Theoretical Computer Science 13 (1981) 17–43

ON DESCRIBING THE BEHAVIOR AND IMPLEMENTATION OF DISTRIBUTED SYSTEMS*

Nancy A. LYNCH

Georgia Institute of Technology, Atlanta, GA 30332, U.S.A.

Michael J. FISCHER

University of Washington, Seattle, WA 98195, U.S.A.

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 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 model's primitivity and generality make it a suitable basis for cost comparison of distributed system implementations.

1. Introduction

A distributed computing system consists of a number of distinct and logically separated communicating asynchronous sequential processes. In order to understand such systems, one would like simple mathematical models which exhibit the essential features of these systems while abstracting away irrelevant details. Such models would 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 sets, sequences, functions, and relations, rather than axiomatic models consisting of lists of desired properties of systems.

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 analyzing and

* This research was supported in part by the National Science Foundation under grants MCS77-02474, MCS77-15628, MCS 3-01689 and by U.S. Army Research Office Contract Number DAAG29-79-C-0155.

comparing different implementations of desired system behavior, using objective complexity measures.

The following five steps summarize a standard method of operation in complexity theory as it is usually applied to sequential computation:

Step 1: Choose a computing model. Deterministic and nondeterministic Turing machines, random-access machines, straight-line programs, finite automata and pushdown automata are all popular.

Step 2: Choose a problem. Mathematical functions are typical examples. Another class of examples is provided by data bases with various storage and retrieval properties.

Step 3: Define what it means for an instance of the model to 'solve' the problem. Conventions for input and output, as well as for treatment of nondeterminism, are needed to determine whether a device computes a particular function or returns correct answers to data base queries. This determination has nothing to do with the internal structure of the device, but depends only on the input/output behavior.

Step 4: Choose complexity measures. The number of steps executed by a machine is usually taken to represent the time complexity. Space complexity is usually measured by the amount of work tape used, or by the largest number calculated during the computation. Other measures of interest describe the structure of the device—its number of states, program size, number of tapes, or alphabet size, for example.

Step 5: Compare solutions and prove upper and lower bounds. The measures are used to compare different solutions to the same problem. Upper bounds are generally proved by exhibiting and measuring a particular solution. Lower bounds are more difficult, since they involve a proof about all possible solutions (within the chosen model).

In this paper, we follow the same sequence of Steps 1–5 for systems of asynchronous parallel processes. The remaining sections are organized as follows.

Section 2 deals with our choice of model. As we have stated, our model is set-theoretic; its style is automata-theoretic rather than (for example) fixpoint style as [11]. Its basic notions are 'process' and 'shared variable'.

No particular internal structure is assumed for the processes. Rather, each process is simply an automaton with a possibly infinite number of internal states and a set of possible transitions. We expect that often it will be useful to impose additional structure in order to describe particular systems. However, use of the more general model strengthens lower bound and other negative results.

Processes are permitted to exhibit infinitely-branching nondeterminism. This is because we want to treat systems of processes uniformly with single processes, using composition operations to construct larger systems from component processes and systems, and describing the behavior of the larger system in terms of the behavior of the components. Since a system of two deterministic processes can exhibit infinite nondeterminism, we include this capability for single processes as well. (Thus, this assumption is made not so much in order to model systems realistically, but rather for economy and elegance of the model.)

We assume that each process takes a step from time to time, but we make no assumptions on how long it waits between steps except that the time is finite – the process does not wait forever. Thus, mechanisms that depend on timing considerations for their correct operation are ruled out. 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 appears to compensate for the limitations imposed on it. We make assumptions about time in Section 6 for the purpose of evaluating system running time complexity, but these assumptions are never used for determining system correctness. Eventually, of course, mechanisms that depend on time for their correct operation should be studied via a suitable formal model.

The shared variable is our basic (and only) communication mechanism. Thus, we do not assume any primitive synchronization mechanism such as is implicit in Petri nets [12] or in the communicating sequential processes of Hoare [9] and of Milne and Milner [11]. Neither do we permit messages or queuing mechanisms as do Feldman [7] and Atkinson and Hewitt [1]. All of these mechanisms involve significant implementation cost and we are interested in examining these costs. None of these mechanisms seems to us 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 asynchronism 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 shared variable seems to be universal, to reflect closely many aspects of physical reality, and to be sufficiently basic to allow problems of communication and synchronization to be studied.

Because of the popularity of message-based distributed systems and a possible immediate reaction that a 'central' shared memory does not constitute true distribution, some words about this choice are in order. At the most primitive level, something must be shared between two processors for them to be able to communicate at all. This 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. (We are not specifying the protocols to be used by the sending and receiving processes which enable communication to take place, since 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 the share variable, respectively. Thus, 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 us consider various possible message primitives. Perhaps the simplest is to assume each process has a 'mailbox' [15] 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 [9, 11] 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.

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 [6]. They all become just special cases of our general model.

One might object to the use of shared variables to model the long-distance communication needed in distributed systems: changes to a shared variable are instantaneous, while long-distance communication has an inherent delay. However, communication with delay can be modelled simply within our framework by a pair of shared variables, joined by a 'channel process' which copies values from one to the other. The arbitrary process delay assumed in our model then appears as an arbitrary communication delay.

In general, the environment of a process (or system of processes) can change a variable at any time; thus, a process (or system) will not necessarily see the same value in a variable that it most recently wrote there. We require of the process (or

system) that it be complete in that it exhibit at least one possible response for *every possible way* the environment might behave.

Operations are given for combining processes and systems into larger systems.

Section 3 deals with our choice of problem. 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 non-deterministic, so one must allow in general for several different outputs to a given sequence of inputs, all of which must be considered 'correct'.

Therefore, we take as our notion of 'problem', an arbitrary set of finite and infinite sequences of 'variable actions', where a variable 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.

Section 4 deals with defining what it means for an instance of the model to solve a problem. We define the behavior of a distributed system to be a set of finite and infinite sequences of interleavings of possible variable actions 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 variable actions by the system, assuming particular variable actions by the environment.

We say that a particular system is a solution to a particular problem if the systems's behavior is *any subset* of the problem. (Trivial cases such as the empty set are ruled out as system behaviors by our process and system definitions.) 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.

It is shown that operations for combining systems and operations for combining problems are related naturally.

In Section 5, we digress from our sequence of steps to present basic characterization results for behaviors. Two theorems are given, as evidence that our process

and behavior definitions are neither too restrictive nor too general. Theorem 5.1 says that any behavior exhibited by any system of processes is also exhibited by a single process; thus our definitions are sufficiently general. Theorem 5.2, on the other hand, says that any behavior exhibited by a process is also exhibited by a system of two deterministic processes; thus, if our definitions were to be restricted, we would be sacrificing either necessary generality or uniformity.

In Section 6, we define several complexity measures appropriate for distributed systems. It is not obvious how best to define time measures for asynchronous systems; we use an interesting time measure first described in [13]. Other important measures describe communication space (bandwidth), local storage space and system structure.

Finally, in Section 7, we give an example of a prototypical distributed problem (an arbiter) and several very different systems which solve it. The solutions are compared using our cost criteria. Equivalent implicit and explicit specifications are given for the arbiter problem.

We do not address in this paper an important aspect of problem specification, namely, what is an appropriate formal language for describing the sets of sequences that comprise a problem specification? The arbiter example is described informally in standard mathematical notation. We expect the work on path expressions [3], flow expressions [14], and other formal systems of expressions might be applicable here.

This paper is part of a larger project on Theory of Asynchronous Parallel Processes. Other papers completed include [2], a study of communication space complexity requirements for implementation of mutual exclusion, [8], a study of space complexity requirements for resource allocation, allowing restricted types of process failure, and [10], a study of a fast resource-allocation algorithm for distributed systems.

2. A model for distributed systems

2.1. Processes and shared variables

The primitive notions in our model are those of 'process' and 'variable'. The interaction among system components occurs at the process-variable interface.

A *variable* x has an associated set (finite or infinite) 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 values(X) $\stackrel{\text{df}}{=} \bigcup_{x \in X}$ values(x) and act(X) $\stackrel{\text{df}}{=} \bigcup_{x \in X}$ act(x).

A *process* p has an associated set (finite or infinite) of *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 $s \in \text{nonfinal}(p)$, $t \in \text{states}(p)$; intuitively, it represents the action of p going from its old state s to its new state t in a single step. (s and t are not required to be distinct.) $\text{Act}(p)$ is the set of all process actions for p. If P is a set of processes, we let $\text{states}(P) \stackrel{\text{df}}{=} \bigcup_{p \in P} \text{states}(p)$ and $\text{act}(P) \stackrel{\text{df}}{=} \bigcup_{p \in P} \text{act}(p)$.

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 $\text{steps}(P, X) \stackrel{\text{df}}{=} \text{act}(P) \times \text{act}(X)$ be the set of execution steps. To specify which steps are permitted in a computation, a process has two other components in its description. $\text{Variables}(p)$ is a set of variables which the process p can access. If P is a set of processes, then $\text{variables}(P) \stackrel{\text{df}}{=} \bigcup_{p \in P} \text{variables}(p)$. $\text{Oksteps}(p)$ is a subset of $\text{steps}(p, \text{variables}(p))$ describing the permissible steps of p; $\text{oksteps}(p)$ is subject to three restrictions:

(a) for any $s \in \text{nonfinal}(p)$, there exist t, u, x, v with $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$. Furthermore, if $((s, p, t), (u, x, v))$ and $((s, p, t'), (u', x', v')) \in \text{oksteps}(p)$, then $x = x'$;

(b) *read anything*: if $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$ and $u' \in \text{values}(x)$, then there exist t', v' with $((s, p, t'), (u', x, v')) \in \text{oksteps}(p)$.

(c) *countable nondeterminism*: $\text{start}(p)$ is countable, and also for any $s \in \text{nonfinal}(p)$, $x \in \text{variables}(p)$ and $u \in \text{values}(x)$, there are only countably may pairs t, v with $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$.

Some intuitive remarks are in order. $\text{Oksteps}(p)$ represents the allowable steps of p. A particular step $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$ 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, and that the next variable accessed is determined by the state. In general, more than one step might be applicable; hence, we are considering nondeterministic 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 to be an atomic, indivisible action. 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 $u \in \text{values}(x)$.

A process is not required to be finite-state, nor to have a finite number of transitions from any state. In Section 5, we show that countable nondeterminism arises from application of natural combination operations to deterministic processes. Since we wish to treat single processes and groups of processes uniformly, we allow the greater generality from the beginning.

We would like next to describe the execution of a process, by describing how the steps are to be combined. Intuitively, it is clear that successive steps of a process should be consistent in their state components – the new state at any step should coincide with the old state at the next step. However, the corresponding consistency condition for variables is more complicated and involves interaction between several processes. Therefore, we defer discussion of process execution until after we have described systems of processes.

2.2. 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.)

We 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, consistency of the values of those variables must be hypothesized, and an initial value must be provided. External variables will not have 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 (resp. processes), a *partial assignment* for X is any partial function $f: X \to \text{values}(X)$ (resp. $\text{states}(X)$) with $f(x) \in \text{values}(x)$ (resp. $\text{states}(x)$) whenever $f(x)$ is defined. If f is defined for all $x \in X$, it is called a *total assignment* for X. A *system of processes* S has four components: $\text{proc}(S)$ is a countable set of processes, $\text{ext}(S)$ is a set of *external variables*, $\text{int}(S)$ is a set of *internal variables*, and $\text{init}(S)$ is a total assignment for $\text{int}(S)$. S is subject to certain restrictions:

(a) $\text{ext}(S) \cap \text{int}(S) = \emptyset$;

(b) for each $p \in \text{proc}(S)$, $\text{variables}(p) \subseteq \text{ext}(S) \cup \text{int}(S)$.

If P is a set of processes and X a set of variables, we let

$$\mathcal{S}(P, X) \stackrel{\text{def}}{=} \{S: S \text{ is a system of processes with } \text{proc}(S) \subseteq P \text{ and } \text{ext}(S) \cup \text{int}(S) \subseteq X\}.$$

2.3. 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.

Let N denote the set of natural numbers, including zero.
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: $A^{\text{count}} \to \mathrm{N} \cup \{\infty\}$ denotes the number of elements in a given sequence.

Let P be a set of processes and X a set of variables. $\mathscr{E}(P, X) \stackrel{\text{df}}{=} \text{steps}(P, X)^{\text{count}}$ 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

$$e \in (\text{oksteps}(p))^{\text{count}} \subseteq \mathscr{E}(p, \text{variables}(p))$$

for which four conditions hold. Let $e = ((s_i, p, t_i), (u_i, x_i, v_i))_{i=1}^{\text{length}(e)}$:

(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 $t_{\text{length}(e)} \in$ final(p);
(d) $t_j = s_{j+1}$ for $1 \leq j <$ length(e).

Let exec(p) denote 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 has yet been assumed, a simple 'shuffle' operation will suffice.

Let A be any set, K any countable 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 $[n] = N$. A sequence $c \in A^{\text{count}}$ is in shuffle(b) iff there is a bijective 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 ith 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 $B_k \subseteq A^{\text{count}}$, then

$$\text{shuffle}(B) \stackrel{\text{df}}{=} \cup \{\text{shuffle}(b) : b = (b_k)_{k \in K} \text{ and } b_k \in B_k,\ k \in K\}.$$

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

We now extend our notions of execution sequences to systems of processes. In doing so, we want to insist on consistency for internal variable values, but not necessarily for external variable values.

If X is a set of variables, let $\mathcal{B}(X) \stackrel{\text{df}}{=} (\text{act}(X))^{\text{count}}$. ($\mathcal{B}$ stands for 'behavior'.) 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 'latest' recursively on the length of b. If $\text{length}(b) = 0$, then

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

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

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

Let X, Y be sets of variables, $b \in \mathcal{B}(X)$, and f a total assignment for Y. We say b is (Y, f)-consistent if for every prefix $b'(u, y, v)$ of b with $y \in Y$, it is the case that $u = \text{latest}(b', y, f)$. For $B \subseteq \mathcal{B}(X)$, define

$$\text{consist}_{Y,f}(B) \stackrel{\text{df}}{=} \{b \in B : b \text{ is } (Y, f)\text{-consistent.}\}$$

Let P be a set of processes, X a set of variables. Then erase : $\text{steps}(P, X) \to \text{act}(X)$ maps each pair (a, a') to its second component a'. Erase is extended to a homomorphism mapping $\mathcal{E}(P, X)$ to $\mathcal{B}(X)$. (Similar extensions of notation are used later in the paper.) For $E \subseteq \mathcal{E}(P, X)$, define

$$\text{consist}_{Y,f}(E) \stackrel{\text{df}}{=} \{e \in E : e \text{ is } (Y, f)\text{-consistent}\}.$$

Now let S be a system of processes.

$$\text{Exec}(S) \stackrel{\text{df}}{=} \text{consist}_{\text{int}(S),\text{init}(S)}(\text{exec}(\text{proc}(S)))$$

$$\subseteq \mathcal{E}(\text{proc}(S), \text{ext}(S) \cup \text{int}(S)).$$

Thus, exec(S) consists of those execution sequences of the system's processes in which the internal variables are consistent throughout the sequence. Note that $\text{exec}(S) \neq \emptyset$.

2.4. Operations on systems and on sets of sequences

One goal of our formalism is to permit complex systems to be understood in terms of simpler ones. For this, we need operations for building larger systems from smaller ones. Corresponding to these operations will be operations on related sets of sequences. This approach is similar to that of Milne and Milner [11].

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

(a) $i \neq j$ implies $\text{proc}(S_i) \cap \text{proc}(S_j) = \emptyset$;

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

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

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

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

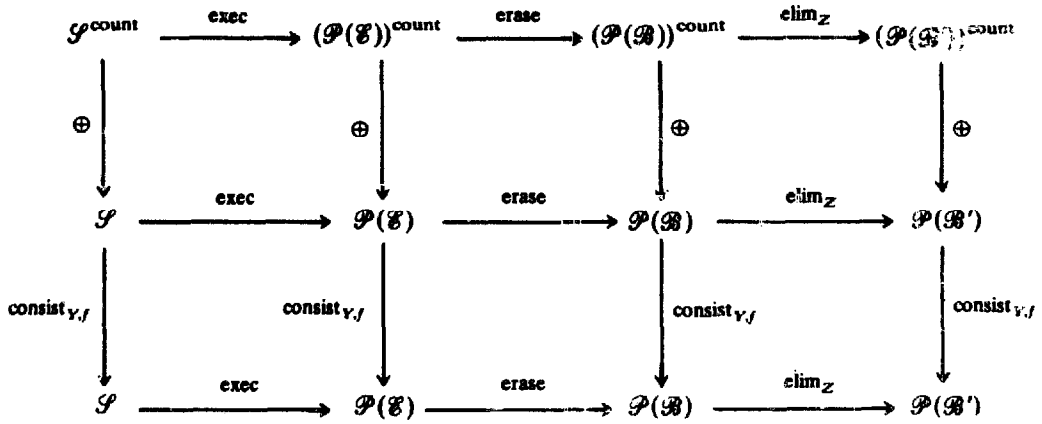
Let P be a set of processes, X a set of variables. Then $\oplus$ is defined on countable indexed families of subsets of $\mathscr{E}(P, X)$ or $\mathscr{B}(X)$ to be simply the shuffle operation.

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

In Section 4, we will be interested in restricting attention to a subset of a system's variables rather than all of its variables. Thus, if X and Y are sets of variables, $b \in \mathscr{B}(X)$, then $\text{restr}_Y(b)$ (resp. $\text{elim}_Y(b)$) is the subsequence of b consisting of the actions involving (resp. not involving) variables in Y. ($\text{elim}_Y(b)$ might be finite even if b is infinite.)

That these definitions interact properly is shown by the following theorem. ($\mathscr{P}$ denotes the power set operator.)

Theorem 2.1. *Let P be a set of processes, X, Y and Z sets of variables with $Y \cap Z = \emptyset$, f a total assignment for Y. Let $\mathscr{S}$, $\mathscr{E}$, $\mathscr{B}$ and $\mathscr{B}'$ denote $\mathscr{S}(P, X)$, $\mathscr{E}(P, X)$, $\mathscr{B}(X)$ and $\mathscr{B}(Z)$ respectively. Then the following diagram commutes:*



Proof. Straightforward.

2.5. Modules

The two operations $\oplus$ and $\text{consist}_{Y,f}$ 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 $\text{init}(S)$ to domain $\text{int}(S')$). Thus, a module is a subsystem whose internal variables are private to it and whose external variables form the interface between the module and both the remaining system and the external world.

A system S is *partitioned into modules* $(S_i)_{i \in I}$ if S_i is a module of S for each $i \in I$, $(\text{proc}(S_i))_{i \in I}$ is a partition of $\text{proc}(S)$, and for all $i, j \in I$, if $i \neq j$, then $\text{int}(S_i) \cap (\text{ext}(S_j) \cup \text{int}(S_j)) = \emptyset$.

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

The following is immediate from the definitions.

Theorem 2.2. (a) *Every system can be partitioned into a countable set of atomic modules.*

(b) *Every system can be obtained from an arbitrary partition into modules by one application of* $\oplus$ *followed by one application of* $\text{consist}_{Y,f}$ *for appropriate* Y, f.

2.6. Remarks on indivisibility of variable access

Our process and execution sequence definitions assume 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 [3] and Burns et al. [2] 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 $s \in \text{states}(p)$, the set

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

has (at least) one of the following properties:

(a) (A describes a 'read operation') for all $((s, p, t), (u, x, v))$ in A, it is the case that $u = v$;

(b) (A describes a 'write operation') if $((s, p, t), (u, x, v))$ and $((s, p, t'), (u', x, v'))$ are two arbitrary elements of A, then $t = t'$ and $v = v'$.

Two simple examples follow.

Example 2.1. Let $\text{states}(p) = \text{start}(p) = \{s\}$, $\text{final}(p) = \emptyset$, $\text{variables}(p) = \{x\}$, $\text{values}(x) = \{0, 1\}$, and $\text{oksteps}(p) = \{((s, p, s), (0, x, 1)), ((s, p, s), (1, x, 0))\}$. 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, it is obvious that neither (a) nor (b) holds.

Example 2.2. Let states(p) = start(p) = $\{s\}$, final(p) = $\emptyset$, variables(p) = $\{x\}$, values(x) = $\{0, 1\}$ 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.

2.7. Communication delay

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.

Let V be any set, states(p) = $(\{\text{write}\} \times V) \cup \{\text{read}\}$, start($p$) = $\{\text{read}\}$, final(p) = $\emptyset$, variables(p) = $\{x, y\}$, values(x) = values(y) = V, and oksteps(p) = $\{((\text{read}, p, (\text{write}, v)), (v, x, v)): u, v \in V\} \cup \{(((\text{write}, u), p, \text{read}), (v, y, u)): u, v \in V\}$. 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 in the manner already described in this section, the consistent execution sequences exactly describe the effect of an arbitrary-delay one-way channel used for communication between the two original processes. Different types of communication channels similarly can be modelled, using corresponding types of channel processes.

3. Distributed problems

In the introduction, we listed four factors making a satisfactory notion of 'distributed problem' more complicated than just a function. We now give a definition that accommodates those factors.

Let X be a set of variables. A *distributed problem* over X is any subset B of $\mathcal{B}(X)$.

The explicit mention of different variables models the multiplicity of input/output sites. The inclusion of finite and infinite sequences in $\mathcal{B}(X)$ models terminating and nonterminating computations. The use of totally ordered sequences of actions preserves the relative ordering of remote events. And finally, the use of subsets (rather than single sequences) recognizes the inherent nondeterminism of distributed systems.

Typical distributed problems consist of sequences restricted by conditions of exclusion, synchronization and fairness.

4. Solutions to distributed problems

4.1. System behavior

Exec(S) gives complete information on how a system S 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,

$$\text{beh}(S) \stackrel{\text{df}}{=} \text{erase}(\text{exec}(S)) \subseteq \mathscr{B}(\text{ext}(S) \cup \text{int}(S)).$$

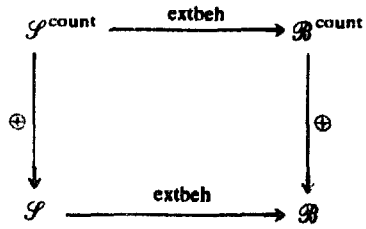
Similarly, we define the *behavior* for a process p and a countable set of processes P:

$$\text{Beh}(p) \stackrel{\text{df}}{=} \text{erase}(\text{exec}(p)) \subseteq \mathscr{B}(\text{variables}(p)),$$
$$\text{Beh}(P) \stackrel{\text{df}}{=} \text{erase}(\text{exec}(P)) \subseteq \mathscr{B}(\text{variables}(P)).$$

Often one is interested only in those actions involving the external variables. We define the *external behavior* of S, $\text{extbeh}(S) \stackrel{\text{df}}{=} \text{elim}_{\text{int}(S)}(\text{beh}(S))$.

The following two results relates extbeh to $\oplus$ and $\text{consist}_{\gamma,f}$.

Theorem 4.1. *Let P be a set of processes, X a set of variables. Let $\mathscr{S}$ and $\mathscr{B}$ denote $\mathscr{S}(P, X)$ and $\mathscr{B}(X)$ respectively. The following diagram commutes:*



Proof. Let $(S_i)_{i\in I} \in \mathscr{S}_{\text{count}}$. Then

$$\begin{aligned}
\text{extbeh}(\oplus(S_i)_{i\in I}) &= \text{elim}_{\text{int}(\oplus(S_i)_{i\in I})}(\text{beh}(\oplus(S_i)_{i\in I})) \\
&= \oplus(\text{elim}_{\text{int}(\oplus(S_i)_{i\in I})}(\text{beh}(S_i)))_{i\in I} \quad \text{by Theorem 2.1} \\
&= \oplus(\text{elim}_{\text{int}(S_i)}(\text{beh}(S_i)))_{i\in I} \quad \text{by domain considerations} \\
&= \oplus(\text{extbeh}(S_i))_{i\in I}.
\end{aligned}$$

Theorem 4.2. *Let S be a system of processes, Y a set of variables for which $Y \cap \text{int}(S) = \emptyset$, f a total assignment for Y. Then*

$$\text{extbeh}(\text{consist}_{Y,f}(S)) = \text{elim}_Y(\text{consist}_{Y,f}(\text{extbeh}(S))).$$

Proof.

$$\begin{aligned}
\text{extbeh}(\text{consist}_{Y,f}(S)) &= \text{elim}_{\text{int}(S)\cup Y}(\text{beh}(\text{consist}_{Y,f}(S))) \quad \text{by definition} \\
&= \text{elim}_Y(\text{elim}_{\text{int}(S)}(\text{beh}(\text{consist}_{Y,f}(S)))) \\
&= \text{elim}_Y(\text{elim}_{\text{int}(S)}(\text{consist}_{Y,f}(\text{beh}(S)))) \quad \text{by Theorem 2.1} \\
&= \text{elim}_Y(\text{consist}_{Y,f}(\text{elim}_{\text{int}(S)}(\text{beh}(S)))) \quad \text{by Theorem 2.1} \\
&= \text{elim}_Y(\text{consist}_{Y,f}(\text{extbeh}(S))).
\end{aligned}$$

4.2. Realization and solution

Let S be a system of processes, $B \subseteq \mathcal{B}(\text{ext}(S))$. Then S *realizes* B provided $\text{extbeh}(S) = B$. This definition might at first appear to capture the conditions under which system S solves the distributed problem B. However, we think that a weaker definition is more appropriate; we say S *solves* B provided $\text{extbeh}(S) \subseteq B$.

Thus, S is not required to exhibit the same degree of nondeterminism as B. Intuitively, B is the set of acceptable input/output sequences; S's input/output sequences must be included among those in B, but they need not encompass all of B. Trivial cases such as $\text{extbeh}(S) = \emptyset$ are ruled out by the definitions for systems and their executions; for instance, recall that $\text{exec}(S)$ cannot be empty.

4.3. Equivalence and substitution

Let S_1, S_2 be systems of processes. Then $S_1 \equiv S_2$ (S_1 is equivalent to S_2) provided $\text{extbeh}(S_1) = \text{extbeh}(S_2)$. This definition might at first appear to capture the conditions under which S_1 might be allowed to replace S_2 as a module in a larger system. Indeed, this style of definition is used for such a purpose in [11]. However, we think, as before, that a weaker definition is appropriate: we say $S_1 \sqsubseteq S_2$ (S_1 is *substitutable* for S_2) provided $\text{extbeh}(S_1) \subseteq \text{extbeh}(S_2)$.

Thus, S_1 is not required to exhibit the same degree of nondeterminism as S_2. In contrast with the usual assumptions about nondeterminism, in the case of asynchronous systems *all possible* nondeterministic choices should be correct. Thus, a system S_1 exhibiting any subset of the execution sequences of S_2 should be acceptable.

Appropriate interactions between our operations and relations are shown in the following theorem.

Theorem 4.3. *Let P be a set of processes, X and Y sets of variables, f a total assignment for Y, and let $\mathcal{S}$ and $\mathcal{B}$ denote $\mathcal{S}(P, X)$ and $\mathcal{B}(X)$ respectively. Then the following hold:*

(a) $\oplus$, $\text{consist}_{Y,f}$, restr_Y *and* elim_Y *as operations on* $\mathcal{P}(\mathcal{B})$ *preserve* $\subseteq$,

(b) $\oplus$ *and* $\text{consist}_{Y,f}$ *as operations on* $\mathcal{S}$ *preserve* $\sqsubseteq$,

(c) $\oplus$ *and* $\text{consist}_{Y,f}$ *as operations on* $\mathcal{S}$ *preserve* $\equiv$,

Proof. (a) is obvious. For (b), the case of $\oplus$ follows from Theorem 4.1, while the case of $\text{consist}_{Y,f}$ follows from Theorem 4.2. (c) is immediate from (b).

Theorem 4.3 implies the following. Assume S_1 is substitutable for S_2, and S_2 is a module of system T. Let T_1 be the system obtained by replacing S_2 by S_1 in T_1. Then T_1 is substitutable for T_2. Similarly, if S_1 is equivalent to S_2, then T_1 is equivalent to T_2. Thus, our external behavior definitions provide a way of describing the behavior of a system in terms of the behavior of its components.

5. Countable nondeterminism

The results of this section justify the 'countably nondeterministic' generality of our process definition. Theorem 5.1 says that the countable nondeterminism of a system (caused by arbitrary interleaving of process steps as well as the nondeterminism of the individual processes) can be simulated by the countable nondeterminism of a single process. More precisely, any external behavior realized by a system is also realized by an atomic system. Thus, our definition is general enough to permit uniform treatment of single processes and groups of processes. On the other hand, Theorem 5.2 says that the countable nondeterminism of a process can be simulated by the countable nondeterminism of the interleaving of steps in a system of two deterministic processes. More specifically, any external behavior realized by an atomic system is also realized by a system of two deterministic processes; thus, our definition could not reasonably be made more restrictive. These results are closely related to some of those of Chandra [3]. (However, our sequences have considerably more structure than his, so that we do not obtain the type of explicit characterization of the class of process behaviors that he does.)

We first show how to reduce the number of processes to one.

Lemma 5.1. *For any system S, there is a system S' with the same external and internal variables such that* $|\text{proc}(S')| = 1$ *and* $\text{beh}(S') = \text{beh}(S)$.

Proof. We describe a process p which simulates the fair execution of all of the processes in $\text{proc}(S)$. The most obvious idea is to allow p repeatedly to use countable nondeterminism to select which process $q \in \text{proc}(S)$ to simulate next, and at the same time use its countable nondeterminism to select among the countably many possible

alternatives of q. The only problem is that nothing insures that p will actually simulate infinitely many steps of each q; p might always choose to simulate one particular process, thereby violating the fairness of the shuffle operation. However, the countable nondeterminism of p can also be used to enforce fairness, as follows. Let $\pi : N - \{0\} \to \text{proc}(S)$ be a fixed total function such that each $q \in \text{proc}(S)$ appears infinitely many times in the range of π. Process p keeps a partial function schedule : $N - \{0\} \to \text{proc}(S)$ in its state, representing the process which has been or is to be simulated at each step of p. At any moment during p's computation, schedule is a finite function. Also, p keeps config, a partial assignment to proc(S), representing the current states of each process which has had a step simulated by p. Config is also finite at any particular moment. The other principal datum in p's state is currentstate, the current state of the next process to be simulated. Each start state of p contains an initialization of schedule as $\{(1, q)\}$, for some $q \in \text{proc}(S)$, an initialization of currentstate as s, for some $s \in \text{start}(q)$, and an initialization of config as $\emptyset$. At each step n of p, p does the following: (Let q denote the value of schedule(n) and s the value of currentstate at the beginning of step n of p. Let x denote the unique variable to be accessed from state s, u the value of variable x at the beginning of step n. The CHOOSE command represents a use of p's countable nondeterminism; the remaining syntax should be self-explanatory.)

```
CHOOSE (t, v) such that ((s, q, t), (u, x, v)) ∈ oksteps(q);
x := v;
config(q) := t;
IF schedule(n + 1) is undefined THEN
  [CHOOSE q' ∈ proc(S); schedule(n + 1) := q']
IF config(schedule(n + 1)) is defined THEN
  currentstate := config(schedule(n + 1))
  ELSE [CHOOSE s' ∈ start (schedule(n + 1)); currentstate := s'];
CHOOSE k such that schedule(k) is undefined;
schedule(k) := π(n)
```

That is, p uses its countable nondeterminism to select from among the moves of the simulated processes, to select the next process to simulate (if necessary), to select start states for the simulated processes, and to select specific steps (of p) when each process in proc(S) will be guaranteed of being simulated.

We leave to the reader the task of translating the code into a process in our formal model. (All of the countably nondeterministic choices must be done at once in the translated version. If schedule($n + 1$) is undefined, then q' is chosen, and if config(q') is then undefined, the code contains a second choice for a start state of q'. It might appear at first that these two choices must be done sequentially; however, it is easy to make a single choice of (q', s'), where $s' \in \text{start}(q')$.)

Let S' be the system with $\text{proc}(S') = \{p\}$, $\text{ext}(S') = \text{ext}(S)$, $\text{int}(S') = \text{int}(S)$ and $\text{init}(S') = \text{init}(S)$. That $\text{beh}(S') = \text{beh}(S)$ follows from the fact that p simulates exactly all of the fair interleavings of steps of the processes in proc(S).

Next, we prove a technical lemma producing a standard form for processes.
A process p is called *treelike* provided (a) and (b) hold:
(a) for all $t_0 \in \text{states}(p)$, $|\{((s, p, t), (u, x, v)) \in \text{oksteps}(p): t = t_0\}| \leq 1$,
(b) for all $t_0 \in \text{start}(p)$, $|\{((s, p, t), (u, x, v)) \in \text{oksteps}(p); t = t_0\}| = 0$.

Lemma 5.2. *If p is a process, then there is a treelike process q with* $\text{beh}(p) = \text{beh}(q)$.

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

It remains to remove internal variables.

Theorem 5.1. *For any system S, there is an atomic system S' such that $S' \equiv S$.*

Proof sketch. By Lemma 5.1, we can assume $\text{proc}(S) = \{p\}$. By Lemma 5.2, 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 $(\text{int}(S), \text{init}(S))$-consistent paths remain. Since p is treelike, there is no ambiguity involved in deciding when to prune. Now p_2 is constructed from p_1 by condensing paths involving variables in $\text{int}(S)$. 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 $\text{int}(S)$ involves transition to a final state of p_2. Finally, S' is the atomic system such that $\text{proc}(S') = \{p_2\}$ and $\text{ext}(S') = \text{ext}(S)$.

We argue that our countably nondeterministic process model is not too general.

Restriction of processes to finitely many states would surely be unnatural, ruling our processes which resemble natural sequential computation models such as Turing machines. But the usual sequential computation models, though allowing infinitely many states, are restricted to finite nondeterminism. This restriction does not seem overly strong in the sequential setting, since it is preserved by natural sequential combination operations. But for the asynchronous parallel case, finite nondeterminism would not be preserved by fair combination operations such as our $\oplus$. The next result says that the external behavior of *any* system can be realized as the external behavior of a pair of communicating deterministic (and therefore finitely nondeterministic) processes. However, Example 5.1 below shows that the set of external behaviors realizable by atomic systems of finitely nondeterministic processes is a proper subset of the set of external behaviors realizable by arbitrary systems.

More precisely, a process p is *finite branching* (resp. *deterministic*) provided $\text{start}(p)$ is finite, (resp. of cardinality 1), and also for any $s \in \text{nonfinal}(p)$, $x \in \text{variables}(p)$, $u \in \text{values}(x)$, there are only finitely many (resp. at most 1) pairs (t, v) with $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$.A system S is finite branching (resp. deterministic) if every process in $\text{proc}(S)$ is finite branching (resp. deterministic).

In the following theorem, let p denote the process of Example 2.2. Process p is deterministic and finite state. Assume variables$(p) = \{x\}$, and $f(x) = 0$. Let T be the atomic system with proc$(T) = \{p\}$, ext$(T) = \{x\}$, and int$(T) =$ init$(T) = \emptyset$.

Theorem 5.2. *Let S be a system of processes. Then there is a deterministic atomic system S_1 such that $S \equiv \text{consist}_{\{x\},f}(S_1 \oplus T)$.*

Proof sketch. By Theorem 5.1, we can assume that S is atomic. Let proc$(S) = \{q\}$. For each $s \in$ states(q), $y \in$ variables(q), $u \in$ values(y), there are only countably many pairs (t, v) such that $((s, q, t), (u, y, v)) \in$ oksteps(q). Fix an ordering for each set of pairs. Also fix an ordering for the elements of start(q). Process q_1 simulates a step of process q as follows. Process q_1 alternatively tests x and increments a counter until it sees that x has been set to 1. It then simulates a step of q, using the counter value to select one of the possible alternative moves, and then resets the counter and variable x to 0 for the next step of simulation. S_1 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 realized as the external behavior of an atomic system, but not of any finite-branching atomic system.

Lemma 5.3. *Let p be a finite-branching process, $x \in$ variables(p), $b \in (\text{act}(x))^\omega$. If* beh$(p)$ *contains infinitely many prefixes of b, then $b \in$* beh(p).

Proof sketch. By König's Lemma.

Example 5.1. Consider the problem of writing a specific value any finite number of times.

More specifically, let x be a variable, $v \in$ values(x), $A = \{(u, x, v): u \in \text{values}(x)\}$. A^* is the set of all finite sequences of actions, each of which 'writes v' into x. A^* can easily be obtained as beh(p) for a process p which uses countable nondeterminism to choose an element of N for a counter initialization. Process p alternatively decrements the counter and writes v, halting when the counter is 0. Therefore, A^* can be realized as the external behavior of an atomic system.

On the other hand, Lemma 5.3 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^*. Therefore, A^* cannot be realized as the external behavior of any finite-branching atomic system.

6. 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 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; one can simply count numbers of processes and variables, numbers of states and variable values. Time and concurrency measures are not so straightforward. We use a version of a measure described in [13]. Intuitively, fixed upper bounds are assumed for the intervals between the occurrence of certain events. (For instance, each process might be assumed to take a step within c_1 units of time. Also, when the system makes certain changes to an external variable, the environment might be assumed to respond in a specific way within c_2 units of time.) With such assumptions, an upper bound can be proved for the running time of a finite execution sequence. Then an upper bound for the time required for a particular event to occur is just the maximum of the upper bounds on the running times of all possible execution sequences, up to the point when that event occurs. No lower bounds are assumed for the intervals between the occurrence of events. Thus, all fair interleavings of steps are still possible, even with the time assumptions. These assumptions are therefore of no use in proving logical correctness of systems. Their only use is for bounding running time.

7. Example: an arbiter

7.1. Behavior specification method

In this section, we specify behavior for a typical distributed system—an arbiter. We do not here espouse any particular formal specification language, but rather express behavior restrictions in general mathematical terminology.

We also describe three particular and diverse implementations within our model that exhibit (i.e. solve) this behavior. Finally, we compare these implementations using our complexity measures.

The specification 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 $M \subseteq (\{\text{user}, \text{system}\} \times \text{act}(X))^{\text{count}}$ 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, $t \in A^{\text{count}}$, L any set, x any element of L, then b^x denotes that element of $(\{x\} \times A)^{\text{count}}$ whose ith element is (x, b_i), where b_i is the ith element of b. (That is, the entire sequence is labelled by x.) This superscript is extended to subsets of A^{count} in the obvious way.

For X, Y sets of variables, L any set, $b \in (L \times \text{act}(X))^{\text{count}}$, f a total assignment for Y, we say that b is (Y, f)-consistent provided the sequence of second components of b is (Y, 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 $b \in \mathcal{B}(X)$ is called *(U, M, f)-acceptable* provided $\{c \in \text{shuffle}(U^{\text{user}}, b^{\text{system}})$: c is (X, f)-consistent$\} \subseteq M$. Then a system of processes S is a correct implementation provided S 'solves' $\{b$: b is (U, M, f)-acceptable$\}$ (that is, provided every sequence in extbeh(S) is (U, M, f)-acceptable).

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 that solves 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.

7.2. Arbiter specification

Let 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 to initiating requests and returning granted resources. More precisely, $U \subseteq \mathcal{B}(X)$ is defined as follows: (Let $a \in \text{shuffle}(\{a_x: x \in X\})$, where each $a_x \in \mathcal{B}(X)$.)

$a \in U$ iff for each $x \in X$, (a)–(c) hold (Let $a_x = (u_i, x, v_i)_{i=1}^{\text{length}(a_x)}$.)

(a) *Correct transitions*: For all i, $1 \leq i \leq \text{length}(a_x)$, if $u_i = E$, then $v_i = E$ or A, and if $u_i = A$, then $v_i = A$. (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 $v_{\text{length}(a_x)} = E$. (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 $v_j = E$ or A. (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 a set 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.

Let $f = \lambda x[E]$, $L = \{\text{user}, \text{system}\}$. $M \subseteq (L \times \text{act}(X))^{\text{count}}$ is defined as follows:

$c \in M$ iff c is (X, f)-consistent and both (a) and (b) hold:

(a) *Local conditions*: (Let $c \in \text{shuffle}(\{c_x : x \in X\})$, each $c_x \in (L \times \text{act}(x))^{\text{count}}$.) For each $x \in X$, both (a1) and (a2) hold: (Let $c_x = (l_i, (u_i, x, v_i))_{i=1}^{\text{length}(c_x)}$.)

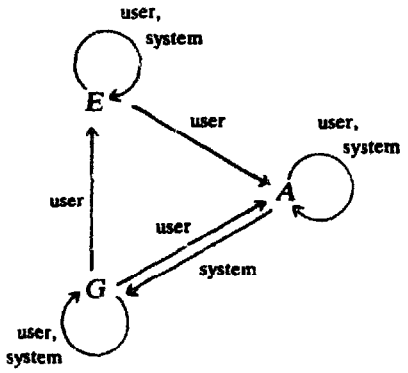
(a1) *Correct transitions*: For all i, $1 \leq i \leq \text{length}(c_x)$, either $u_i = v_i$ or else one of (a11)–(a13) holds:

(a11) $l_i = \text{user}$, $u_i = E$ and $v_i = A$;

(a12) $l_i = \text{user}$ and $u_i = G$;

(a13) $l_i = \text{system}$, $u_i = A$, and $v_i = G$.

(The allowed transitions are depicted below.)



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

(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 $\text{latest}(e, x_1, f) = \text{latest}(e, x_2, f) = G$.

Next, we define B, thereby providing an explicit characterization of the set of correct sequences.

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

(a) *Initialization or user observed to be incorrect*: (Let $b \in \text{shuffle}(\{b_x : x \in X\})$ as before.) For some $x \in X$, one of (a1)–(a3) holds: (Let $b_x = (u_i, x, v_i)_{i=1}^{\text{length}(b_x)}$.)

(a1) $u_1 = G$:

(a2) For some i, it is the case that $v_i = E$ and $u_{i+1} = G$, or else $v_i = A$ and $u_{i+1} = E$ or G;

(a3) length $(b_x) = \infty$, and $u_i = G$ for all sufficiently large i. (Thus, we have not required any particular error detection behavior; we permit arbitrary system behavior if incorrect action by the user or incorrect variable initialization occurs. Note that it would be easy to program a system to check for errors such as those represented in (a1) and (a2), but (a3) errors cannot be detected at any finite point during the computation.)

(*b*) *Correctness conditions*: Both (b1) and (b2) hold:

(b1) *Local conditions*: (Let $b \in \text{shuffle}(\{b_x : x \in X\})$ as before.) For each $x \in X$, (b11)–(b13) all hold. ($b_x = (u_i, x, v_i)_{i=1}^{\text{length}(b_x)}$.)

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

(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 $\text{latest}(d, x_1, f) = \text{latest}(d, x_2, f) = G$.

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

Theorem 7.1. $B = \{b : b \text{ is } (U, M, f)\text{-acceptable}\}$.

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

Since $a \in U$ and c is (X, f)-consistent, it follows 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 $c \in \text{shuffle}(\{c_x : x \in X\})$, and for fixed x, write $c_x = (l_i, (u_i, x, v_i))_{i=1}^{\text{length}(c_x)}$. If $(l_i, (u_i, x, A))$ is an element of c, then (b12) and (b13) of B together imply that $v_i \neq A$ for some $j > i$. If $(l_i, (u_i, x, G))$ is an element of c, then let j be the largest number $\leq i$ with $l_j = \text{user}$. By (b11) of B, j exists and $v_j = A$ or G. Then by (b) of U, there exists $k > i$ with $l_k = \text{user}$. If $u_k \neq G$ we are done. Otherwise, (c) of U implies that $v_m \neq G$ for some $m \geq k$.

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

$\supseteq$: Let $b \notin B$. We must produce $a \in U$, $c \in \text{shuffle}(a^{\text{user}}, b^{\text{system}})$, c (X, f)-consistent, and $c \in M$. 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 $a \in U$, $c \in \text{shuffle}(a^{\text{user}}, b^{\text{system}})$ 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 $a \in U$, $c \in \text{shuffle}(a^{\text{user}}, b^{\text{system}})$, 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 $a \in U$, $c \in \text{shuffle}(a^{\text{user}}, b^{\text{system}})$ which is (X, f)-consistent will fail to satisfy (a2) of M. Such a, c can be constructed as before.

(b21) *fails*: Let $b = (u_i, x_i, v_i)_{i=1}^{\text{length}(b)}$, 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, $j < m < k$ is it the case that $x_m = x_j$.

Consider $a \in U$, $c \in \text{shuffle}(a^{\text{user}}, b^{\text{system}})$, 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 $(\text{user}, (u, x_j, v))$ 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.

7.3. Three solutions and their comparison

The arbiter problem as stated above admits many different implementations—i.e. systems of processes with external behavior a subset of B but with different internal structure and execution behavior. Outlines of three such implementations follow. Complexity bounds are estimated for all of the implementations.

Let $n = |x|$, the number of external variables. Assume c_1 to be an upper bound on the time between steps of each process of the implementation system. Also, assume c_2 to be an upper bound on the time between the granting of a resource and the return of that resource by the user. We calculate upper bounds on the time between the initiation and granting of a request. We calculate similar bounds with the additional restriction that at most k other requests overlap the given request in time.

Implementation 1. The simplest implementation is an atomic system S consisting of 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.

It is obvious that $\text{extbeh}(S) \subseteq B$ (but note that equality does not hold). The single process p has $2n$ states and no internal variables.

The worst case time for a request occurs when a user makes a request just as he returns the resource; he must wait for p to examine all of the other variables, possibly granting the resource to each other user in turn. The upper bound is $c_2(n-1) + 2c_1 n$. If there are at most k other requests active at the same time as the given request, the upper bound is $c_2 k + 2c_1 n$. (Thus, if there are no other active requests, the time to grant the request is bounded by $2c_1 n$.)

Implementation 2. Assume for simplicity that $n = 2^m$, $m \geq 1$. The idea of Implementation 1 can be extended to allow 'more concurrency' 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 and waits. 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 $n = 2$.

All internal (i.e. father) variables are initialized at E. The alternating strategy guarantees eventual granting of all requests. All other properties in the definition of B are easy to check, so that extbeh$(S) \subseteq B$ for this system S. (Once again, equality does not hold.)

The system consists of $n-2$ non-root processes, each with 12 states, and one root process with four states. There are $n-2$ internal variables, each with three values.

If at most k other requests are active at the same time as a given request, the time for granting the given request can be bounded by $c_2k + O(c_1(k+1)\log n)$. (Thus, if there are no other active requests, the time to grant the request is $O(\log n)$. If there is no bound assumed on the number of concurrently active requests, then the time can be bounded by $c_2(n-1) + O(c_1 n \log n)$, as we show using a system of recurrence equations:

Classify the variables of S into *levels*, with the root process's two variables at level 1, the external variables of S at level m, and intermediate levels in the tree numbered consecutively. Let $T(i)$, $1 \leq i \leq m$, denote the longest time between the initiation and granting of a request at level i. Let $R(i)$, $1 \leq i \leq m$, denote the longest time between the granting and return of a resource at level i. Then

$$R(m) = c_2, \qquad R(i) = 4c_1 + R(i+1), \quad 1 \leq i \leq m-1,$$

$$T(1) = 4c_1 + R(1), \qquad T(i) = 12c_1 + R(i) + 2T(i-1), \quad 2 \leq i \leq m.$$

The first two equations are straightforward. The third equation is a special case of an equation in Implementation 1. The fourth equation arises when a process (or user) makes a request just as it returns the resource. The father passes the return up, then polls its other son for a request. If that son has a request, that request is passed up, and must be granted (giving rise to one $T(i-1)$ term). Then the father grants the resource to its other son, and awaits the return (giving rise to the $R(i)$ term). Next, the father passes the return up, returns to poll the original son and finally detects its request. The father passes this request up, waits for it to be granted (giving rise to the second $T(i-1)$ term) and grants the resource.

It is easy to see that

$$R(i) = c_2 + 4c_1(m-i), \quad 1 \leq i \leq m.$$

Thus,

$$T(1) = c_2 + 4c_1 m \quad \text{and} \quad T(i) = 12c_1 + c_2 + 4c_1(m-i) = 2T(i-1), \quad 2 \leq i \leq m.$$

This latter expression is in turn equal to

$$(2^{i-1}-1)(12c_1 + c_2 + 4c_1 m) + (2^{i-1})(c_2 + 4c_1 m) - (2^i + 2^{i-1} - 1 - i)(4c_1),$$

so that the needed bound, $T(m)$, is at most

$$(2^{m-1}-1)(12c_1 + c_2 + 4c_1 m) + (2^{m-1})(c_2 + 4c_1 m) - (2^m + 2^{m-1} - 1 - m)(4c_1)$$
$$= 2^{m+2} c_1 m + 2^m c_2 - 8c_1 - c_2 = 4c_1 (n \log n - 2) + c_2(n-1),$$

as needed.

Implementation 3. The third implementation is based on the state-model algorithms used in [2, 5]. The implementation system S consists of identical processes p_x, each of which has access to exactly one interface variable $x \in X$. In addition, there is a common variable y to which all the processes p_x have access. Algorithm A of [2] is used, for definiteness. This algorithm enables asynchronous processes requiring mutual exclusion synchronization to communicate using y to achieve the needed synchronization, with a small bound on the number of times any single process might be bypassed by any other (and with a very small number of values for y). The processes themselves must be willing, however, to execute a complicated protocol. In this paper, we have defined a very simple arbiter protocol and do not require a user to know the more complicated protocol of Algorithm A. 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), p_x examines x until value A is detected. Then p_x enters the trying protocol of Algorithm A using y as the shared variable. 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 of Algorithm A using y. 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 based on the correctness of Algorithm A. Once again, $\text{extbeh}(S) \subseteq B$.

The system consists of n processes, each with $O(n^2)$ states. There is one internal variable with $n+5$ values. Time can be bounded by $c_2(2n-3) + O(n^2)$ in the worst case. (The first term represents the possibility that the requestor is forced to wait for $2n-3$ distinct returns of the resource. For the second term, note that each time the resource is granted in Algorithm A, an $O(n)$-sized count is transmitted in unary via the shared variable. It would be easy to modify Algorithm A to transmit counts in binary, thereby reducing the second term to $O(n \log n)$ at the cost of a small increase in number of variable values.) If there are at most k concurrent requests, then the bound is $c_2 k + O((k+1)^2)$. (Thus, if there are no other active requests, the time to grant the request is bounded by a constant.)

Thus, 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, the systems are all solutions to the arbiter problem.

Note that the time complexity in the general case is smallest for Implementation 1, whereas Implementations 2 and 3 perform faster if requests are relatively infrequent.

References

[1] R. Atkinson and C. Hewitt, Specification and proof techniques for serializers, AI Memo 438, Massachusetts Institute of Technology (1977).

[2] J.E. Burns, 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, *Proc 1978 International Conference on Parallel Processing*, Bellaire, MI (1978); see also, Data requirements for implementation of N-process mutual exclusion using a single shared variable, GIT-ICS-79/02, to appear in *J.ACM*.

[3] R. Campbell and A. Habermann, The specification of process synchronization using path expressions, Lecture Notes in Computer Science **16** (Springer, Berlin, 1974).

[4] A.K. Chandra, Computable nondeterministic functions, *Proc. 19th Annual Symposium on Foundations of Computer Science* (1978).

[5] A. Cremers and T.N. Hibbard, Mutual exclusion of N processes using $O(N)$-valued message variable, USC Department of Computer Science, manuscript (1975).

[6] E.W. Dijkstra, Co-operating sequential processes, in: *Programming Languages*, NATA Advanced Study Institute (Academic Press, New York, 1968).

[7] J. Feldman, Synchronizing distant cooperating processes, TR 26, Department of Computer Sciences, University of Rochester (1977).

[8] M. Fischer, N. Lynch, J. Burns and A. Borodin, Resource allocation with immunity to limited process failure, *Proc. 20th Annual Symposium on Foundations of Computer Science*, Puerto Rico (1979); see also GIT-ICS-79/10.

[9] C.A.R. Hoare, Communicating sequential processes, Technical Report, Department of Computer Science, the Queen's University, Belfast, Northern Ireland (1976).

[10] N. Lynch, Fast allocation of nearby resources in a distributed system, *Proc. 1980 Symposium on Theory of Computing*, Los Angeles (1980) 70–81.

[11] G. Milne and R. Milner, Concurrent processes and their syntax, *J.ACM* (April 1979) 302–321.

[12] C.A. Petri, Kommunikation mit Automaten, Schriften des Rheinisch Westfalischen Institut Instrumentelle Mathematik, Bonn. 1962.

[13] G. Peterson and M. Fischer, Economical solutions to the critical section problem in a distributed system, *Proc. 9th Annual ACM Symposium on Theory of Computing* (1977).

[14] A.C. Shaw, Software descriptions with flow expressions, *IEEE Trans. Software Engrg.* 4 (3) (1978).

[15] VAX11 software handbook, Digital Equipment Corporation (1978).