs*$(A) \subseteq$ *fairbehs*(P). An automaton A *implements* a problem P if A and P have the same external action signature (that is, the same external interface) and *finbehs*$(A) \subseteq$ *finbehs*(P). Notice that if A solves P, then A cannot be a trivial solution of P since the fact that A is input-enabled ensures that *fairbehs*(A) contains a response by A to every possible sequence of input actions. For analogous reasons, the same is true if A implements P.

Since we may want to carry out correctness proofs hierarchically in several stages, it is convenient to state the definitions of 'solves' and 'implements' more generally. For example, we may want to prove that one automaton solves a problem by showing that the automaton 'solves' another automaton, which in turn 'solves' another automaton, and so on, until some final automaton solves the original problem. Therefore, let M and M' be modules (either automata or schedule modules) with the same external action signature. We say that M *solves* M' if *fairbehs*$(M) \subseteq$ *fairbehs*(M') and that M *implements* M' if *finbehs*$(M) \subseteq$ *finbehs*(M').

To illustrate these definitions, let us consider some interesting specifications of correct candy machine behavior.

Some basic requirements for a candy machine can be described by the schedule module SAFE-CM. SAFE-CM has the same action signature as CM-1, and has as its set of schedules the set of sequences over the symbols 1,2,S,H,A satisfying the following condition: every S is immediately preceded by a 1, and every A or H is immediately preceded by a 2.

In order to show that CM-1 is a safe candy machine (that is, that it implements the problem described by SAFE-CM), we must show that all finite behaviors of CM-1 satisfy the given requirement. We proceed by induction on the length of a behavior, using an inductive hypothesis that characterizes the state of CM-1 in terms of the preceding events: button_pushed = 1 if the last event in the sequence is PUSH1, button_pushed = 2 if the last event in the sequence is PUSH2, and button_pushed = 0 otherwise (that is, if the sequence is empty or the last event is a dispensation event). The inductive step considers cases based on the five possible actions. For instance, if SKYBAR occurs, its precondition implies that button_pushed = 1 just prior to the dispensation; thus, the immediately preceding symbol in the sequence is 1, as needed. The other cases are similar. It follows that CM-1 implements SAFE-CM, and hence that CM-1 is a safe candy machine. In fact, the same proof also shows that CM-1 solves SAFE-CM.

It is also easy to see that CM-2 is a safe candy machine. However, saying that CM-1 and CM-2 are safe candy machines is not really saying enough, since the same is also true for CM-3. CM-3's finite behaviors are just the finite sequences of 1's and 2's, which trivially satisfy the required condition. Although CM-3 is a safe candy machine, it is not a very interesting one. Therefore, we give a stronger specification below. In order to do this, we need an additional definition.

Since an automaton cannot block input actions, in discussing correct candy machine behavior it is helpful to consider certain 'well-formedness' conditions on the interaction between the machine and its environment. For example, we may want to restrict attention to interactions in which push and dispensation events alternate strictly. Define a sequence of candy machine actions to be *well-formed* if it consists of alternating input and output (push and dispensation) actions, starting with an input action. Notice that CM-1 has behaviors, in fact fair behaviors, that are not well-formed. For example, 11S11S... is a non-well-formed fair behavior of CM-1. This is because CM-1 does not have the power to insure that its environment satisfies the well-formedness condition.

A stronger set of requirements than SAFE-CM can be described by the schedule module LIVE-CM. LIVE-CM has the same action signature as CM-1. Its set of sequences are those that are safe candy machine sequences and that in addition satisfy the following condition: 'If the sequence is well-formed, then every 1 event is followed by a later S event, and every 2 event is followed by a later H or A event.'[6] That is, every request for a candy bar is eventually satisfied by a candy bar of the correct type.

Let us consider which of our candy machines are live candy machines; that is, which candy machines solve LIVE-CM. CM-3 is not a live candy machine because it has fair behaviors, such as the sequence consisting of the single event 1, that do not satisfy this condition. (This sequence satisfies the well-formedness hypothesis, but does not satisfy the liveness conclusion.) On the other hand, CM-1 is a live candy machine, which we can prove as follows. Suppose not; then there is a fair behavior of CM-1 that is well-formed and that contains a push event that is not followed by any later dispensation event of the correct type. By well-formedness and the fact that CM-1 is a safe candy machine, the only possibility is that the sequence is finite and ends with the given push event. Say, for example, that the push event is PUSH1. Then by the state characterization given above, the state after the given schedule has button_pushed = 1. Then the SKYBAR dispensation action is enabled in this state. But the definition of a fair execution implies that no action of CM-1 can be enabled in the final state, which yields a contradiction. CM-2 is also a live candy machine, even though it has less nondeterminism than CM-1. The proof is similar to that for CM-1.

Notice that while CM-1 and CM-2 both solve LIVE-CM, neither implements LIVE-CM since there are finite behaviors of both machines ending with the push of a button that are not contained in LIVE-CM. Conversely, while it can be shown that CUST-3 implements CUST-1, CUST-3 does not solve CUST-1 since there are fair behaviors of CUST-3, such as the empty sequence, that are not fair behaviors of CUST-1. In general, given an automaton A and a problem P, it is not the case that if A solves P then A implements P, nor is it the case that if A implements P then A solves P.

One might ask the technical question whether it might be reasonable to eliminate the well-formedness hypothesis in the live candy machine behavior

[6] This can be formalized in terms of temporal logic.

specification. If we did this, then we would arrive at a stronger specification for a live candy machine, one that requires that the machine must always issue candy sometime after each push, regardless of whether the pushes happen in a well-formed manner. While this might be a reasonable requirement for a candy machine, CM-1 does not satisfy it. For consider the (non-well-formed) behavior 12H12H12H... of CM-1. This contains 1 events that are not followed by S events. However, it is a fair behavior of CM-1 since infinitely often an action from the single class $\{S, A, H\}$ of $part(CM-1)$ is performed. Consequently, CM-1 does not satisfy the proposed stronger specification.

As we have seen, there are many ways to argue that an automaton A solves a problem P. We now turn our attention to two more general techniques.

3.4.1 Proof Techniques: Modular Decomposition

One common technique for reasoning about the behavior of an automaton is *modular decomposition*, in which we reason about the behavior of a composition by reasoning about the behavior of the component automata individually.

It is often the case that an automaton behaves correctly only in the context of certain restrictions on its input. These restrictions may be guaranteed in the context of the composition with other automata comprising the remainder of the system, or may be restrictions defined by a problem statement describing conditions under which a solution is required to behave correctly. (Recall, for example, the well-formedness conditions defined earlier for candy machines.) A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors: as long as the environment does not violate this property, neither does the module.

In practice, this notion is of most interest when the property is prefix-closed, and when the property does not concern the module's internal actions. A set of sequences $\mathcal{P}$ is said to be *prefix-closed* if $\beta \in \mathcal{P}$ whenever both β is a prefix of α and $\alpha \in \mathcal{P}$. For example, the set of well-formed sequences defined for candy machines is prefix-closed. A module M (either an automaton or schedule module) is said to be *prefix-closed* provided that $finbehs(M)$ is prefix-closed. For example, the schedule module SAFE-CM is prefix-closed, and every automaton is prefix-closed. Let M be a prefix-closed module and let $\mathcal{P}$ be a nonempty, prefix-closed set of sequences of actions from a set Φ satisfying $\Phi \cap int(M) = \emptyset$. We say that M *preserves* $\mathcal{P}$ if $\beta\pi|\Phi \in \mathcal{P}$ whenever $\beta|\Phi \in \mathcal{P}$, $\pi \in out(M)$, and $\beta\pi|M \in finbehs(M)$.

It is not hard to see, for example, that in this sense the candy machine CM-1 preserves well-formedness: although the customer may press a button twice without waiting for a candy bar to be dispensed, the candy machine dispenses a candy bar only if a button has been pressed since the last candy bar was dispensed. In general, if a module preserves a property $\mathcal{P}$, then the module is not the first to violate $\mathcal{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathcal{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathcal{P}$. This definition, however, deserves closer inspection. First, notice that we consider only sequences β with the property that $\beta\pi|M \in \mathit{finbehs}(M)$. This implies that we consider only sequences β that contain no internal actions of M. Second, notice that we require sequences β to satisfy only $\beta|\Phi \in \mathcal{P}$ rather than the stronger property $\beta \in \mathcal{P}$. Suppose, for example, that $\mathcal{P}$ is a property of the actions Φ at one of two interfaces to the module M. In this case, it may be that for no $\beta \in \mathcal{P}$ and $\pi \in \mathit{out}(M)$ is it the case that $\beta\pi|M \in \mathit{finbehs}(M)$, since all finite behaviors of M containing outputs include activity at both interfaces to M. By considering β satisfying only $\beta|\Phi \in \mathcal{P}$, we consider all sequences determining finite behaviors of M that, at the interface concerning $\mathcal{P}$, do not violate the property $\mathcal{P}$.

One can prove that a composition preserves a property by showing that each of the component automata preserves the property:

Proposition 8: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. If A_i preserves $\mathcal{P}$ for every $i \in I$, then A preserves $\mathcal{P}$.

For example, since CM-1 and CUST-1 both preserve well-formedness, the composition CM-1·CUST-1 does so as well.

In fact, we can prove a slightly stronger result. An automaton is said to be *closed* if it has no input actions. In other words, it models a closed system that does not interact with its environment.

Proposition 9: Let A be a closed automaton. Let $\mathcal{P}$ be a set of sequences over Φ. If A preserves $\mathcal{P}$, then $\mathit{finbehs}(A)|\Phi \subseteq \mathcal{P}$.

In the special case that Φ is the set of external actions of A, the conclusion of this proposition reduces to the fact that $\mathit{finbehs}(A) \subseteq \mathcal{P}$. The proof of the proposition depends on the fact that Φ does not contain any of A's input actions, and hence that if the property $\mathcal{P}$ is violated then it is not an input action of A committing the violation. In fact, this proposition follows as a corollary from the following slightly more general statement: If A preserves $\mathcal{P}$ and $\mathit{in}(A) \cap \Phi = \emptyset$, then $\mathit{finbehs}(A)|\Phi \subseteq \mathcal{P}$.

Combining Propositions 8 and 9, we have the following technique for proving that an automaton implements a problem:

Corollary 10: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata with the property that $A = \prod_{i\in I} A_i$ is a closed automaton. Let P be a problem with the external action signature of A. If A_i preserves $\mathit{finbehs}(P)$ for all $i \in I$, then A implements P.

That is, if we can prove that each component A_i preserves the external behavior required by the problem P, then we will have shown that the composition A preserves the desired external behavior; and since A has no input actions that could be responsible for violating the behavior required by P, it follows that all finite behaviors of A are behaviors of P.

A similar technique follows from the following proposition:

Proposition 11: Let $\{A_i\}_{i\in I}$ be a collection of strongly compatible automata, and let $\{P_i\}_{i\in I}$ be a collection of problems. If A_i solves P_i for every i, then $\prod_{i\in I} A_i$ solves $\prod_{i\in I} P_i$.

This says we can prove that the composition of the automata $\{A_i\}_{i\in I}$ solves a problem by proving that each component A_i solves a problem P_i and then proving that the composition of the problems $\{P_i\}_{i\in I}$ solves the original problem. For example, consider proving that every fair behavior of the composition of CM-1 and CUST-1 is an infinite well-formed sequence of actions in which each dispensation action dispenses an appropriate candy. Let LIVE-CUST be the schedule module whose signature is the same as CUST-1's, and whose schedules are exactly those in which (i) the customer is not the first to violate well-formedness, and (ii) if the sequence is well-formed, then it is either infinite or else finite and ending with a push event. Then it is easy to see that CUST-1 solves LIVE-CUST. We have already argued that CM-1 solves the schedule module LIVE-CM described earlier. So it suffices to prove that every behavior of the composition of LIVE-CUST and LIVE-CM is an infinite well-formed sequence of actions in which each dispensation action dispenses an appropriate candy. This is not difficult to show: well-formedness holds because neither component is the first to violate it, appropriate responses follow from the specification of LIVE-CM, and the sequence is infinite because neither component stops at its own turn.

3.4.2 Proof Techniques: Hierarchical Decomposition

A second common technique for proving that an automaton solves a problem is *hierarchical decomposition* in which we prove that the given automaton solves a second, that the second solves a third, and so on until the final automaton solves the given problem. One way of proving that one automaton A solves another automaton B is to establish a relationship between the states of A and B and use this relationship to argue that the fair behaviors of A are fair behaviors of B. One helpful such relationship is a possibilities mapping, which we now define.

We define an *extended step* of an automaton A to be a triple of the form (s', β, s), where s' and s are states of A, β is a finite sequence of actions of A, and there is an execution fragment of A having s' as its first state, s as its last state, and β as its schedule. (This execution fragment might consist of only a single state, in the case that β is the empty sequence.) Suppose A and B are automata with the same external action signature, and suppose f is a mapping from $states(A)$ to the power set of $states(B)$. That is, if s is a state of A then $f(s)$ is a set of states of B. The mapping f is said to be a *possibilities mapping* from A to B provided the following conditions hold:

1. For every start state s_0 of A, there is a start state t_0 of B such that $t_0 \in f(s_0)$.

2. If s' is a reachable state of A, $t' \in f(s')$ is a reachable state of B, and (s', π, s) is a step of A, then there is an extended step (t', γ, t) of B such that

   (a) $\gamma | ext(B) = \pi | ext(A)$, and

   (b) $t \in f(s)$.

It is easy to show, for example, that there is a possibilities mapping f from CUST-2 to CUST-1 that maps each state s of CUST-2 to the singleton set containing the state of CUST-1 that only contains the 'waiting' variable of s.

The existence of a possibilities mapping from A to B, together with additional results relating the fair behaviors of A and B, can be used to prove that A solves B. Some such additional results are given in [LT1] and [WLL]. For example, using our possibilities mapping from CUST-2 to CUST-1 we can prove that CUST-2 actually solves CUST-1. A straightforward proof can be based directly on the definition of fair execution and the fact that for every state s of CUST-2, some output action is enabled in s for CUST-2 exactly if some output action is enabled in the single state in $f(s)$ for CUST-1.

In cases in which we are only interested in finite behaviors and not fair behaviors, the following simple result is often useful.

Proposition 12: Suppose that A and B are automata with the same external action signature. If there is a possibilities mapping from A to B, then A implements B.

So, for example, the existence of the possibilities mapping f from CUST-2 to CUST-1 implies that CUST-2 implements CUST-1.

4 Choosing a Ring Leader

In this section we sketch a more sophisticated example than the candy machines studied in the previous section, the election of a leader in a ring of processors. This example exhibits much more interesting concurrent activity than the candy machine example. It shows how one can use the model to reason about interesting concurrent algorithms, and suggests how the model can be used to carry out complexity analysis and prove lower bound and impossibility results.

We assume a ring of n processors, each starting with a unique identifier chosen from a universal totally ordered identifier set I. Each processor can communicate with each of its neighbors in the ring, using a pair of one-way channels. The processors do not know the size of the ring, nor the specific subset of I that is actually being used as identifiers. The object is for the processors to choose a unique leader from among themselves. This problem has been widely studied in the area of distributed algorithms.

Each processor and each communication channel is modeled as an I/O automaton. Each channel automaton has input actions of the form SEND(M) and output actions of the form RECEIVE(M).[7] Its state is a multiset, consisting of those messages that have been sent but not yet received; initially, the multiset is empty. The transition relation is as follows:

SEND(M)		
	Effect:	messages $\leftarrow$ messages $\cup$ $\{M\}$
RECEIVE(M)		
	Precondition:	M $\in$ messages
	Effect:	messages $\leftarrow$ messages - $\{M\}$

The partition puts all output actions (all RECEIVE actions) in the same equivalence class; this has the effect of hypothesizing that if there is a message to be delivered, then some message is eventually delivered.

Each processor is also modeled as an I/O automaton, having SEND output actions and RECEIVE input actions. In addition, it has a LEADER output action by which it can announce that it has been chosen as the leader processor. It may also have internal actions.

A collection of channel and processor automata is composed into a single system automaton, and then the hiding operator is used to produce a new system automaton in which the only external actions are LEADER actions. The problem to be solved by the system can be described by the schedule module whose external action signature has no input actions and only LEADER output actions, and whose set of schedules consists of the set of sequences of length exactly 1. That is, in a correct behavior, exactly one LEADER event occurs.

We now describe a particular algorithm for solving this problem, based on that of LeLann [Le] and Chang and Roberts [CR]. Each processor sends its identifier clockwise around the ring. When a processor receives an identifier, if the identifier is less than its own, the processor discards the received identifier. If it is greater than its own, the processor passes the received identifier clockwise. If it is equal to its own, the processor performs a LEADER output action.

In more detail, the state of a processor with identifier i has a variable 'pending' which holds a subset of I, initially $\{i\}$. It also has a variable 'leader-status', which takes on values from {'unknown', 'elected', 'announced'} and has initial value 'unknown'. The actions are defined as follows:

[7]Since the model uses a global naming scheme, the actual action names would have to be subscripted with information identifying the particular channel.

RECEIVE(j), $j \in I$		
	Effect:	if $j > i$ then pending $\leftarrow$ pending $\cup \{j\}$ if $j = i$ then leader-status $\leftarrow$ 'elected'
SEND(j), $j \in I$		
	Precondition:	$j \in$ pending
	Effect:	pending $\leftarrow$ pending - $\{j\}$
LEADER		
	Precondition:	leader-status = 'elected'
	Effect:	leader-status $\leftarrow$ 'announced'

The partition puts all output actions in the same equivalence class. It is not hard to carry out a correctness proof of this algorithm using the model. The safety proof (that is, that no more than one LEADER event ever occurs) involves proving an invariant assertion relating the identifiers that appear in different places in the ring, both as processor id's and in messages. More specifically, it must be shown that if $i < j$, then a processor with identifier i, a processor with identifier j, and a message containing identifier i cannot appear in that order, reading clockwise around the ring.

In order to prove liveness (that is, that some LEADER event eventually occurs), another invariant is used, expressing conservation of the message corresponding to the maximum identifier. Then a 'variant function' is defined, describing the progress that has been made toward election of a leader: for each state, the value of the variant function in that state is the sum of the distances of all id's back to their originating processors, measured in a clockwise direction. At every point where the value of the variant function is nonzero, any action that occurs (other than the LEADER action) can be shown to decrease its value. Furthermore, at every point where the value of the variant function is nonzero, some action is enabled. Thus, the function value eventually reaches zero, and hence a LEADER event eventually occurs.

The model can be used to carry out complexity analysis. For any execution of the algorithm, the number of SEND or RECEIVE events can be used as a measure of the amount of communication; it is not hard to prove that $2n^2$ is a worst-case upper bound on this number. Also, for any execution, time can be measured as follows. Assign a 'real time' to each event, as large as possible, subject to the requirement that for each class of the partition, the time between successive 'turns' for that class is at most 1. Then the difference between the real time assigned to the LEADER event and the start time can be taken as a time measure for the entire execution. Since $2n^2$ is a worst-case upper bound on the number of SEND and RECEIVE events, it is not hard to see that $2n^2+1$ is a worst-case upper bound for this time measure.[8]

The given algorithm is not optimal in its communication requirements; for

[8] The standard analysis of this algorithm attains an $O(n)$ upper bound, by assuming all messages are delivered within time 1 regardless of the congestion of the message channels. We do not assume this, and so obtain a quadratic bound.

example, [P] contains an algorithm with an $O(n \log n)$ upper bound. The algorithm in [P] can also be formalized and analyzed using our model. Also, [Bu] proves an $\Omega(nlogn)$ lower bound on the worst-case amount of communication; this result also is describable in our model.

5 Other Applications

The model has been used to describe and reason about many different kinds of algorithms, both in systems applications and in the algorithms research literature. In this section, we describe some of these uses.

5.1 Network Resource Allocation

Our first use of the model was for describing network resource allocation algorithms. [LT1] presents a network arbiter design and proves its correctness using I/O automata. The algorithm is based on a resource performing a treewalk of a spanning tree of the network graph. The conditions proved include safety properties (mutual exclusion) and liveness properties (no lockout).

The correctness proof is done in three levels of abstraction. The problem definition is presented as a high-level schedule module, in which inputs are requests and returns and outputs are grants, all for a particular resource. The intermediate level is a description of the algorithm in terms of graph theory, formalized as an automaton together with a restricted set of executions. Finally, the complete distributed algorithm is described as a composition of automata at the lowest level. It is shown that each level solves the level above it, and thus that the distributed algorithm solves the arbiter problem.

Most of the interesting reasoning about the algorithm is done at the intermediate level, in terms of graphs. This reasoning is close to the intuitive reasoning one would normally use to understand and explain the algorithm. The interesting work involves showing that the intermediate level solves the high-level problem statement. Showing that the lowest level solves the intermediate level is a long but straightforward case analysis.

[LT1] also contains an analysis of the time complexity of the algorithm, demonstrating an $O(n)$ worst-case upper bound, where n is the number of nodes in the network, and an $O(d)$ worst-case upper bound when requests do not overlap, where d is the diameter of the network. The time analysis proof follows the proof of 'no lockout' very closely, suggesting that there may be a more general correspondence between liveness proofs and proofs of upper bounds on time.

We have also used the model to study other network resource allocation algorithms. For example, in [LW], we give an algorithm for the 'Drinking Philosophers' problem: in this problem, users request sets of resources by name, with the same user possibly requesting different sets of resources each time it makes a request. [CM2] contains an algorithm for this problem, constructed by modifying a particular Dining Philosophers algorithm. Our algorithm, based on the one in [CM2], is described as a composition of automata that solve the Dining Philosophers problem and automata that carry out additional bookkeeping. Our use of composition allows us to use any Dining Philosophers algorithm as a 'subroutine;' some choices can be shown to yield better time performance for the resulting Drinking Philosophers algorithm than is yielded by the algorithm of [CM2].

5.2 Synchronizers

In [A], Awerbuch describes a *synchronizer* algorithm — a distributed algorithm designed to convert programs written for synchronous networks into versions that can be used in asynchronous networks. In this algorithm, the network nodes are partitioned into *clusters*, and different strategies are used to synchronize within clusters and among clusters. The algorithm is clever, but complex, and is presented without formal proof. In [FLS], we provide a new presentation and a proof for Awerbuch's algorithm. The algorithm is decomposed into separate automata for intercluster and intracluster synchronization. The intercluster synchronizer is further decomposed into a piece executing at each node. In fact, Awerbuch's actual program for each node is described as the composition of two automata, one participating in intercluster and one in intracluster synchronization.

5.3 Communication

In [WLL], we present a correctness proof for the intricate distributed minimum spanning tree algorithm of [GHS]. The techniques used are based on the hierarchical structure used in [LT1]. However, instead of a linear hierarchy of algorithms, we use a *lattice* of algorithms. The complete algorithm has several different projections onto higher level 'subalgorithms', where each subalgorithm represents one task performed by the main algorithm. The proof involves showing that the subalgorithms all solve the minimum spanning tree problem and that the full algorithm 'solves' all of the subalgorithms. In showing the latter, we make use of many properties of the separate subalgorithms. We develop the basic theory needed for lattice-structured proofs; some work on a similar theory appears in [LaS].

Another proof of the correctness of the algorithm of [GHS] appears in [CG]. This proof uses techniques closely related to the notion of communication-closed layers [EF], and is based on a model which is essentially the same as the I/O automaton model.

More recently, we have used I/O automata to characterize correct behavior for physical channels and data links [LMF]. We prove that certain types of data link behavior can be implemented in terms of certain types of physical channels, while other types cannot. Preliminary results show that interesting data link behavior seems to require at least some stable storage (whereas previous work shows that a single stable bit at each end suffices). Also, under certain technical assumptions, the data link protocol must use unbounded size headers to achieve reasonable behavior, in case the underlying physical channels are not FIFO.

5.4 Concurrency Control

We have been using the model as the formal foundation for a new theory of *atomic transactions*. Transactions arose originally in database systems, but are now used as a basic construct for general data-oriented distributed programming. Use of transactions in general-purpose languages has required their extension to allow nesting; nested transactions permit more concurrency than single-level transactions, and permit localized handling of failures.

In [LM], we use I/O automata to model nested transactions, state the correctness conditions that they must satisfy, describe an exclusive locking algorithm for nested transactions, and carry out a correctness proof. In later papers, we extend this treatment to more general locking algorithms and timestamp-based algorithms. We also prove correctness of algorithms for management of 'orphan' transactions — transactions that continue to execute even though some ancestor in the transaction nesting structure has been aborted. We are able to use I/O automata to decompose the orphan algorithms so that concurrency control and recovery are handled by one module, and orphan management is handled by another. Correctness properties for the two kinds of modules are proved separately, and then combined to yield correctness properties for the complete algorithm.

We have had similar success in describing correctness of algorithms for replicated data management. We are able to decompose certain replicated data algorithms into modules that handle concurrency control and recovery at the level of individual data replicas and modules that implement the data replication algorithm. A book [LMWF] is now in progress, describing this theory.

Although the model has proved to be a very usable tool for describing these results, its full power has not yet been used in this work. In particular, only finite executions have so far been considered, and only safety properties have been proved.

5.5 Shared Atomic Objects

A topic of recent research interest has been the study of wait-free implementability of concurrently-accessible atomic objects in terms of other atomic objects. An object is said to be *atomic*, roughly speaking, if it responds to concurrent invocations of operations as if the operations were executed indivisibly at some time between the invocation and response times. So far, most of the work has focused on read-write registers for use by various numbers of readers and writers. Many of the algorithms are very complex and difficult to understand precisely.

The paper [La2], which initiated this research area, contains an interesting formal model based on partial orderings of operations. However, most of the subsequent papers do not use Lamport's model, but instead include their own models and definitions. The multiplicity of models has contributed to making the papers very difficult to read.

In [Bl], Bloom uses the I/O automaton model as the basis for stating correctness conditions for atomic read-write registers, for describing a new algorithm (which implements 2-writer n-reader registers from 1-writer n+1-reader registers) and for proving the algorithm correct. He describes the solution as a composition of automata for each of the reader and writer protocols and automata for the 1-writer registers used in the implementation. The combination is shown to implement the desired 2-writer register. The work is rigorous and clear; we hope that a similar presentation will help clarify some of the other algorithms.

New work by Schaffer [S] uses the I/O automaton model to point out errors in a published register algorithm, modify the algorithm, and prove the correctness of the modified algorithm. New work by Herlihy on impossibility results for atomic object implementations [He] also uses the I/O automaton model.

5.6 Dataflow

In [LS], we formulate the semantics of dataflow networks in terms of I/O automata. We define the notion of 'determinacy' (that is, that the sequence of output actions is uniquely defined by the sequence of input actions), a notion that is considered important in dataflow computation. We state a theorem that expresses Kahn's main result about dataflow networks [K] — that the semantics of networks of determinate components can be uniquely defined using the least fixed point operator applied to certain equations involving behavior of the individual components. We then prove a theorem showing the equivalence of our operational semantics and Kahn's fixed-point semantics. In fact, the work of [LS] generalizes Kahn's work since the determinate I/O automata used in [LS] to model processes compute all continuous stream functions whereas Kahn's processes compute a more restricted class of functions.

5.7 Real-Time Computing

Finally, some recent work [TMM] suggests some ways in which time can be introduced into the I/O automaton model. Based on these definitions, Lynch has suggested [Ly] some preliminary ideas on how the I/O automaton model can be used to model and reason about real-time computing.

References

[A] Awerbuch, B. Complexity of Network Synchronization. JACM 32(4), October, 1985, pp. 804-823.

[Bl] Bloom, B. Constructing Two-Writer Atomic Registers. *6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Vancouver, British Columbia, Canada, August, 1987, pp. 249-259. Also, to appear in Special Issue of IEEE Transactions on Computing, on Parallel and Distributed Algorithms.

[Bu] Burns, J. A Formal Model for Message Passing Systems. Technical Report TR91, Indiana University, May, 1980.

[CM1] Chandy, K.M., and Misra, J. A Foundation of Parallel Program Design. Addison-Wesley, 1988.

[CM2] Chandy, K.M., and Misra, J. The Drinking Philosophers Problem. ACM-TOPLAS 6(4), October, 1981, pp. 632-646.

[CR] Chang, E. and Roberts, R. An Improved algorithm for decentralized extrema-finding in circular configurations of processes. CACM 22, May, 1979, pp.281-283.

[CG] Chou, C.-T. and Gafni, E. Understanding and Verifying Distributed Algorithms Using Stratified Decompositions. In *Proceedings of 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Toronto, Ontario, Canada, August, 1988, pp. 44-65.

[EF] Elrad, T. and Francez, N. Decomposition of Distributed Programs into Communication-Closed Layers. *Science of Computer Programming 2(3),* 1982, pp. 155-173.

[FLS] Fekete, A., Lynch, N., and Shrira, L. A Modular Proof of Correctness for a Network Synchronizer. *2nd International Workshop on Distributed Algorithms,* Amsterdam, The Netherlands, July,1987.

[F] Francez, N. Fairness. Springer-Verlag, Berlin, 1986.

[GHS] Gallager, R., Humblet, P. and Spira, P. A Distributed Algorithm for Minimum-Weight Spanning Trees TOPLAS, Vol. 5, No. 1 (January, 1983), pp. 66-77.

[GL] Goldman, K.J., and Lynch, N.A. Quorum Comsensus in Nested Transaction Systems. *6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,*Vancouver, British Columbia, Canada, August, 1987.

[He] Herlihy, M. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Toronto, Ontario, Canada, August, 1988, pp. 276-290.

[Ho] Hoare, C. A. R. Communicating Sequential Processes. Prentice-Hall, 1985.

[K] Kahn, G. The Semantics of a Simple Language For Parallel Programming. *Information Processing 74.* North-Holland Publishing Co., 1974.

[La1] Lamport, L. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering SE-3(2)*, 1977, pp. 125-143.

[La2] Lamport, L. On Interprocess Communication, Parts I and II. *Distributed Computing 1(2),* 1986, pp. 77-101.

[LaS] Lam, S. and Shankar, U. Protocol Verification via Projections. *IEEE Trans. on Software Engineering SE10(4).* July, 1984.

[Le] LeLann, G. Distibuted Systems, Towards a Formal Approach. *IFIP Congress,* Toronto, 1977, pp. 155-160.

[Ly] Lynch, N.A. Modeling Real-Time Systems. ONR Workshop on Real-Time Computing, Alexandria, Virginia, November, 1988.

[LF] Lynch, N.A., and Fischer, M.J. On Describing the Behavior and Implementation of Distributed Systems. *Theoretical Computer Science 13*, 1981, pp. 17-43.

[LM] Lynch, N.A., and Merritt, M. Introduction to the Theory of Nested Transactions. *ICDT'86 International Conference on Database Theory.* Rome, Italy, September, 1986, pp. 278-305. Also, MIT/LCS/TR-367 July 1986. A revised version will appear in *Theoretical Computer Science.*

[LMF] Lynch, N., Mansour, Y., and Fekete, A. Data Link Layer: Two Impossibility Results. In *Proceedings of 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Toronto, Ontario, Canada, August, 1988, pp. 149-170.

[LMWF] Lynch, N., Merritt, M., Weihl, W., and Fekete, A. Atomic Transactions. In progress.

[LS] Lynch, N.A., and Stark, E.W A Proof of the Kahn Principle for Input/Output Automata. To appear in *Information and Computation.*

[LT1] Lynch, N.A., and Tuttle, M.R. Hierarchical Correctness Proofs for Distributed Algorithms. Master's Thesis, Massachusetts Institute of Technology, April, 1987. MIT/LCS/TR-387, April, 1987.

[LT2] Lynch, N.A., and Tuttle, M.R. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Vancouver, British Columbia, Canada, August, 1987, pp. 137-151.

[LW] Lynch, N.A., and Welch, J.L. Synthesis of Efficient Drinking Philosophers Algorithms. In progress.

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

[P] Peterson, G.L. An O(nlogn) Unidirectional Algorithm for the Circular Extrema Problem. *ACM TOPLAS (4),* October, 1982. pp. 758-762.

[S] Schaffer, R. On the Correctness of Atomic Multi-writer Registers. Technical Memo MIT/LCS/TM-364, MIT Laboratory for Computer Science, June, 1988.

[TMM] Tuttle, M., Merritt, M., and Modugno, F. Time Constrained Automata. Unpublished manuscript.

[WLL] Welch, J., Lamport, L., and Lynch, N. A Lattice-Structured Proof of a Minimum Spanning Tree Algorithm. In *Proceedings of 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Toronto, Ontario, Canada, August, 1988, pp. 28-43.