I/O Automata: A Model for Discrete Event Systems[1]

Nancy Lynch
Massachusetts Institute of Technology
Cambridge, Mass. 02139

1. Introduction

The *input/output automaton* model has recently been defined, in [LT1,LT2], as a tool for modelling 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. The simplicity and generality of the model and its similarities with other new models [RW,CM1] suggest that it will prove useful in other application areas, such as control theory and manufacturing.

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 contains formal definitions and some basic results. Section 4 contains an illustrative example, candy machines. Section 5 contains a second example, a system that elects a leader. Finally, Section 6 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. The components, as well as the entire system, may be "reactive" in the sense that they interact with their environments in an ongoing manner (rather than, say, simply accepting an input, computing a function of that input and halting). Although I/O automata can be used to model synchronous systems, they are best suited for modelling systems in which the components operate asynchronously.

Each system component is modelled as an "I/O automaton", which is a mathematical object somewhat like a traditional automaton. However, an I/O automaton need not be finite-state, but can have an infinite state set. The actions of an I/O automaton are classified as either "input", "output", or "internal". The automaton generates output and internal actions autonomously, and transmits output actions 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

[1]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.

an input action.[2]

The fact that our automata are unable to block inputs distinguishes our model from Hoare's CSP (Communicating Sequential Processes) [Ho]. There, input blocking is used for two purposes: as a way of eliminating undesirable inputs, and as a way of blocking the activity of the environment. Our model does not have any way of blocking the environment, but does have other ways of coping with bad inputs. For example, suppose that we wish to constrain the behavior of an automaton only in case the environment observes certain restrictions on the production of inputs. Instead of requiring the automaton to block the bad inputs, we permit these inputs to occur; however, we may permit the automaton to exhibit arbitrary behavior in case they do. Alternatively, we may require the automaton to detect bad inputs and respond to them with error messages. Thus, we have simple ways of describing 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. Our composition operator connects each output action of one automaton with input actions of any number (usually one) of other automata. In the resulting system, an output action is thus generated autonomously by one component and instantaneously transmitted to all the other components 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.

Since I/O automata are intended to model complex systems with any number of primitive components, each automaton comes equipped with an abstract notion of "component"; formally, these components are described by an equivalence relation on the automaton's output and internal actions, where all the actions in one equivalence class are to be thought of as under the control of the same primitive system component.

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 (i.e., 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.

[2]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].

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 automaton definitions imply that every automaton has a nonempty set of fair behaviors.[3] The fact that inputs are always allowed gives another reason why our definition of solving a problem is nontrivial: 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 constrain the user to describe all automata in this manner; for example, the model is general enough to serve also as a formal basis for languages that include more elaborate constructs for sequential flow of control.

Our model also allows precise statement 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. Our model is general enough to serve as a semantic model for many different languages for 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]Even a trivial automaton having no actions at all has one fair behavior — the empty sequence of actions.

3. Definitions and Basic Results

This section contains some of the basic definitions and results about the model. This material is adapted from [LT1].

3.1. Actions and Action Signatures

We assume a universal set of *actions*. Sequences of actions are used in this work, for describing the behavior of modules in concurrent systems. Since the same action may occur several times in a sequence, it is convenient to distinguish the different occurrences. Thus, we refer to a particular occurrence of an action in a sequence as an *event.*

The actions of each automaton are classified as either "input", "output", or "internal". The distinctions are that input actions are not under the automaton's control, output actions are under the automaton's control and externally observable, and internal actions are under the automaton's control but not externally observable. In order to describe this classification, each automaton comes equipped with an "action signature".

An *action signature* S is an ordered triple consisting of three pairwise-disjoint sets of actions. We write *in(S)*, *out(S)* and *int(S)* for the three components of S, and refer to the actions in the three sets as the *input actions*, *output actions* and *internal actions* of S, respectively. We let *ext(S)* = in(S) $\cup$ out(S) and refer to the actions in ext(S) as the *external actions* of S. Also, we let *local(S)* = out(S) $\cup$ int(S), and refer to the actions in local(S) as the *locally-controlled actions* of S. Finally, we let acts(S) = in(S) $\cup$ out(S) $\cup$ int(S), and refer to the actions in acts(S) as the *actions* of S. An *external action signature* is an action signature consisting entirely of external actions, that is, having no internal actions. If S is an action signature, then the *external action signature* of S is the action signature *extsig(S)* = (in(S),out(S),$\emptyset$), i.e., the action signature that is obtained from S by removing the internal actions.

3.2. Input/Output Automata

Now we are ready to define the basic component of our model. 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(sig(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)* on local(sig(A)), having at most countably many equivalence classes.

We refer to an element (s',π,s) of steps(A) as a *step* of A. The step (s',π,s) is called an *input step* of A if π is an input action. *Output steps, internal steps, external steps* and *locally-controlled steps* are defined analogously. 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.* The input-enabling property means that the automaton is not able to block input actions. The partition part(A) is what was described in the introduction as an abstract description of the "components" of the automaton. It is used to define fairness.

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 $s_0,\pi_1,s_1,\pi_2,\ldots,\pi_n,s_n,\ldots$ of alternating states and actions of A such that (s_i,π_{i+1},s_{i+1}) is a step of A for every i. An execution fragment beginning with a start state is called an *execution*. We denote the set of executions of A by *execs(A)*, and the set of finite executions of A by *finexecs(A)*. A state is said to be *reachable* in A if it is the final state of a finite execution of A.

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 else α contains infinitely many occurrences of states in which no action of C is enabled.

Thus, a fair execution gives "fair turns" to each class of part(A). We denote the set of fair executions of A by *fairexecs(A)*.

The *schedule* of an execution fragment α of A is the subsequence of α consisting of actions, and is denoted by *sched(α)*. We say that β is a *schedule* of 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)*. 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)*. 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)*. 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 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)*.

3.3. Schedule Modules

In order to describe problems to be solved by automata, we need to describe sets of sequences. More precisely, a problem will be specified by a pair consisting of an action signature and a set of sequences over the actions in that signature. (In most interesting cases, the action signature will be an external action signature.) The mathematical object used to describe a problem is called a "schedule module".

A *schedule module* H consists 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. Let *finscheds(H)* denote the set of finite members of scheds(H).

The *behavior* of a schedule β of H is the subsequence of β consisting of external actions, and is denoted by *beh(β)*. We say that β is a *behavior* of H if β is the behavior of an execution 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 use the term *module* to designate either an automaton or schedule module. If M is a module, we sometimes write *acts(M)* as shorthand for acts(sig(M)), and likewise for in(M), out(M), etc. If β is any sequence of actions and M is a module, we write $\beta|M$ for $\beta|$acts(M).

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, let *Behs(M)*, *Finbehs(M)* and *Fairbehs(M)* be the schedule modules having 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.)

3.4. Solving Problems

Now we are ready to define our notion of "solving".[4] This notion is intended for describing the way in which particular automata solve particular problems (formalized as schedule modules). However, it is convenient to state the definition more generally. Let M and M' be modules (i.e., either automata or schedule modules) with the same external action signature. Then M' is said to *solve* M if fairbehs(M') $\subseteq$ fairbehs(M).

In the most interesting case, M' is an automaton and M is a schedule module. However, the more general formulation allows us to carry out proofs in several stages: in order to show that an automaton solves a problem, we can show that the automaton "solves" another automaton, which in turn solves another automaton, and so on, until some final automaton solves the problem. A variety of techniques can be used to show that an automaton M' solves a schedule module M; we will mention some of these below.

3.5. Implementation

One way of showing that one module solves another is to use an intermediate result about inclusion for the sets of finite behaviors. Thus, we define an analog of the "solving" definition for finite behaviors only. Let M and M' be modules with the same external action signature. Then M' is said to *implement* M if finbehs(M') $\subseteq$ finbehs(M).

It is often possible to show that one automaton implements another using a mapping between automaton states. 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). The mapping f is said to be a *possibilities mapping* from A to B if the following conditions hold:

1. For every start state s of A, there is a start state t of B such that t $\in$ f(s).
2. For every reachable state s' of A, every step (s',π,s) of A, and every reachable state t' $\in$ f(s') of B:
    a. If $\pi \in$ acts(B), then there is a step (t',π,t) of B such that t $\in$ f(s).

    b. If $\pi \notin$ acts(B), then t' $\in$ f(s).

Lemma 1: Suppose that A and B are automata with the same external action signature and there is a possibilities mapping from A to B. Then A implements B.

It is possible to show that one module M' solves another module M using this lemma together with

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

additional results showing correspondences between fairness properties of M and M'. Some such additional results are given in [LT1] and [WLL].

3.6. Composition

The most useful way of combining I/O automata is by means of a composition operator, as defined in this subsection.

3.6.1. Composition of Action Signatures

Let I be an index set that is at most countable. A collection $\{S_i\}_{i\in I}$ of action signatures is said to be *strongly compatible*[5] if for all i, j ∈ I, we have

1. $out(S_i) \cap out(S_j) = \emptyset$,
2. $int(S_i) \cap acts(S_j) = \emptyset$, and
3. no action is in $acts(S_i)$ for infinitely many i.

Thus, no action is an output of more than one signature in the collection, and internal actions of any signature do not appear in any other signature in the collection.

The *composition* $S = \Pi_{i\in I}S_i$ of a 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
- $int(S) = \cup_{i\in I}int(S_i)$.

Thus, output actions are those that are outputs of any of the component signatures, and similarly for internal actions. Input actions are any actions that are inputs to any of the component signatures, but outputs of no component signature.

3.6.2. Composition of Automata

A collection $\{M_i\}_{i\in I}$ of modules is said to be *strongly compatible* if their action signatures are strongly compatible. The *composition* $A = \Pi_{i\in I}A_i$ of a strongly compatible collection of automata $\{A_i\}_{i\in I}$ has the following components:

- $sig(A) = \Pi_{i\in I}\, sig(A_i)$,
- $states(A) = \Pi_{i\in I}\, states(A_i)$,[6]
- $start(A) = \Pi_{i\in I}\, start(A_i)$,
- 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 $\vec{s}_1[i] = \vec{s}_2[i]$,[7] and
- $part(A) = \cup_{i\in I}part(A_i)$.

---

[5]Such a collection is said to be *compatible* if it satisfies the first two of the three listed properties. 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] and [LMW].

[6]Note that the second and third components listed are just ordinary Cartesian products, while the first component uses a previous definition.

[7]We use the notation $\vec{s}[i]$ to denote the ith component of the state vector $\vec{s}$

Since the automata A_i are input-enabled, so is their composition, and hence their composition is an automaton. Each step of the composition automaton consists of all the automata that have a particular action in their signatures performing that action concurrently, while the automata that do not have that action in their signatures do nothing. The partition for the composition is formed by taking the union of the partitions for the components. Thus, a fair execution of the composition gives fair turns to all of the classes within all of the component automata. In other words, all component automata in a composition continue to act autonomously. If $\alpha = \vec{s}_0\pi_1\vec{s}_1\ldots$ is an execution 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 $\vec{s}_j[i]$.

The following basic results relate executions, schedules and behaviors of a composition to those of the automata being composed. The first result says that the projections of executions of a composition onto the components are executions of the components, and similarly for schedules, etc. The parts of this result dealing with fairness depend on the fact that at most one component automaton can impose preconditions on each action.

Lemma 2: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i\in I}A_i$. If $\alpha \in$ execs(A) then $\alpha|A_i \in$ execs(A_i) for all $i \in I$. Moveover, the same result holds for finexecs, fairexecs, scheds, finscheds, fairscheds, behs, finbehs and fairbehs in place of execs.

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

Lemma 3: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i\in I}A_i$. For all $i \in I$, let α_i be an execution of A_i. Suppose β is a sequence of actions in ext(A) such that $\beta|A_i$ = beh(α_i) for every i. Then there is an execution α of A such that β = beh(α) and $\alpha_i = \alpha|A_i$ for all i. Moreover, if α_i is a fair execution of A_i for all i, then α may be taken to be a fair execution of A.

Similarly, schedules or behaviors of component automata can be patched together to form schedules or behaviors of the composition.

Lemma 4: Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata, and let $A = \Pi_{i\in I}A_i$. Let β be a sequence of actions in acts(A). If $\beta|A_i \in$ scheds(A_i) for all $i \in I$, then $\beta \in$ scheds(A). Moreover, the same result holds for fairscheds, behs and fairbehs in place of scheds.

The previous lemmas are often useful in proving that certain automata solve certain problems. In particular, sometimes correctness conditions are formulated to say that every behavior of an automaton is also a behavior of a given composition A. One way of showing that a given sequence of actions is a behavior of A is by first showing that its projections are behaviors of the components of A and then appealing to the preceding lemmas.

3.6.3. Composition of Schedule Modules

Corresponding to our composition operator for automata, we also define a composition operator for schedule modules. The *composition* $H = \Pi_{i\in I}H_i$ of strongly compatible schedule modules $\{H_i\}_{i\in I}$ is defined to be the schedule module with

- sig(H) = $\Pi_{i\in I}$ 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 lemma shows how composition of schedule modules corresponds to composition of automata.

Lemma 5: Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Then Scheds(A) = $\Pi_{i \in I}$Scheds(A_i), Fairscheds(A) = $\Pi_{i \in I}$Fairscheds(A_i), Behs(A) = $\Pi_{i \in I}$Behs(A_i) and Fairbehs(A) = $\Pi_{i \in I}$Fairbehs(A_i).

3.7. Preserving Properties

Although automata in our model are unable to block input actions, it is often convenient to restrict attention to behaviors in which the environment obeys certain "well-formedness" restrictions. A useful way of discussing such restrictions is in terms of the notion that a module "preserves" a property of behaviors: as long as the environment does not violate the property, neither does the module. Such a notion is primarily interesting for properties that are "prefix-closed".

A set of sequences P is *prefix-closed* provided that whenever $\alpha \in P$ and β is a prefix of α, it is also the case that $\beta \in P$. A module M is said to be *prefix-closed* provided that behs(M) is prefix-closed. Let M be any prefix-closed module and let P be a prefix-closed set of sequences of actions in ext(M). We say that M *preserves* P if $\beta = \beta'\pi \in$ finbehs(M), $\pi \in$ out(M) and $\beta' \in P$ together imply that $\beta \in P$. Thus, if a module preserves a property P, the module is not the first to violate P: as long as the environment only provides inputs such that the cumulative behavior satisfies P, the module will only perform outputs such that the cumulative behavior satisfies P.

3.8. Hiding Actions

Here we define an operator that "hides" some of the output actions of a module by converting them to internal actions. We begin with a hiding operator on action signatures: if S is an action signature and Σ is a subset of out(S), define *hide*$_\Sigma$(S) = S', where in(S') = in(S), out(S') = out(S) — Σ and int(S') = int(S) $\cup\ \Sigma$. Now we use the hiding operator on signatures to define a hiding operator for automata and schedule modules: if M is a module with signature S, and $\Sigma \subseteq$ out(S), then let *hide*$_\Sigma$(M) be the module M' that coincides with M except that sig(M') = hide$_\Sigma$(sig(M)).

4. Candy Machines

In this section, we illustrate many of the preceding definitions using examples of simple candy machines. (This class of examples is popular in the CSP literature, so this choice should facilitate comparison of the models.) These examples show how our model is used to define simple nondeterministic processes. They also show how problems can be stated, and how it can be proved that certain automata solve certain problems. Finally, they show how processes can interact in the model, although the style of interaction is very simple (normally a strict alternation of button pushes and candy dispensations).

4.1. Candy Machines

In this subsection, we describe three specific candy machines as I/O automata. Candy machine model CM-1 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, respectively, and the three dispensation actions as S, H and A. The state of CM-1 consists of one variable "button_pushed", which takes on values 0, 1 and 2, initially 0. Next we describe the transition relation of CM-1. It should not be hard for the reader to translate the given description into a transition relation: (s',π,s) is a step of the automaton exactly if the precondition of π (if any) is satisfied in s' and s is a possible result of running the code in π's "Effect" starting from s'.

PUSH1
  Effect: button_pushed := 1

PUSH2
  Effect: button_pushed := 2

SKYBAR
  Precondition: button_pushed = 1
  Effect: button_pushed := 0

HEATHBAR
  Precondition: button_pushed = 2
  Effect: button_pushed := 0

ALMONDJOY
  Precondition: button_pushed = 2
  Effect: button_pushed := 0

Thus, 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 partition for this automaton, part(CM-1), is defined to group together ALMONDJOY and HEATHBAR and to keep SKYBAR in a singleton set.

Candy machine model CM-2 is identical to CM-1 except that its HEATHBAR action has Precondition "false". This candy machine never dispenses HEATHBARs, but is able to dispense SKYBARs and ALMONDJOYs. Model CM-3 is identical to CM-1 except that all three candy dispensation actions have Precondition "false". That is, it never dispenses candy. As one might expect, it is not a very useful candy machine from the point of view of the customer.

4.2. Specifications for Candy Machine Behavior

Now we describe some interesting notions of correct candy machine behavior.

4.2.1. Safe 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, i.e., that it solves the problem described by SAFE-CM, we must show that all fair behaviors of CM-1 satisfy the given requirement. Note that this requirement, (as usual for safety requirements) holds for an infinite sequence if and only if it holds for all finite prefixes of the infinite sequence. Therefore, it suffices to 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, i.e., button_pushed = 1 if the last event in the sequence is PUSH1, 2 if the last event in the sequence is PUSH2, and 0 otherwise (i.e., if the sequence is empty, or if the last event is a dispensation event). The inductive step considers cases based on the five possible events. 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 is a safe candy machine.

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 fair behaviors are just the finite and infinite 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 will give a stronger specification below.

4.2.2. Well-Formedness

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, e.g. 11S11S... is a non-well-formed fair behavior of CM-1. This is not surprising, since CM-1 does not (in our model) have the power to insure that its environment preserves well-formedness. However, it is easy to see that any safe candy machine, including CM-1, preserves well-formedness, according to the definition of "preserves" given in Section 3.

4.2.3. Live Candy Machine Behavior

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 push event has a later dispensation event."[8]

CM-3 is not a live candy machine, because it has fair behaviors, such as the sequence with the single element 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. By well-formedness, 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

[8]This can be expressed using temporal logic, in the form $W \rightarrow \Box(P \rightarrow \Diamond D)$, where W is the set of well-formed candy machine sequences, P is the set of sequences beginning with a push action, and D is the set of sequences beginning with a dispensation action.

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.

For the reasons discussed in Section 2, LIVE-CM does not admit trivial solutions. Anything that satisfies the specification must be able to respond to any pattern of pushes (since it is an I/O automaton, with the input-enabling condition). Moreover, responses have to be safe, and if the pushes arrive in a well-formed way, responses must in fact be made.[9]

4.3. Customers

We now describe particular customers that might interact with a candy machine. It is convenient also to describe such customers as I/O automata also. Customer CUST-1 continues to request candy bars ad infinitum, nondeterministically choosing which button to push. CUST-1's action signature is the "complement" of that 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", initially "no". CUST-1's actions are as follows.

SKYBAR
Effect: waiting := no

HEATHBAR
Effect: waiting := no

ALMONDJOY
Effect: waiting := no

PUSH1
Precondition: waiting = no
Effect: waiting := yes

PUSH2
Precondition: waiting = no
Effect: waiting := yes

The partition part(CUST-1) puts PUSH1 and PUSH2 together in one equivalence class. It is easy to

[9]One might ask the technical question whether it might be reasonable to eliminate the well-formedness hypothesis in the live candy machine behavior 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 121212... of CM-1. This contains push events that are not followed by dispensation events. However, we claim it is a fair behavior of CM-1, since each class in the partition part(CM-1), {S} and {A,H}, has infinitely many points in the sequence at which no action in that class is enabled. (It might be helpful for the reader to imagine that there are two "processes" inside the candy machine, where process 1 is in charge of dispensing SKYBARS and process 2 is in charge of dispensing ALMONDJOYS and HEATHBARS. Every time process 1 tries to perform its task, it happens that the value of button_pushed is 2, so it cannot do anything. Similarly, every time process 2 tries to perform its task, the value of button_pushed is 1. So neither process can cause any output to occur.) Since we have exhibited a fair behavior of CM-1 that contains a push but no later dispensation, CM-1 does not satisfy the proposed stronger specification.

see that CUST-1 preserves well-formedness; in fact, it never pushes unless all previous pushes have been followed by dispensations. Also, in any well-formed fair behavior, after any dispensation event, CUST-1 eventually pushes a button once again.

Customer CUST-2 is somewhat more selective than CUST-1. It pushes button 2 repeatedly just until the machine dispenses a HEATHBAR. Then it 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 := no; heathbar_received := yes

PUSH1
 Precondition: waiting = no; heathbar_received = yes
 Effect: waiting := yes

PUSH2
 Precondition: waiting = no; heathbar_received = no
 Effect: waiting := yes

It is easy to show that CUST-2 implements CUST-1, using a possibilities mapping f 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. In fact, it can be shown that CUST-2 solves CUST-1, according to our formal definition of "solves". 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 f(s) for CUST-1.

Customer CUST-3 is similar to CUST-1 except that it is required eventually to take 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
 Effect: satiated := yes

Also, each of PUSH1 and PUSH2 has the additional Precondition "satiated = no". The BECOME_SATIATED action is in a class by itself in part(CUST-3).

Note that CUST-3 implements CUST-1, but does not solve CUST-1; there are fair behaviors of CUST-3, such as the empty sequence, that are not fair behaviors of CUST-1.

4.4. Candy Machines and Customers

Now we consider the composition of candy machines and customers. First consider the composition of CM-1 and CUST-1. Since each component preserves well-formedness, the composition has only well-formed behaviors. We claim that all fair behaviors of the composition are infinite. Suppose not: then consider any finite fair execution. By well-formedness and a simple assertion characterizing the states after finite executions, the state of the composition after the execution either has waiting = "no" and button_pushed = 0, or has waiting = "yes" and button_pushed = 1 or 2. In the former case, PUSH1 is enabled, while in the latter case, either SKYBAR or HEATHBAR is enabled. But the definition of a fair execution implies that no action of the composition can be enabled in the final state.

In fact, it is not hard to see that the fair behaviors of the composition of CM-1 and CUST-1 are exactly the infinite well-formed sequences in which each dispensation action dispenses an appropriate candy (according to the most recent push).

The composition of CM-1 and CUST-2 yields exactly the sequences of the form 2,A,2,A,...,2,A,2,A..., or 2,A,2,A,...,2,A,2,H,1,S,1,S,... as its fair behaviors. The composition of CM-1 and CUST-3 produces exactly the even-length finite well-formed sequences in which each dispensation action dispenses an appropriate candy. Also, the composition of CM-2 and CUST-2 yields the single sequence 2,A,2,A,...,2,A,2,A.... as its only fair behavior. All of these, and similar characterizations for the behavior of the other compositions, can be proved by straightforward methods similar to those used above.

The previous arguments about the behavior of compositions of automata are based directly on the internal structure of the component automata. Sometimes it is possible to break up such a proof, using properties of the behavior of the component automata to prove a property of the composition. Formally, in order to prove that the composition of the automata $\{A_i\}_{i \in I}$ solves a problem, one might prove that each component automaton A_i solves a schedule module H_i, and then prove that the composition of the $\{H_i\}_{i \in I}$ solves the problem.

For example, we reconsider 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 1. the customer is not the first to violate well-formedness, and 2. 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.

5. Choosing a Ring Leader

Now we give a brief sketch of another example, 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 a unique processor to perform a "leader" output action. This problem has been widely studied in the distributed algorithms research area.

Each processor and each communication channel is modelled as an I/O automaton. Each channel automaton has input actions of the form SEND(M) and output actions of the form RECEIVE(M).[10] 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 := messages ∪ {M}

RECEIVE(M)
 Precondition: M ∈ messages
 Effect: messages := messages — {M}
```

The partition puts each different RECEIVE action in a separate equivalence class; this has the effect of hypothesizing that every message that is sent eventually gets received.

Each processor is also modelled 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]. 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 steps are as follows.

```
RECEIVE(j), j ∈ I
 Effect: if j > i then pending := pending ∪ {j}
  if j = i then leader-status := "elected"

SEND(j), j ∈ I
 Precondition: j ∈ pending
 Effect: pending := pending — {j}

LEADER
 Precondition: leader-status = "elected"
 Effect: leader-status := "announced"
```

Each action is in a separate class of the partition. It is not hard to carry out a correctness proof of this

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

algorithm using the model. The safety proof (i.e., 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 (i.e., 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 variant function yields the remaining distance the maximum identifier needs to travel before reaching its originating processor. The value of this function is shown never to increase during execution, and at any point where it is nonzero, the fairness properties of I/O automata imply that some event will eventually occur to decrease the value. Thus, eventually, the function value reaches zero, which implies that a LEADER event 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 see that n^2 is a worst-case upper bound on this number, where n is the number of processors in the ring. 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 real time assigned to the LEADER event can be taken as a time measure for the entire execution. It is not hard to see that $2n + 1$ is a worst-case upper bound for the time measure.

The given algorithm is not optimal in its communication requirements; for 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 $\Omega(n \log n)$ lower bound on the worst-case amount of communication; this result also is describable in our model.

6. 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.

6.1. 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. In contrast, 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 a request does not overlap with any others, 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 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 he 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 Philosphers 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].

6.2. 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 fairly 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.

6.3. 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].

More recently, we have been using I/O automata to characterize correct behavior for physical channels and data links. We are attempting to 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, it appears that the data link protocol must use unbounded size headers to achieve reasonable behavior, in case the underlying physical channels are not FIFO.

6.4. 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 [LMW] 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.

6.5. 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 focussed 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 [L], 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 Herlihy on impossibility results for atomic object implementations [He] also uses the I/O automaton model.

6.6. Dataflow

In [LS], we formulate the semantics of dataflow networks in terms of I/O automata. We define the notion of "determinacy" (i.e., 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.

Bibliography

[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.

[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.

[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.

[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. Submitted for publication.

[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.

[L] 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.

[LF] Lynch, N.A., and Fisher, 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, to appear in *Theoretical Computer Science.*

[LMW] Lynch, N., Merritt, M., and Weihl, W. Atomic Transactions. In progress.

[LS] Lynch, N.A., and Stark, E.W A Proof of the Kahn Principle for Inout/Output Automata. Submitted for publication.

[LT1] 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.

[LT2] 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.

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

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

[RW] Ramadge, P.J., and Wonham, W.M. Supervisory Control of a Class of Discrete Event Processes. University of Toronto. November, 1985. Systems Control Group Report #8515.

[WLL] Welch, J., Lamport, L., and Lynch, N. A Lattice-Structured Proof of a Minimum Spanning Tree Algorithm. Submitted for publication.

References

Table of Contents