On Describing the Behavior and Implementation of Distributed Systems*

Nancy A. Lynch
Georgia Institute of Technology
Atlanta, Georgia 30332/USA

Michael J. Fischer
University of Washington
Seattle, Washington 98195/USA

Abstract: A simple, basic and general model for describing both the (input-output) behavior and the implementation of distributed systems is presented. An important feature of the model is the separation of the machinery used to describe the implementation and the behavior. This feature makes the model potentially useful for design specification of systems and of subsystems.

The implementation model relies on the basic notions of process and variable, assuming indivisibility of variable access. Long-distance communication is modelled by a special process representing a "channel." Process executions are considered to be completely asynchronous; this consideration is reflected in the fairness of the operations for combining processes. The primitivity and generality of the model make it an apparently suitable basis for cost comparison of various message-passing protocols and other higher-level programming constructs, as well as of complex distributed system implementations.

A system's (input-output) behavior is modelled by a set of finite and infinite sequences of actions, each action involving access to a variable.

Basic definitions, examples and characterization results are given. An extended example, involving specification and implementation of an arbiter system, is presented. For this example, equivalent implicit and explicit specifications are given. Several different implementations are described, each of which exhibits the required behavior.

General remarks are made about cost comparison of distributed system implementations.

*This research was supported in part by the National Science Foundation under grants MCS77-02474 and MCS77-15628.

I. Introduction

A distributed computing system consists of a number of distinct and logically separated communicating asynchronous sequential processes. To gain a theoretical understanding of such systems, it is necessary to find simple mathematical models which reflect the essential features of these systems while abstracting away irrelevant details. Such models allow problems to be stated precisely and make them amenable to mathematical analysis.

In this paper, we present a mathematical model of distributed systems and a mathematical model of their input/output behavior. Both are set-theoretic models built from standard mathematical constructs such as set, sequence, function, and relation, rather than axiomatic models consisting of lists of desired properties of systems without a basis for validity or consistency.

In constructing a model, choices must be made regarding which features of actual systems to preserve and which to abstract away, and how these choices are made depends on the intended applications of the model. Our interests are in finding a low-level model that reflects closely many aspects of physical reality and that permits problems of communication and synchronization to be studied. Thus, we do not assume any primitive synchronization mechanism such as is implicit in Petri nets [1] or in the communicating sequential processes of Hoare [2] and of Milne and Milner [3]. We have also chosen to omit from our model any notion of time. Although we realize clocks and time-outs are important mechanisms in real distributed systems, many aspects of distributed computation can nevertheless be modelled without reference to such concepts, and the resulting simplicity and tractability of the model more than compensates for the limitations imposed on it. Eventually, of course, time needs to be introduced into a suitable formal model and studied.

We are concerned with the cooperative behavior of processes, not their internal structure. Hence, we assume simply that each process is an automaton with a possibly infinite number of internal states and an arbitrary set of possible transitions. Each process from time to time takes a step, but we make no assumptions on how long it waits between steps except that the time is finite -- it does not wait forever.

We also permit our processes to exhibit infinitely-branching nondeterminism. This is done because we wish our notion of "process" to encompass not only what a single processor acting alone can do but also what a subsystem of processes or module can do. That will permit us to describe the behavior of a complete system in terms of the behaviors of component modules. Since a system of two deterministic processes can exhibit infinite nondeterminism, we are led to include this capability in our model.

We have chosen the shared variable as our basic (and only) communication mechanism. Because of the popularity of message-based distributed systems and the immediate reaction that a "central" shared memory does not constitute true distribution, some words

about this choice are in order.

First of all, at the most primitive level, something must be shared between two processors for them to be able to communicate at all. This something is usually a wire in which, at the very least, one process can inject a voltage which the other process can sense. We can think of the wire as a binary shared variable whose value alternates from time to time between 0 and 1. Note that we are not specifying the protocols to be used by the sending and receiving processes which enable communication to take place -- indeed part of our interest is in modelling and studying such protocols. All we have postulated so far is that the sending process can control the value on the wire and the receiving process can sense it. The setting and sensing correspond to writing and reading, respectively. We contend that shared variables are at the heart of every distributed system.

Because of our decision to leave time out of the model, it is clear that the only way for the receiving process to be sure of seeing a value written by the sending process is for the latter to leave the value there until it gets some sort of acknowledgement from the receiver. Thus, we cannot model the asynchronous serial communication that is commonly used to communicate between terminals and computers, for the success of that method relies on sender and receiver having nearly identical clocks.

We have argued so far that shared variables underlie any timing-independent system, but that certain kinds of communication which depend on time cannot be modelled. Does introducing timing-dependent communication primitives into our otherwise timing-independent system add any new power? Let's consider various possible message primitives. Perhaps the simplest is to assume each process has a "mailbox" [VAX/VMS] or "message buffer" into which another process can place a message. Now, what happens when the sender wants to send a second message before the receiver has seen the first? If the second message simply overwrites the first, then the buffer behaves exactly like a shared variable whose values range over the set of possible messages. If the sender is forced to wait, then there is an implicit built-in synchronization mechanism as in [2,3] which we have already rejected for our model. As a third possibility, the message might go into a queue of waiting messages. If the queue is finite, the same problem reappears when the queue gets full. An infinite queue, on the other hand, seems very non-primitive and can be rejected for that reason alone. In any case, if the needed storage is available, the infinite message queue can be modelled in our system by a process with two shared variables: an input buffer and an output buffer. The process repeatedly polls its two buffers, moving incoming messages to its internal queue, and moving messages from the queue to the output buffer whenever it becomes empty. Of course, the sender must wait until the input buffer becomes empty before writing another message, but it seems to be an essential property of any communication system that there will be a maximum rate at which messages can be sent, and the sender attempting to exceed that rate must necessarily wait if information is

not to be lost. (We note also that the delays inherent in long-distance communication between asynchronous processes can also be modelled simply in our framework.)

From the above discussion, we see that various message systems can be modelled naturally using shared variables, provided the variables are not restricted to binary values. Also, there are situations in which it is natural for a variable to have more than one reader or writer. We incorporate such generalized variables in our model. Finally, we generalize our model in one more respect by permitting a variable to be read and updated in a single step. We call such an operation test-and-set. This simplifies the model since both reads and writes are special cases of test-and sets. Moreover, there are situations in which the natural primitive operations are not read and write but are other test-and-set operations such as Dijkstra's P and V [4]. They all become just special cases of our general model. The formal definition of the model appears in Section 2.

A class of interesting and important questions to be addressed by a theory of distributed systems concerns the relative "goodness" of various systems all of which solve the "same" problem. Before these questions can be investigated, one needs appropriate measures of "goodness" (complexity measures) and one needs a precise notion of the "problem" to be solved by a distributed system. We make some brief remarks about complexity measures in Section 5, but a thorough treatment must await another paper. In Section 3, we construct a formal notion of "distributed problem" and define precisely when a given system solves a given problem. Section 4 gives an example of a distributed problem and several radically different systems for solving it.

Several factors contribute to making a satisfactory notion of "distributed problem" considerably more complicated than the simple input-output function which is often identified with the behavior of a sequential program.

1. There is generally more than one site producing inputs and receiving outputs.
2. Infinite, non-terminating computations are the rule rather than the exception.
3. The relative orders of reading inputs and producing outputs is significant as well as the actual values produced.
4. Variations in timing make distributed systems inherently nondeterministic, so one must allow in general for several different possible outputs to a given sequence of inputs, all of which must be considered "correct".

Briefly, we define the behavior of a distributed system to be a set of finite and infinite sequences of interleavings of possible activities at certain external variables (which are assumed to be used for communication with the outside world). Each sequence in the set describes a possible sequence of actions by the system, assuming particular actions affecting the variables by the environment. An action is a triple (u,x,v) consisting of a variable x, the value u read from the variable and the value v written back into the variable. Since the environment can change a variable at any time, it is not true that the system will necessarily see the same value in a variable that it most recently wrote there. We require of the

behavior only that it be complete in that it describe at least one possible series of responses by the system for every possible way that the environment might behave.

A problem specification is an arbitrary set of input/output sequences. A particular system is a solution to the problem if its behavior is contained in the problem specification. The problem specification is the set of acceptable computations, while the solution behavior is the set actually realized.

Our definition only requires the solution system to be correct; there is no stipulation that the maximum permitted degree of nondeterminism actually be exhibited. We regard the latter as a performance or complexity issue to be dealt with separately. We remark that the distributed computing paradigm leads one to a very different view of nondeterminism or concurrency than for multiprocessing. In the latter case, the system implementer is presumed to have control of the scheduler, so the greater the possible concurrency among the processes he is trying to schedule, the greater his freedom to do so efficiently. In a truly asynchronous environment, however, one has no direct control over the scheduling, so it is natural to be concerned with the worst case (which might actually occur) rather than the best case. Hence, decreasing the amount of nondeterminism in this situation can never hurt.

We do not address in this paper another important aspect of problem specification, namely, what is an appropriate formal language for describing the sets of sequences that comprise a problem specification? Our example in Section 4 is described informally in standard mathematical notation. We expect the work on path expressions [5, etc.], flow expressions [6], and other formal systems of expressions might be applicable here.

2. A Model for Distributed Systems

Processes and Shared Variables

The primitive notions in our model are that of "process" and variable". A process can be thought of as a sequence of changes of state; likewise, a variable is a sequence of changes of value. The interaction among system components occurs at the process-variable interface.

Each variable x has an associated set of values, values(x), which the variable can assume. A variable action for x is a triple (u, x, v) with u, v $\in$ values(x); intuitively, it represents the action of changing the value of x from u (its old value) to v (its new value). (u and v are not required to be distinct.) Act(x) is the set of all variable actions for x. If X is a set of variables, we let act(X) = $\bigcup_{x \in X}$ act(x).

A process p has an associated set (finite or infinite) of process states, states(p), which it can assume. Start(p) is a nonempty set of starting states, and final(p) a set of final or halting states. We let nonfinal(p) = states(p) - final(p).

A process action for p is a triple (s, p, t) with s ε nonfinal(p), t ε states(p); it represents intuitively the action of p going from state s to state t in a single step. (s and t are not required to be distinct.) Act(p) is the set of all process actions for p. If P is a set of processes, we let $act(P) \stackrel{df}{=} \bigcup_{p \in P} act(p)$.

Every process action occurs in conjunction with a variable action; the pair forms a complete execution step. If P is a set of processes and X a set of variables, we let $steps(P,X) \stackrel{df}{=} act(P) \times act(X)$ be the set of execution steps. To specify which steps are permitted in a computation, a process has two other components in its description. Variables(p) is a set of variables which the process can access. Oksteps(p) is a subset of steps(p, variables(p)) describing the permissible steps of p. Oksteps(p) is subject to three restrictions:

(a) For any s ε nonfinal(p), there exist t, u, x, v with ((s, p, t), (u, x, v)) ε oksteps(p).

(b) (Read Anything): If ((s, p, t), (u, x, v)) ε oksteps(p) and u' ε values(x), then there exist t', v' with ((s, p, t'), (u', x, v')) ε oksteps(p).

(c) (Countable Nondeterminism): Start(p) is countable, and also for any s ε nonfinal(p), x ε variables(p) and u ε values(x), there are only countably many pairs t, v with ((s, p, t), (u, x, v)) ε oksteps(p).

Some intuitive remarks are in order. Oksteps(p) represents the allowable steps of p. A particular step ((s, p, t), (u, x, v)) ε oksteps(p) is applicable in a given situation only if p is in state s and x has value u. (a) indicates that some step is applicable from every nonfinal state. In general, more than one step might be applicable; hence, we are considering non-deterministic processes. However, restriction (c) limits the number of applicable steps to being countable, a technical restriction we need later for some of our results. The effect of taking the step is to put p into state t and to write v into x. A step is considered an atomic, indivisible action in our model. With respect to the variable x, a step involves a read followed by a write -- the read to verify that the transition is applicable and the write to update its value. We term such an action a "test-and-set". This is a generalization of the familiar Boolean semaphores or test-and-set instructions found on many computers.

Restriction (b) formalizes an important assumption that a process be able to respond in some way to anything that might be given to it as input. In other words, if it is possible for a process in state s to access variable x, then there must be a transition from s accessing x for every u ε values(x).

A process is not required to be finite-state, nor to have a finite number of transitions from any state. Later (Theorem 3.7), we will see that countable nondeterminism arises from application of natural combination operations to even deterministic processes. Since we wish to treat single processes and groups of processes uniformly,

we allow the greater generality from the beginning.

Systems of Processes

The way in which processes communicate with other processes and with their environment is by means of their variables. A value placed in a variable is available to anybody who happens to read that variable until it is replaced by a new value. Unlike message-based communication mechanisms, there is no guarantee that anyone will ever read the value, nor is there any primitive mechanism to inform the writer that the value has been read. Thus, for meaningful communication to take place, both parties must adhere to previously-agreed-upon protocols, though we place no restrictions on what kinds of protocols are allowed. Indeed, part of our motivation in defining systems in this way is to give us a formal model in which to study such protocols.

We wish to consider variables accessed by a process or system of processes to be either internal or external. Internal variables are to be used only by the given process or system; thus, some consistency of the values of those variables must be hypothesized, and an initial value must be provided. External variables will not have such consistency requirements. That is, a process or system of processes is to be able to respond to values of these variables other than the ones it most recently left there. Intuitively, the external variables may be accessible to other processes (or other external agents) which could change the values between steps of the given process or system.

More formally, if X is a set of variables, a partial assignment for X is any partial function $f : X \to \bigcup_{x \in X} values(x)$ with $f(x) \in values(x)$ whenever $f(x)$ is defined. If f is defined for all $x \in X$, it is called a total assignment for X. The full specification of a system of processes S has four components: proc(S) is a finite set of processes, ext(S) is a set of external variables, int(S) is a set of internal variables, and init(S) is a total assignment for int(S). S is subject to certain restrictions:

(a) $Ext(S) \cap int(S) = \emptyset$.

(b) For each $p \in proc(S)$, $variables(p) \subseteq ext(S) \cup int(S)$.

If P is a set of processes and X a set of variables, we let $\mathcal{S}(P,X) \stackrel{df}{=} \{S : S$ is a system of processes with $proc(S) \subseteq P$ and $int(S) \cup ext(S) \subseteq X\}$.

Execution Sequences

The execution of a system of processes is described by a set of execution sequences. Each sequence is a list of steps which the system could perform when interleaved with appropriate actions by the external agent.

If A is any set, A^* (A^ω) denotes the set of finite (infinite) sequences of A-

elements. A^{count} denotes $A^* \cup A^\omega$, the set of finite or infinite sequences of elements of A. Length: $A^{count} \to N \cup \{\infty\}$ denotes the number of elements in a given sequence.

Let P be a set of processes and X a set of variables. $E(P,X) \stackrel{df}{=} (steps(P,X))^{count}$ is the domain of sequences used to describe executions of processes and sets of processes over P and X.

To define the allowable execution sequences of a system, we first define the execution sequences for processes and sets of processes.

Let p be a process. An execution sequence for p is a sequence $e \in (oksteps(p,variables(p)))^{count} \subseteq E(p,variables(p))$ for which four conditions hold.

Let $e = ((s_i, p, t_i), (u_i, x_i, v_i))_{i=1}^{length(e)}$.

(a) If length(e) = 0, then $start(p) \cap final(p) \neq \emptyset$.

(b) If length(e) $\neq$ 0, then $s_1 \in start(p)$.

(c) If e is finite, then $t_{length(e)} \in final(p)$.

(d) $t_j = s_{j+1}$ for $1 \leq j < length(e)$.

Finally, exec(p) is the set of execution sequences for p. (Note, for example, that this set is nonempty.) Thus, an execution sequence for a process exhibits consistency for state changes, but not necessarily for variable value changes.

Next we describe the execution of a set of processes. We wish the execution to be fair in the sense that each process either reaches a final state or continues to execute infinitely often; it cannot be "locked out" forever by other processes when it is able to execute. In other words, processes are completely asynchronous and thus cannot influence each other's ability to execute a step. Since no consistency of values of variables will yet be assumed, a simple "shuffle" operation will suffice.

Let A be any set and $b = (b_k)_{k\in K}$ be an indexed set of elements of A^{count}. Shuffle(b) is the set of sequences obtained by taking all of the sequences in b and "merging" them together in all possible ways to form new sequences. Formally, if $n \in N$, then define $[n] = \{1,\ldots,n\}$. If $n = \infty$, then $[n] = N$. Now a sequence $c \in A^{count}$ is in shuffle(b) iff there is a 1-1, onto, partial map $\pi : K \times N \to [length(c)]$ such that (a)-(c) hold.

(a) π is defined for (k,n) iff $n \in [length(b_k)]$.

(b) π is monotone increasing in its second argument.

(c) If $\pi(k,i) = j$, then c_j = the i^{th} element in the sequence b_k.

The shuffle operator is easily extended to an indexed set of subsets of A^{count}, viz. if $B = (B_k)_{k\in K}$, where $B_k \subseteq A^{count}$, then $shuffle(B) \stackrel{df}{=} \bigcup \{shuffle(b) : b = (b_k)_{k\in K}$ and $b_k \in B_k$, $k\in K\}$.

If P is a set of processes, define $\underline{exec(P)} \stackrel{df}{=} shuffle((exec(p))_{p \in P})$.

We now extend our notions of execution sequences to systems of processes.

If X is a set of variables, let $\underline{\mathcal{B}(X)} \stackrel{df}{=} (act(X))^{count}$. Let $b \in \mathcal{B}(X)$, $x \in X$, and f be a partial assignment for X. Latest(b, x, f) is the value left in x after performing the actions in b, assuming x had initial value f(x). We define it recursively on the length of b. If length(b) = 0 then

$$latest(b, x, f) = \begin{cases} f(x) \text{ if } f(x) \text{ is defined;} \\ \text{undefined otherwise.} \end{cases}$$

Now assume $length(b) \geq 1$, and $b = b' \cdot (u, y, v)$ for some $(u, y, v) \in act(X)$.

$$\text{Then } latest(b, x, f) = \begin{cases} v \text{ if } x=y; \\ latest(b', f, x) \text{ if } x \neq y \text{ and } latest(b', f, x) \text{ is defined;} \\ \text{undefined otherwise.} \end{cases}$$

Let X, K be sets of variables, $b \in \mathcal{B}(X)$, and f a total assignment for K. We say b is (K,f)-consistent if for every prefix $b' \cdot (u, y, v)$ of b with $y \in K$, then $u = latest(b', y, f)$. For sets of action sequences $B \subseteq \mathcal{B}(X)$, define

$$\underline{consist_{K,f}}(B) \stackrel{df}{=} \{b \in B : b \text{ is } (K,f)\text{-consistent}\}.$$

Let P be a set of processes. A sequence of steps $e \in E(P,X)$ is (K,f)-consistent provided erase(e) is (K,f)-consistent, where erase : $E(P,X) \to \mathcal{B}(X)$ is a homomorphism mapping each pair $(a,b) \in steps(P,X)$ to its second member b. For sets of execution sequences $E \subseteq E(P,X)$, define $\underline{consist_{K,f}}(E) \stackrel{df}{=} \{e \in E : e \text{ is } (K,f)\text{-consistent}\}$. Now let S be a system of processes. $\underline{Exec(S)} \stackrel{df}{=} consist_{int(S),init(S)}(exec(proc(S))) \subseteq E(proc(S), ext(S) \cup int(S))$. Thus, exec(S) consists of those execution sequences of the system's processes in which the internal variables are consistent across the sequence.

Behavior Sequences

Exec(P) gives complete information on how a set of processes might execute in any given environment. Often, however, one is not interested in how the processes execute but only in their effect on the environment, that is, the way they change the variables. We obtain this information from the execution sequences by extracting the variable actions.

If S is a system of processes, we define the behavior of S, $\underline{beh(S)} \stackrel{df}{=} erase(exec(S)) \subseteq \mathcal{B}(ext(S) \cup int(S))$.

Similarly, we define the behavior for a process p and a set of processes P.

$$\underline{Beh(p)} \stackrel{df}{=} erase(exec(p)).$$

$$\underline{Beh(P)} \stackrel{df}{=} erase(exec(P)).$$

One might be interested in only these actions involving the external variables. Let X, K be sets of variables, $b \in \mathcal{B}(X)$, then $\underline{elim_K(b)}$ is the subsequence of b consisting of the actions not involving variables in K. ($Elim_K(b)$ might be finite even if b is infinite.) We define the external behavior of S,

$$\underline{extbeh(S)} \stackrel{df}{=} elim_{int(S)}(beh(S)).$$

The following proposition demonstrates the use of some of the preceding notation, and shows an elementary implication of the read-anything property ((b) in the definition of a process).

Proposition 2.1:

Let p be a process, $x \in variables(p)$, $(v_i)_{i=1}^{\infty}$ any infinite sequence of elements of values(x). Then for some $(w_i)_{i=1}^{\infty}$, it is the case that

$$b = (v_i,x,w_i)_{i=1}^{length(b)}$$

is in $elim_{variables(p)-\{x\}}(beh(p))$. (That is, there is some possible execution of p for which the sequence of values read from x is given by $(v_i)_{i=1}^{\infty}$ or some prefix thereof.)

Proof sketch. By repeated use of the read-anything property.

□

Operations on Systems

One goal of our formalism is to permit complex systems to be understood in terms of simpler ones. For this, we need some operations for building larger systems from smaller ones. Corresponding to these operations will be operations on execution sequences and behaviors. This approach is similar to that of Milne and Milner [3].

The first operation joins a finite collection of systems into a single one. Let $(S_i)_{i\in I}$ be a finite indexed family of systems such that

(a) $i \neq j$ implies $proc(S_i) \cap proc(S_j) = \emptyset$.

(b) $i \neq j$ implies $int(S_j) \cap (int(S_j) \cup ext(S_j)) = \emptyset$.

Then $\circledast (S_i)_{i\in I}$ is the system S such that

$$proc(S) = \bigcup_{i\in I} proc(S_i),$$

$$ext(S) = \bigcup_{i\in I} ext(S_i),$$

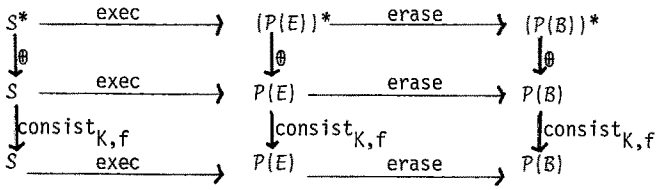
$$\text{int}(S) = \bigcup_{i\in I} \text{int}(S_i),$$
$$\text{init}(S) = \bigcup_{i\in I} \text{init}(S_i).$$

We define Φ for finite indexed families of execution sets and indexed families of behaviors to be simply the shuffle operation.

The second operation on systems is the one of turning selected external variables into internal ones. Let S be a system, K a set of variables and f a total assignment for K such that $K \cap \text{int}(S) = \emptyset$. We define $\text{consist}_{K,f}(S)$ to be the system S' such that $\text{proc}(S') = \text{proc}(S)$, $\text{ext}(S') = \text{ext}(S) - K$, $\text{int}(S') = \text{int}(S) \cup K$, and $\text{init}(S') = \text{init}(S) \cup f$. $\text{Consist}_{K,f}$ has already been defined for execution sets and behaviors.

That these definitions all make sense together is shown by the following.

Theorem 2.2. The following diagram commutes. ($\mathcal{P}$ denotes the power set operator.)



Here we assume a fixed set P of processes and X of variables, and we let $S = S(P,X)$, $E = E(P,X)$, and $B = B(X)$.

We omit the straightforward but tedious proof.

Modules

The two operations of Φ and $\text{consist}_{K,f}$ are sufficient to build any system from one-process systems in a simple way.

Let S, S' be systems. S' is a module of S if $\text{proc}(S') \subseteq \text{proc}(S)$, $\text{ext}(S') \subseteq \text{ext}(S) \cup \text{int}(S)$, $\text{int}(S') \subseteq \text{int}(S)$, and $\text{init}(S') = \text{init}(S)/\text{int}(S')$ (the restriction of the function init(S) to domain int(S')). Thus, a module is a subsystem whose internal variables are private to it and whose external variables form the interface between the module and the remaining system and/or the external world.

S is partitioned into modules $(S_m)_{m\in M}$ if M is finite, S_m is a module of S for each $m\in M$, $(\text{proc}(S_m))_{m\in M}$ is a partition of proc(S), and for all $m, n \in M$, if $m \neq n$, then $\text{int}(S_m) \cap (\text{int}(S_n) \cup \text{ext}(S_n)) = \emptyset$.

A system S is atomic if it consists of a single process with no internal variables, i.e. if $|\text{proc}(S)| = 1$ and $\text{int}(S) = \text{init}(S) = \emptyset$.

The following propositions are immediate from the definitions.

Proposition 2.3. Every system can be partitioned into a finite number of atomic modules.

Proposition 2.4. Every system can be obtained from an arbitrary partition into modules by one application of Φ followed by one application of $consist_{K,f}$ for appropriate K,f.

Remarks on Communication Mechanisms

The basic communication mechanism in our model is the availability of the last written value. We work at a "primitive" level, not basing communication on "messages" as do Hoare [2], Feldman [7] and Milne and Milner [3]. Some message models involve an implicit queuing mechanism or implicit process synchronization, neither of which we wish to assume as basic. Both of these mechanisms involve significant implementation cost and require cost analysis in terms of a more primitive common basis. Neither mechanism seems to be universal in the sense that the most efficient programs for arbitrary tasks would always be written using it. Moreover, the abstraction of automatic process synchronization serves to hide the asynchrony of the basic model. Since we wish to understand asynchronous behavior, we prefer not to mask it at the primitive levels of our theory.

The generality of our process and execution sequence definitions assumes possible indivisibility of a fairly powerful form of variable access. In particular, processes that can both read and change variables in one indivisible step (such as the "test-and set" processes of Cremers-Hibbard [8] and Burns et al [9] are included in the general definitions. Some readers may consider this general access mechanism to be unreasonably powerful, arguing that a process model based on indivisibility of "reads" and "writes" only is more realistic. Such a process model can be defined by certain restrictions on our general model (as we describe below). Thus, our development not only specializes to include consideration of a read-write model, but also allows comparison of the power of the read-write model with that of the more general access model. The specialization can be carried out as follows.

A process p is called a read-write process provided for each $s \in states(p)$ and each $x \in variables(p)$, the set

$$oksteps(p) \cap \{((s,p,t),(u,x,v)) : t \in states(p),\ u,\ v \in values(x)\}$$

can be partitioned into a collection of subsets $\mathcal{T}$, with each $T \in \mathcal{T}$ satisfying (at least) one of the following.

(a) (T describes a "read".)
For all $((s,p,t),(u,x,v))$ in T, it is the case that $u = v$. Moreover, for each $u \in values(x)$ there exists t with $((s,p,t),(u,x,u))$ in T.

(b) (T describes a "write".)
For some t,v, $T = \{((s,p,t),(u,x,v)) : u \in values(x)\}$.

Two very simple examples follow.

Example 2.5. Let states(p) = start(p) = {s}, final(p) = ∅, variables(p) = {x}, values(x) = {0,1}, and oksteps(p) = {((s,p,s),(0,x,1)), ((s,p,s),(1,x,0))}. Process p simply examines x repeatedly, changing its value at each access. The change is clearly an activity that involves both reading and writing, so that, intuitively, p is not a read-write process. Formally, if p were a read-write process, oksteps(p) would be partitionable as above. No T can describe a read since it is never the case that u = v. So ((s,p,s),(0,x,1)) is in T for some T which describes a write. But then ((s,p,s),(1,x,1)) is in T ⊆ oksteps(p) as well, a contradiction.

Example 2.6. Let states(p) = start(p) = {s}, final(p) = ∅, variables(p) = {x}, and oksteps(p) = {((s,p,s),(0,x,1)), ((s,p,s),(1,x,1))}. Process p simply examines x repeatedly, writing "1" every time. It is easy to see that p is a read-write process.

So far, our model describes asynchronous processes communicating by shared variables, a situation which suggests that the processes are physically located sufficiently near to each other to share memory without delay. We also wish to model more general "distributed" systems of asynchronous processes, in which communication is done by means of a channel with significant transmission delay. No new primitives are required in order to extend the present model to handle such communication. A one-way channel is simply modelled by a special "channel process" p, as detailed below.

Example 2.7. Let V be any set, states(p) = {write} × V ∪ {read}, start(p) = {read}, final(p) = ∅, variables(p) = {x,y}, values(x) = values(y) = V, and oksteps(p) = {((read,p,(write,v)),(v,x,v)) : u,v ∈ V} ∪ {(((write,u),p,read),(v,y,u)) : u,v ∈ V}. Process p is thought of as sharing a variable with each of two other processes. It alternately reads from one of the variables and writes the value read in the other variable. (p is obviously a read-write process.) When p is combined with two processes at its ends in the manner already described in this section, the consistent execution sequences exactly describe the effect of an arbitrary-delay channel used for communication between the two original processes.

3. Characterizations and Elementary Examples of Behaviors

The principal justification for a formalism for describing distributed systems is that techniques can thereby be developed for specifying requirements for their operation. It should be possible to determine whether a particular system is a satisfactory realization of the specified requirements. Typical requirements might involve exclusion, fairness, synchronization and other logical correctness properties; they can also involve performance and efficiency.

Requiring that a system exhibit exactly a specified set of execution sequences is generally too strong. For instance, if p_1 and p_2 are processes with $\text{exec}(p_1) \subseteq \text{exec}(p_2)$, then p_1 is always an adequate replacement for p_2. In contrast to the usual

assumptions about nondeterminism, in the case of asynchronous systems all possible nondeterministic choices should be "correct". Thus, a system exhibiting any subset of the specified execution sequences should be acceptable. (Recall that a process cannot have an empty set of execution sequences.)

The subset requirement above is still stronger than one would necessarily want. We are not generally interested in requiring that the complete detail of the specified execution sequences be exhibited by an implementing system, but rather only certain abstracted aspects. Such aspects might be of two different types. One possibility is to specify state reachability requirements as in Cremers-Hibbard [8] and Burns et al [9]. A second possibility, appropriate for specifying processes or groups of processes to be used as modules in larger systems, is to specify external behavior. That is the type of specification we emphasize in this paper.

Monotonicity and the Adequate Replacement Property

Let S_1, S_2 be systems. Define $S_1 \sqsubseteq S_2$ iff $extbeh(S_1) \subseteq extbeh(S_2)$. We call the partial order "$\sqsubseteq$" on systems the adequate replacement order, for we argue that S_1 should always be an adequate replacement for S_2, at least for the purpose of determining logical correctness of input-output behavior.

Proposition 3.1. Φ, $consist_{K,f}$, and $elim_K$ as operations on $P(\mathcal{B})$ preserve inclusion of subsets of $\mathcal{B}$.

Proof: Immediate. □

Proposition 3.2.

(a) Let $(B_i)_{i\in I}$ be an indexed family of sets, where each $B_i \subseteq \mathcal{B}(X)$, K any set of variables.
Then $Elim_K(\Phi(B_i)_{i\in I}) = \Phi\,(Elim_K(B_i))_{i\in I}$.

(b) If K, K' are disjoint sets of variables, f a total assignment for K, $B \subseteq \mathcal{B}(X)$; then $Elim_{K'}(Consist_{K,f}(B)) = Consist_{K,f}(Elim_{K'}(B))$.

Proof: Immediate. □

Proposition 3.3.

Φ and $Consist_{k,f}$ as operations on S preserve $\sqsubseteq$.

Proof: By Theorem 2.2 and Propositions 3.1 and 3.2. □

It follows that if a module S_1 of a system S is replaced by an adequate replacement S_1', then the resulting system S' is an adequate replacement for S.

Equivalence of Systems

We can also define an equivalence among systems based only on their input-output behaviors. While this equivalence is still too strong for many purposes, it nevertheless is not so strong as notions of equivalence based on simulation.

Let S_1, S_2 be systems. Define $S_1 \equiv S_2$ iff extbeh(S_1) = extbeh(S_2). (Thus, $S_1 \equiv S_2$ iff $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$.)

We now proceed to show that any system is equivalent to an atomic system. Thus, groups of processes and single processes can be treated uniformly, an indication of the usefulness of our model for modular design of systems.

We give the construction in two parts. First we show how to reduce the number of processes to one, then we show how to eliminate the internal variables.

Lemma 3.4. For any system S, there is a system S' with the same external and internal variables such that $|proc(S')| = 1$ and beh(S') = beh(S).

Proof Sketch. By induction on $|proc(S)|$. For instance, given a system of two processes p_1 and p_2, we must define a single process p whose behavior is exactly the shuffle of those of p_1 and p_2. The first obvious idea might be to allow states of p to represent pairs consisting of states of p_1 and p_2. Transitions could be composed naturally from the transitions of p_1 and p_2, essentially allowing either one. The only problem is that nothing prevents the nondeterministic choice from always choosing to simulate one process over the other, violating the fairness of the shuffle operation. However, the countable branching capability of processes can be used to enforce fairness. When p begins simulating one of p_1, p_2, it nondeterministically chooses an integer ≥ 1 representing the number of steps p will simulate for that process before shifting to the other process.

□

A process p is called treelike provided (a) and (b) hold.

(a) For all $t_0 \in$ states(p), $|\{((s,p,t),(u,x,v)) \in$ oksteps(p) : $t = t_0\}| \leq 1$.

(b) For all $t_0 \in$ start(p), $|\{((s,p,t),(u,x,v)) \in$ oksteps(p) : $t = t_0\}| = 0$.

Lemma 3.5. If p is a process, there exists a treelike process q with Beh(p) = Beh(q).

Proof Sketch. Process p can be "opened up into a tree" by replicating states; process q has states corresponding to finite paths in p.

□

Theorem 3.6. For any system S, there is an atomic system S' such that $S' \equiv S$.

Proof Sketch. By Lemma 3.4, we can assume proc(S) = {p}. By Lemma 3.5, we can assume p is treelike. A process transformation is carried out in two steps, the intermediate result of which need not be a process. First, p_1 is constructed from p by "pruning"

p's tree so that only (K,f)-consistent paths remain, where K = int(S) and f = init(S). Since p is treelike, there will be no ambiguity involved in deciding when to prune. Now p_2 is constructed from p_1 by condensing paths involving variables in K. This construction is not carried out in stages because of the possible condensation of infinite paths to finite paths. The possibility that p_1 could continue forever on branches involving only variables in K involves transition to a final state of p_2. Finally, S' is the atomic system such that proc(S') = $\{p_2\}$ and ext(S') = ext(S).

□

Unbounded Nondeterminism

We argue that it is natural to use countable nondeterminism for the basic process model. Restriction to finitely many states would surely be unnatural, ruling out processes which resemble natural sequential computation models such as Turing machines. But the usual models, though having infinitely many states, are restricted to finite nondeterminism. This restriction does not seem overly strong in more conventional settings, since it is preserved by natural sequential combination operations. But for the asynchronous parallel case, the finite-branching property would not be preserved by our combination and internalizing constructions. The next result implies that any behavior of a process can be realized as the external behavior of a pair of communicating finite-branching processes. Since behaviors realizable by finite-branching processes form a proper subset of those realizable by all processes (as we show by Example 3.9), uniformity requires at least countable nondeterminism.

More precisely, a process p is finite branching provided start(p) is finite, and also for any s ε nonfinal(p), x ε variables(p), u ε values(x), there are only finitely many t, v with ((s,p,t),(u,x,v)) ε oksteps(p). A system S is finite branching if every process in proc(S) is finite branching. In the following theorem, let p denote the process of Example 2.6. Process p is finite branching and finite state. Assume variables(p) = {x}, and f(x) = 0.

Theorem 3.7. Let S be a system of processes, p ∉ proc(S). Then there exists an atomic finite branching system S_1 such that $S \equiv consist_{\{x\},f}(S_1 \circledast S_p)$, where S_p is the fixed atomic system with proc(S_p) = {p}, ext(S_p) = {x}, and int(S_p) = init(S_p) = ∅.

Proof Sketch. By Theorem 3.6, we can assume that S is atomic. Let proc(S) = {q}. For each s ε states(q), y ε variables(q), u ε values(y), there are only countably many pairs (v,t) such that ((s,q,t),(u,y,v)) ε oksteps(q). Some ordering is fixed for each such set of pairs. An ordering is also fixed for the elements of start(q). Process q_1 simulates a step of process q as follows. Process q_1 alternately tests x and increments a counter until it sees that x has been set to 1. It then uses the value of its counter to select one of the possible alternatives of q to simulate and resets the counter and variable x to 0 in preparation for the next step of simulation. S_1

then is the system with $proc(S_1) = \{q_1\}$, $ext(S_1) = ext(S) \cup \{x\}$, $int(S_1) = init(S_1) = \emptyset$. □

We conclude this section with an example of a set of sequences which can be obtained as the behavior of a process, but not of any finite-branching process.

Lemma 3.8. Let p be a finite-branching process, $x \in variables(p)$, $b \in (act(x))^\omega$. If $beh(p)$ contains infinitely many prefixes of b, then $b \in beh(p)$.

Proof Sketch. By a König's Lemma-style argument. □

Example 3.9: Intuitively, we consider the specification to "write a value any finite number of times."

Let x be a variable, $v \in values(x)$, $A = \{(u,x,v) : u \in values(x)\}$. A^* is the set of all finite sequences of actions, each of which "writes v" into x. A^* can easily be realized as $beh(p)$ for a process p which uses countable nondeterminism to choose an element of N for a counter initialization. Process p alternately decrements the counter and writes v, halting when the counter is 0.

On the other hand, Lemma 3.8 implies that A^* is not $beh(p)$ for any finite-branching process p, since $b = (v,x,v)^\omega$ has all of its finite prefixes in A^*.

4. Examples

In this section, we discuss behavior specification for a typical distributed system - an arbiter. (A similar treatment has been worked out for a ticket distribution system, but space limitations preclude inclusion of the details of this second example.) We also describe particular and diverse implementations within our model that realize this behavior. We do not espouse any particular formal specification language, but rather express behavior restrictions in general mathematical terminology.

The specification example follows a pattern which has more general applicability, so we first describe that pattern. A finite set X of variables is accessed by a "user" and by a "system". The user is required to follow a simple and restrictive behavior pattern; formally, a set $U \subseteq \mathcal{B}(X)$ of "correct user sequences" is defined. The system is to be designed so that when it is combined with a user exhibiting correct behavior, with correct initialization of variables, certain conditions (on the values of variables) hold. Formally, a set $M \subseteq (\{user,system\} \times act(X))^{count}$ is defined in order to describe the desired conditions. A total assignment f for X is defined in order to describe correct initialization of variables.

In a sense, U, M and f may together be regarded as a specification for the behavior of the desired system: any $b \in \mathcal{B}(X)$ can be considered "acceptable" if whenever it is combined consistently with a sequence in U, the resulting combination is in M. A system of processes is a correct implementation if all of its external behavior

sequences are acceptable.

More formally, if A is any set, $t \in A^{count}$, L any set, x any element of L, then $\underline{t^x}$ denotes that element of $(\{x\} \times A)^{count}$ whose i^{th} element is (s,t_i), where t_i is the i^{th} element of t. (That is, the entire sequence is labelled by x.) This superscript operator is extended to subsets of A^{count} in the obvious way.

For X, K sets of variables, L any set, $t \in (L \times act(X))^{count}$, f a total assignment for K, we say that t is (K,f)-consistent provided the sequence of second components of t is (K,f)-consistent.

In the present examples, L is taken to be {user,system}, a set of identifying labels for the modules of interest.

A sequence $b \in B(X)$ is called <u>(U,M,f)-acceptable</u> provided $\{c \in shuffle(U^{user}, b^{system}) : c \text{ is } (X,f)\text{-consistent}\} \subseteq M$. Then a system of processes S would be considered to be a correct implementation provided every sequence in extbeh(S) is (U,M,f)-acceptable.

However, this type of description may be somewhat difficult for a system designer to use as a specification, so that it may be helpful to define explicitly a set B of (U,M,f)-acceptable sequences. Any system of processes S with $extbeh(S) \subseteq B$ is then considered correct. B should be as large as possible so as not to constrain the system designer unnecessarily. In the following example, we are able to obtain B exactly equal to the set of (U,M,f)-acceptable sequences, thus providing an explicit correctness characterization. We do not yet have a general equivalence theorem for specifications, however.

<u>Example 4.1: Arbiter</u>

Values(x) = {E,A,G} for each $x \in X$. Intuitively, E indicates "empty", A indicates "ask" and G indicates "grant" of a resource. The user is restricted simply to initiating requests and returning granted resources. More precisely, $U \subseteq B(X)$ is defined as follows.

(Let $a \in shuffle(\{a_x : x \in X\})$, where each $a_x \in B(x)$.

<u>$a \in U$</u> iff for each $x \in X$, (a)-(c) hold

(Let $a_x = (u_i,x,v_i)_{i=1}^{length(a_x)}$.)

<u>(a) Correct Transitions</u>

For all i, $1 \leq i \leq length(a_x)$, if $u_i = E$ then $v_i \neq G$, and if $u_i = A$ then $v_i = u_i$.

(The user cannot grant a request, and once he has initiated a request he cannot retract it.)

(b) Stopping

If a_x is finite and nonempty, then $v_{length(a_x)} = E$. (The user cannot leave the system when a request is pending or granted.)

(c) Return of Resource

For all i, if $u_i = G$ then there exists $j \geq i$ with $v_j \neq G$.

(If the user sees that his request has been granted, he must eventually return the resource.)

■

Thus, user correctness is defined locally at each variable. In particular, the user can consist of separate processes, one for each variable, with no communication between them. It is easy to design various sets of processes with behavior a subset of U.

Correct operation for our arbiter system will require that all requests eventually be granted, and that no two requests be granted simultaneously. Of course, variants on these conditions could be specified instead.

Let $f = \lambda x[E]$, $L = \{user, system\}$. $M \subseteq (L \times act(X))^{count}$ is defined as follows. $c \in M$ iff c is (X,f)-consistent and both (a) and (b) hold.

(a) Local Conditions

(Let $c \in shuffle(\{c_x : x \in X\})$, each $c_x \in (L \times act(x))^{count}$.)

For each $x \in X$, both (a1) and (a2) hold.

(Let $c_x = (\ell_i,(u_i,x,v_i))_{i=1}^{length(c_x)}$.)

(a1) Correct Transitions

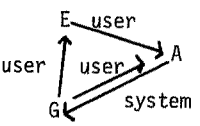
For all i, $1 \leq i \leq length(c_x)$, either $u_i = v_i$ or else one of (a11)-(a13) holds.

(a11) ℓ_i = user, $u_i = E$ and $v_i = A$.

(a12) ℓ_i = user and $u_i = G$.

(a13) ℓ_i = system, $u_i = A$, $v_i = G$.

(The allowed transitions are depicted at right.)



(a2) Progress

For all i, if $v_i \neq E$ then there exists $j \geq i$ with $v_j \neq v_i$.

(Any value other than E is eventually changed.)

(b) Global Conditions

(Let $c = (\ell_i,(u_i,x_i,v_i))_{i=1}^{length(c)}$, $d = (u_i,x_i,v_i)_{i=1}^{length(c)}$.)

(b1) Mutual Exclusion

For no $x_1,x_2 \in X$, $x_1 \neq x_2$ and no prefix e of d is it the case that

$latest(e,x_1,f) = latest(e,x_2,f) = G$.

■

Next, we define B.

$b \in B$ iff either (a) or (b) holds.

(a) Initialization or User Observed to be Incorrect

(Let $b \in shuffle(\{b_x : x \in X\})$ as before.)

For some $x \in X$, one of (a1)-(a3) holds.

(Let $b_x = (u_i,x,v_i)_{i=1}^{length(b_x)}$.)

(a1) $u_1 = G$.

(a2) For some i, $v_i = E$ and $u_{i+1} = G$, or else $v_i = A$ and $u_{i+1} \neq A$.

(a3) length $(b_x) = \infty$, and $u_i = G$ for all sufficiently large i.

(Thus, a sequence is "correct" if it involves incorrect action on the part of the user or an incorrect initialization of the variables. It is the job of the system designer to discover how such errors can be detected during system operation. It is easy to program a system to check for errors such as those represented in (a1) and (a2), but (a3) errors could not be detected at any finite point during the computation. However, the system is required to obey some conditions involving infinite execution sequences. It is possible to allow some of the system's "eventual" behavior to wait for the user's "eventual" behavior. An example will be seen in Implementation 1.)

(b) Correctness Conditions

Both (b1) and (b2) hold.

(b1) Local Conditions

(Let $b \in shuffle(\{b_x : x \in X\})$ as before.)

For each $x \in X$, (b11)-(b13) all hold.

$(b_x = (u_i,x,v_i)_{i=1}^{length(b_x)}.)$

(b11) Correct Transitions

For all i, if $u_i = E$ or G, then $v_i = u_i$, and if $u_i = A$, then $v_i = A$ or G.

(b12) Infinite Examination

b_x is infinite

(b13) Response

For all i, if $u_i = A$, then for some $j \geq i$ it is the case that $v_j \neq A$.

(b2) Global Conditions

(Let $b = (u_i,x_i,v_i)_{i=1}^{length(b)}$.)

(b21) Mutual Exclusion

For no $x_1,x_2 \in X$, $x_1 \neq x_2$, and no prefix d of b it is the case that

$latest(d,x_1,f) = latest(d,x_2,f) = G$. ∎

The following theorem shows that our explicit characterization for system behavior is as general as possible.

Theorem 4.2: For U,M,f,B of this example, B = {b : b is (U,M,f)-acceptable}.

Proof: $\subseteq$: Let $b \in B$, $a \in U$, $c \in shuffle(a^{user},b^{system})$, c (X,f)-consistent. We must show $c \in M$.

Since $a \in U$ and c is (X,f)-consistent, we can show that b fails to satisfy (a) of (the definition of) B. Thus, b satisfies (b) of B.

We check that c satisfies each condition of M. c satisfies (a1) of M because of (a) of U and (b11) of B. To verify (a2) of M, write $c \in shuffle(\{c_x : x \in X\})$, and for fixed x, write $c_x = (\ell_i,(u_i,x,v_i))_{i=1}^{length(c_x)}$. If $(\ell_i,(u_i,x,A))$ is an element of c, then (b12) and (b13) of B together imply that for some $j > i$, $v_i \neq A$. If $(\ell_i,(u_i,x,G))$ is an element of c, then let j be the largest number $\leq i$ with ℓ_j = user. By (b11) of B, j exists and $v_j = A$ or G. Then by (b) of U, there exists $k > i$ with ℓ_k = user. If $u_k \neq G$ we are done. Otherwise, (c) of U implies that for some $m \geq k$, $v_m \neq G$.

(b1) of M follows easily from (b21) of B and (a) of U.

$\supseteq$: Let $b \notin B$. We must produce $a \in U$, $c \in shuffle(a^{user},b^{system})$, c (X,f)-consistent, and $c \notin M$. Clearly, b fails to satisfy (a) of B. In addition, b will fail to satisfy at least one of (b11), (b12), (b13) and (b21) of B.

We consider four cases.

(b11) fails: Any $a \in U$, $c \in shuffle(a^{user},b^{system})$ which is (X,f)-consistent will fail to satisfy (a1) of M. One such c can be constructed by immediately preceding each element (system,(u,x,v)) of c which is derived from an action of b by an element (user,(y,x,u)). The value of y is uniquely determined by the consistency requirements on c; since b fails to satisfy (a) of B, this determination produces $a \in U$.

(b12) fails: Consider x such that actions (u,x,v) only appear finitely often in b.

Construct a $\in$ U, c $\in$ shuffle(a^{user},b^{system}), c (X,f)-consistent, with the following property. In c, following all elements of the form (system, (u,x,v)) (for any u,v), there is an element of the form (user,(u,x,A)) (for some u), and following that element there are infinitely many elements of the form (user,(A,x,A)). Such a, c can be constructed by a slight addition to the construction for the preceding case. The resulting c fails to satisfy (a2) of M.

(b13) fails: Consider x such that (A,x,A) occurs in b and moreover for all following actions in b of the form (u,x,v), we have v = A.

Then any a $\in$ U, c $\in$ shuffle(a^{user},b^{system}) which is (X,f)-consistent will fail to satisfy (a2) of M. Such a, c can be constructed as before.

(b21) fails: Let $b = (u_i,x_i,v_i)_{i=1}^{length(b)}$, where (u_j,x_j,G) and (u_k,x_k,G) are actions witnessing the contradiction to (b21) of B. We can assume that $j < k$, $x_j \neq x_k$ and for no m, $j < m < k$ it is the case that $x_m = x_j$.

Construct a $\in$ U, c $\in$ shuffle(a^{user},b^{system}), c (X,f)-consistent, with the following property. In c, the elements derived from b's actions (u_j,x_j,G) and (u_k,x_k,G) have no intervening elements of the form (user, (u,x_j,v)) for any u, v. Such a, c fail to satisfy (b1) of M.

Such a, c can be constructed as before.

□

The given description of B seems sufficiently manageable to be used to specify system behavior. B is also sufficiently general to admit many different implementations - i.e. processes or communicating groups of processes with behavior a subset of B but with very different internal structure and execution behavior. Outlines of three such examples follow.

Implementation 1: The simplest implementation is a single process p which polls each variable in circular sequence. When A is read, p changes it to G and then repeatedly reads that variable until its value reverts either to E or A. When this occurs, p resumes polling with the next variable.

Note that p may fail to examine some variable after some time, contradicting (b12) of the definition of B. But the only way this can occur is if the user acts incorrectly, failing, for example, to change G to E or A. Then the execution will satisfy, for example, (a3) of the definition of B. Thus, although p does not actually detect certain incorrect user behavior, it nevertheless can cause its own correct eventual behavior to depend on the eventual correctness of user behavior.

Checking that beh(p) $\subseteq$ B is straightforward.

Implementation 2: The idea of Implementation 1 can be extended to allow "more con-

currency" using a binary tree of polling processes, with the leaves accessing the interface variables $x \in X$.

Each non-root process p alternately polls its left and right son variables. When A is seen, p changes its own father variable to A. When the father variable changes to G, p grants its pending son's request by changing the appropriate A to G. p then waits for that son variable to revert to either E or A. When this occurs, p changes its father variable to E and then resumes polling its sons with the other son being polled next.

The root process acts just like p of Implementation 1 for $|X| = 2$.

One must do a little work to convince oneself that the alternating strategy guarantees eventual granting of all requests. All other properties in the definition of B are easy to check, if all father variables are assumed to be initialized at E.

Implementation 3: The third implementation is based on the state-model algorithms used in Burns et al [9], (see also Cremers-Hibbard [8]). This time, the implementing system consists of identical processes p_x, each of which has access to exactly one of the interface variables. In addition, there is a common variable x* to which all the processes p_x have access. One of the algorithms from [9], such as algorithm A, is used. This algorithm enables asynchronous processes requiring mutual exclusion synchronization to communicate using x* to achieve the needed synchronization, with good bounds on the number of times any single process might be bypassed by any other (and with a very small number of values for x*). The processes themselves must be willing, however, to execute a complicated protocol. In the present development, we have defined a very simple arbiter protocol and do not require a user to learn the more complicated protocol of the earlier algorithm. We can still use the earlier ideas, however, by isolating the earlier protocol in the system processes and allowing a user to communicate with one of those processes.

In outline, and referring to some ideas from algorithm A, the p_x accessing x examines x until A is detected. Then p_x enters the trying protocol using x*. When p_x is allowed (in algorithm A) to enter its critical region, it passes the permission on by changing the value of x to G. p_x then examines x until it reverts to E or A, and then p_x enters the exit protocol using x*. When p_x has completed its exit protocol, it is ready to begin once again, examining x for further requests.

Correctness of the resulting system of communicating processes is easy to understand based on that of Algorithm A.

□

The main point to be made by this example is that there are many different processes and systems of processes which can meaningfully be said to realize the same input-output behavior. In the three implementations above, the systems vary both in process configuration and in execution. There is no realistic sense in which the

internal states and transitions (i.e. the execution sequences) of the different implementations could be thought to simulate each other. And yet, they are all solutions to the problem of constructing an arbiter.

A technical question which may be of interest for the purpose of obtaining a sequence-based characterization for behaviors in whether B in the above example is exactly equal to extbeh(S) for some system S. It is not hard to show that U can be so obtained.

5. Complexity Measures

Separation of behavior and implementation opens the way for comparison of different implementations of the same behavior, a fundamental subject of study for any theory of computation. Intuitively, comparisons might be made on the basis of process configuration, local process space requirements, communication space requirements, number of local process steps executed, number of changes made to variables, and possible "amount of concurrency". Tradeoffs would be expected.

Configuration and space measures seem easy to formalize. For instance, the three implementations in Example 4.1 use 1, n-1 and n processes, 0, n-2 and 1 auxiliary communication variables, and 0, 3 and n+5 values for each communication variable, respectively.

In contrast, time and concurrency measures are not so straightforward. For instance, "response time" might be expected (sometimes) to be better for Implementations 2 and 3 than for Implementation 1 of Example 4.1, because of "use of concurrency". But much work remains to be done in quantifying such time comparisons.

In order to state time bounds, one must meet several requirements. First, one must decide what actions to count during execution. Second, in order to state time bounds as closed-form functions (e.g. "runtime = $2n^2$"), one requires an appropriate notion of the "size of the task being accomplished", (i.e. an appropriate parameter n on which to base complexity analysis). Finally, one needs to establish appropriate quantification over alternatives in the present nondeterministic setting. We believe that partial orders of the type studied by Greif [11] and Hewitt [12] will provide useful ways of satisfying the first requirement but do not yet know how best to satisfy the remaining requirements.

In some detail, let X be a set of variables, p a set of processes, $a = ((s_i,p_i,t_i), (u_i,x_i,v_i))_{i=1}^{\text{length}(a)}$ be a sequence of elements of steps(P,X). For $i, j \in N$, define $i\ P'\ j$ iff $i < j$ and either $x_i = x_j$ or $p_i = p_j$. Let $\underline{P}$ be the transitive closure of P'. In words, P formalizes the ordering of steps of a imposed by the sequentiality of each individual process and each variable. P seems to provide much useful information about the "running time" and "possible concurrency" in a, including some seemingly natural formal measures. An important remaining task is the

use of these measures to obtain clean statements of upper and lower complexity bounds, both for particular systems and for the collection of systems realizing particular specified behavior.

REFERENCES

[1] Petri, C.A., "Kommunikation mit Automaten," Schriften des Reinish Westfalischen Inst. Instrumentelle Mathematik, Bonn. 1962.

[2] Hoare, C.A.R., "Communicating Sequential Processes," Technical Report, Department of Computer Science, the Queen's University, Belfast, Northern Ireland, December, 1976.

[3] Milne, G. and R. Milner, "Concurrent Processes and Their Syntax," Internal Report CSR-2-77, Department of Computer Science, Edinburg, May, 1977.

[4] Dijkstra, E.W., "Co-operating Sequential Processes," Programming Languages, NATA Advanced Study Institute, Academic Press, 1968.

[5] Campbell, R. and A. Habermann, "The Specification of Process Synchronization Using Path Expressions," Lecture Notes in Computer Science, 16, Springer-Verlag, 1974.

[6] Shaw, A.C., "Software Descriptions with Flow Expressions," IEEE Trans. on Software Engineering SE-4, 3 (1978), 242-254.

[7] Feldman, J., "Synchronizing Distant Cooperating Processes," Technical Report 26, Department of Computer Sciences, University of Rochester, October, 1977.

[8] Cremers, A. and T. N. Hibbard, "Mutual Exclusion of N Processes Using an O(N) - Valued Message Variable," USC Department of Computer Science Manuscript, 1975.

[9] Burns, J.E., M. J. Fischer, P. Jackson, N.A. Lynch, and G. L. Peterson, "Shared Data Requirements for Implementation of Mutual Exclusion Using a Test-and-Set Primitive," Proceedings of 1978 International Conference on Parallel Processing (1978).

[10] Chandra, A.K., "Computable Nondeterministic Functions," Proceedings of 19th Annual Symposium on Foundations of Computer Science, 1978.

[11] Greif, Irene, "A Language for Formal Problem Specification," Comm. ACM, 20, 12 (1977), 931-935.

[12] Atkinson, R. and C. Hewitt, "Specification and Proof Techniques for Serializers," AI Memo 438, Massachusetts Institute of Technology, August, 1977.