An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection

MANI CHANDY and JAYADEV MISRA
University of Texas at Austin

We propose a methodology for the development of concurrent programs and apply it to an important class of problems: quiescence detection. The methodology is based on a novel view of programs. A key feature of the methodology is the separation of concerns between the core problem to be solved and details of the forms of concurrency employed in the target architecture and programming language. We begin development of concurrent programs by ignoring issues dealing with concurrency and introduce such concerns in manageable doses. The class of problems solved includes termination and deadlock detection.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*deadlocks*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Deadlock detection, program development, stepwise refinement, termination detection

1. INTRODUCTION

We propose a methodology for the development of concurrent programs and apply it to an important problem. The methodology is based on a novel view of what a program is. We view a program as an initial condition and a *set* of atomic statements. The operation of a nonterminating program is as follows. Repeat forever: execute a statement selected nondeterministically, ensuring that in an infinite number of selections each statement is selected infinitely often. (We do not describe terminating programs in this paper.)

The state of a computation is given by the values of its variables. The only effect of executing a statement is to change values of variables. This effect is achieved by a multiple assignment statement. Hence we view a program as a declaration of variables and their initial values, and a set of multiple assignment statements.

This work was supported by the Air Force Office of Scientific Research under grant AFOSR 85-0252.
Authors' current address: Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712-1188.

The key features of the methodology are

(a) Concerns about the core problem to be solved are separated from the forms of concurrency available in the hardware on which the program is to be executed and the language in which the program is to be written.
(b) We adopt a global view of systems when specifying and reasoning about them during early stages of design.
(c) Our reasoning about systems employs predicates on system states. Our proofs are based on properties possessed by all states of the system that might occur during a computation.

Our reasons for the above desiderata are given later.

We present algorithms to solve a class of problems: detecting quiescent properties in distributed systems. Such properties include termination and deadlock. A consequence of our derivation is that we obtain the weakest conditions under which the algorithms can operate. Specific instances of the quiescence detection problem have been studied extensively [1–10, 12–18, 21–25]. These algorithms have the feature that each process is observed over some interval during the computation, and the intervals are related in some manner—for instance, intervals of neighboring processes overlap. Our solution differs in that we derive a class of solutions to a collection of problems for a variety of concurrent architectures, and our algorithms permit processes to be inspected at *arbitrary* times and in *arbitrary* order.

The organization of the paper is as follows. In the remainder of this section we explain our choice of desiderata for concurrent programming methodologies, present our model of programs and our methodology, and show how our methodology achieves the desiderata. In Section 2 we specify the quiescence detection problem and derive solutions in a series of refinements, stopping short of giving a complete program. In Section 2 the specification and derivation are in terms of a shared-variable model; the derivation is also appropriate, however, for a message-passing distributed system. The partial solution obtained in Section 2 can be extended to obtain a program suitable for concurrent architectures. We have chosen to extend it in Section 3 to obtain a message-passing program employing an asynchronous communication protocol.

1.1 Desiderata for a Concurrent Programming Methodology

Separating Concerns About the Core Problem from Details About Concurrency. Many papers on concurrent programs lump concerns about the core problem to be solved, the language in which the program is to be written, and the hardware on which the program is to be executed into a single agglutinous unit. Some argue that in cases where language and hardware are specified as part of a concurrent systems problem, concerns about the core problem, language, and hardware are inseparable. For instance, programs executing on a distributed network of computers must employ some form of message passing; in such cases concerns about message passing appear inseparable from concerns about the core problem. Similarly, since the presence or absence of primitives (such as process creation and termination) in the programming language influence the program, it appears that language issues are inseparable from others. Despite these

arguments we maintain that it is possible and important to separate these concerns—indeed it is even more important to do so for concurrent systems than sequential systems because concurrency is less well understood.

The ideas that form the foundation of good programming transcend different forms of concurrency employed in different implementations. Lumping all concerns together results in fundamental ideas getting lost in a welter of detail. A methodology should make the generality of important ideas manifest so as to avoid having solutions rederived from scratch for each form of concurrency.

Programs outlive the architecture for which they were initially designed. Experience suggests that we should anticipate requests to modify our programs to keep pace with modifications in architecture—witness attempts to "parallelize" sequential programs. Dijkstra [11] points out that a modification of a program is really a refinement of one of its ancestors; the further removed the ancestor, the more difficult the modification. It is difficult to make modifications necessitated by changes in the form of concurrency employed in a target architecture if the specific form of concurrency is a primary concern early in the design cycle. History tells us that we should not begin to solve a problem by asking ourselves whether we are going to use shared variables, message passing, or sequential programs, any more than we begin by asking ourselves if the word size is to be 60, 32, or 16 bits.

The Process-Eye View versus a Global Perspective. It may be more natural for a human being to "identify with" a single sequential process than with a system in which many actions happen "simultaneously" in different places. This identification results in arguments based on what each process "sees," "knows," and "learns" at specific points in the computation rather than on unvarying facts about the system. Reasoning about a system from the point of view of what is observable to each process and denying oneself a global perspective is to handicap oneself to no purpose. Therefore, we avoid arguments based exclusively on a collection of process eye-views. This view of reasoning has been strongly advocated by Lamport [19] and also appears in Manna and Pnueli [20].

Postulating subsystems to implement a desired system is an important part of program development. In the initial stages of program development we may not know what processes we are going to employ; we must perforce take a global perspective at this stage.

Reasoning About Unchanging System Properties versus Operational Reasoning. Operational arguments are about process behaviors unfolding over time. These arguments have the following flavor: "when process u receives a token it knows no other process is in its critical-section, and so it enters its critical-section and then, when it gets out, it sends the token and then, . . .". This form of reasoning specifies one or more sequences of actions for system execution and derives properties of the system from these action sequences. There is evidence that such arguments are error-prone. Operational reasoning is more difficult than reasoning about system properties because most of us find it more difficult to comprehend unfolding histories of actions than unchanging properties. This is especially important when dealing with nondeterministic systems because

nondeterminism leads to a combinatorial explosion in the number of possible histories. A danger with operational arguments is that some possible sequence of actions may be overlooked, while a proof must cover all possible sequences.

Specifications

Specifications are given in terms of predicates on system states. Let u, v be predicates on system states and let t be a statement in the program. We use

$$\{u\}\ t\ \{v\}$$

to denote that if u holds immediately before execution of t, then the execution of t terminates and v holds upon termination.

A predicate I is an *invariant* means I is true initially and $\{I\}\ t\ \{I\}$, for all statements t in the program. We define a binary relation $\rightarrow$ (read "leads to") between predicates, with respect to a given program, as follows:

$$u \rightarrow v \qquad \text{holds for a program} \quad \equiv$$

(1a) for all statements t in the program: $\{u \textit{ and not } v\}\ t\ \{u \textit{ or } v\}$ *and*
(1b) there exists a statement t in the program such that

$$\{u \textit{ and not } v\}\ t\ \{v\}, \quad \textit{or}$$

(2) for some predicate w,

$$(u \rightarrow w) \quad \textit{and} \quad (w \rightarrow v).$$

From (1a) it follows that if u holds at any point d in a computation, then (1) v holds at point d, *or* (2) u *and not* v holds continuously from d onwards until eventually v holds, *or* (3) u *and not* v holds at d and continuously thereafter; the third case is ruled out by our rule of program execution and (1b). The $\rightarrow$ relation is transitive from (2).

Hence, $u \rightarrow v$ holds for a given program means that if u holds, then within finite time (i.e., within a finite number of executions of program statements), v holds.

Heuristics for Stepwise Refinement

In this paper we focus attention on three heuristics.

(1) During early stages of design we give ourselves the freedom of using whatever variables are necessary to formulate a solution. Concerns about the distributed implementation of such variables are postponed to a later stage of design.

(2) We often *generalize* predicates on systems to predicates on subsystems. For instance, the predicate "system P is idle" may be generalized to "subsystem S of P is idle." This form of generalization often suggests how the next refinement step is to be carried out.

(3) We exploit locality of interactions in distributed systems by replacing in predicates "there exist processes p, q" by "there exist processes p, q which interact," when appropriate.

Consequences of Using the Model

We suggest that the methodology as outlined has the desirable features discussed in Section 1.2. Specifications are in terms of (constant) system properties: invariants and the relation $\rightarrow$. The focus on constant system properties facilitates the derivation of programs hand-in-hand with their proofs [13]. The uniform view of programs, independent of architecture, encourages the separation of concerns of the core problem from the form of concurrency employed in the target architecture and language. By focusing attention on the total system being considered at a refinement step, the model discourages reasoning based on the process eye-view. By employing nondeterminism to the limit and avoiding all forms of sequencing, the model inhibits operational reasoning.

Though the target architecture in this paper is a distributed system, the model and methodology (with additional heuristics) have been used to derive programs for diverse architectures including systolic arrays, PRAMs (parallel random access memory machines), and uniprocessors.

Notation. We use $s \parallel t$ where s and t are statements to denote their parallel execution. Where s, t are assignments statements, $s \parallel t$ is a multiple assignment statement. The only form of conditional we use is, **if** b **then** s **else** t (and where s, t are assignment statements this is equivalent to a multiple assignment statement with conditional expressions in the right-hand side). We also use send m along c, receive m along c to denote sending and receiving (respectively) of message m along channel c; again, these may be viewed as assignment statements to channel state variable sc of channel c where sc is a queue of messages: send m along c is equivalent to

$$sc := sc;\ m \quad \{; \text{ denotes concatenation}\}$$

receive m along c is equivalent to

$$(sc := tail(sc) \parallel m := head(sc)) \textbf{ if } sc \neq empty$$

2. SPECIFICATION OF DETECTION PROBLEMS

We first specify the general form of detection problems and later narrow the specification to quiescence detection.

We are given a program called the *underlying* program and a predicate W on the underlying program such that W is preserved by the underlying program (i.e., once W holds it continues to hold). It is required to "superpose" a program on the underlying program where the superposed program has a boolean variable *claim* satisfying:

$$\begin{aligned} &\textit{Invariant: } W \textit{ or not claim} \\ &\textit{Progress: } W \rightarrow \textit{claim} \end{aligned}$$

The invariant means that if *claim* holds, then so does W. The progress condition means that if W holds, then *claim* holds in finite time.

The superposed program can record but not affect the underlying computation. The superposed program can employ variables not named in the underlying program; for instance, *claim* is such a variable.

Example. Let W be given by $W \equiv$ the number of statement executions in the underlying program exceeds 10.

We superpose a program by transforming the underlying program as follows. Introduce superposed variables *count*, *claim* of types *integer*, *boolean* and initial values 0, *false*, respectively. Transform each statement s in the underlying program to

$$\langle s \parallel count := count + 1 \rangle$$

and add a statement t to the program where

$$t: claim := (count > 10).$$

We specify W in terms of *count* as follows: $W \equiv (count > 10)$.

The invariant (*count* > 10 *or not claim*) is easily proved. The progress condition (*count* > 10) $\rightarrow$ *claim* follows from: for all statements s in the transformed program:

$$\{count > 10 \text{ and not } claim\}\ s\ \{count > 10 \text{ or } claim\}$$

and, there exists a statement, namely t, in the transformed program:

$$\{count > 10 \text{ and not } claim\}\ t\ \{claim\}.$$

This little example illustrates what we mean by superposition.

We have specified detection problems in general. We now turn our attention to a subclass of detection problems: quiescence detection.

Quiescence detection deals with a specific property W and a specific class of underlying programs. The underlying program is a *concurrent* program consisting of a fixed set P of processes. In our notation, a concurrent program is an initial condition and a set of statements; a process is a subset of statements, and the union of all processes together with the initial conditions forms the program. In the following, p, q are processes, and all propositions about p, q are universally quantified unless stated otherwise. We are given a binary relation *affects* between processes, and associated with each p is a predicate *p.qui*. The underlying program satisfies the following *local quiescence* property: for all statements t in the underlying program:

$$\{p.qui \text{ and } [\forall\ q \text{ such that } q \text{ affects } p\colon q.qui]\}\ t\ \{p.qui\}.$$

This property means that p can transit from quiescence (*p.qui*) to nonquiescence (*not p.qui*) only if it has a nonquiescent affector.

The W to be detected is

$$W \equiv [\forall\ p\colon p.qui].$$

From the local quiescence property, it follows that W is *preserved* (i.e., once true it remains true).

At this point in program development, we do not interpret *p.qui* except to require that transitions from *p.qui* to *not p.qui* take place only if p has a nonquiescent affector. In particular, we do not specify whether *p.qui* is a local

variable of p. (Later we shall find that there are some architectures in which $p.qui$ is not local to p.)

Deriving a Program Skeleton

For convenience in reading we repeat the specifications:

$$W \equiv [\forall\, p\colon p.qui]$$

$$\begin{aligned} &\textit{Invariant}\colon\ W \textit{ or not } \textit{claim} \\ &\textit{Progress}\colon\ W \rightarrow \textit{claim} \end{aligned}$$

A superposed program (added to the underlying program) that meets the specification is

$$\begin{aligned} &\textit{initially}\colon\ \ \textit{claim} = \textit{false}; \\ &\textit{statement set}\colon\ \textit{claim} := W \end{aligned}$$

Though this is a satisfactory program for a sequential machine, we cannot implement it directly on a distributed system because it is not possible to evaluate the conjunction W (all processes are quiescent) in an atomic statement, so we now add to our initial specification the constraint that each atomic statement in the superposed program can only access variables named in a single component process. We now turn our attention to evaluating the conjunction, given this constraint.

Refinement Step

Processes are inspected one-at-a-time, and a process is added to a set *checked* of processes if the process satisfies some condition (and we postpone consideration of what this condition should be). We postulate that

$$\textit{claim} \equiv (\textit{checked} = P) \qquad \text{where } P \text{ is the set of all processes.}$$

In other words, "all processes are in *checked*" means *claim* holds. For brevity, processes in *checked* are called *checked* processes, and those not in *checked* are called *unchecked* processes. The idea of inspecting processes one-at-a-time and "checking them off" until all are checked off is an obvious way of satisfying the constraint that it is not possible to inspect all processes "simultaneously"; let us see where the idea leads. Eliminating *claim* from the specification, we get

$$\begin{aligned} &\textit{Invariant}\colon\ W \textit{ or } (\textit{checked} \neq P) \\ &\textit{Progress}\colon\ W \rightarrow (\textit{checked} = P) \end{aligned}$$

Predicate W is a system-wide property. Our next refinement is obtained by generalizing W to obtain a subsystem property w defined on process sets S where $S \subseteq P$, such that

$$w(P) \equiv W$$

We use the obvious definition: $w(S) \equiv [\forall\, p \text{ in } S\colon p.qui]$.

The reason that we replace system-wide properties by their generalizations is that we want the specifications to give us guidance about the variables of our program. In particular, we want the specifications to give us more guidance about *checked*.

Refinement Step

We rewrite the specification in terms of the generalization w as

$$\textit{Invariant}: \; w(\textit{checked}) \text{ or } \textit{checked} \neq P$$
$$\textit{Progress}: \; w(P) \rightarrow (\textit{checked} = P)$$

The equivalence of the invariant predicates in the previous and the current refinement steps follows from: if $\textit{checked} \neq P$, then both predicates evaluate to *true* (since the second terms of the disjunction hold), and if $\textit{checked} = P$, then both predicates evaluate to W (since the second terms of the disjunction do not hold, and $W \equiv w(P)$).

Refinement Step

In sketching out the algorithm in the first refinement step we said that an unchecked process is added to *checked* only if it satisfies some condition, and we postponed consideration of what that condition should be. Let us call the condition for p, *p.inc* or the "inclusion condition for p." The inclusion condition is a boolean predicate on system states. We postpone consideration of the precise definition and implementation of the inclusion condition.

We postulate the following statements (one for each p) in the superposed program:

if *p.inc* **then** $\textit{checked} := \{p\} \cup \textit{checked}$

Process p can be added to *checked* only by execution of the above statement.

Suppose $w(\textit{checked})$ does not hold prior to executing the above statement and suppose p is the only unchecked process. After executing the statement $w(\textit{checked})$ continues to remain *false* (see the definition of w). Therefore, for the invariant to hold we must have $\textit{checked} \neq P$, and hence p must remain unchecked. Therefore, a precondition to the above statement is that if $w(\textit{checked})$ does not hold, then there is at least one unchecked process q for which *not q.inc* holds. This argument suggests that we strengthen the invariant to

$$\textit{Invariant}: \; w(\textit{checked}) \quad \textit{or} \quad [\exists \text{ unchecked } q : \textit{not } q.\textit{inc}]$$

We elaborate our progress condition to

Progress:

(a) $w(P) \rightarrow [\forall\, p : p.\textit{inc}]$ *and*
(b) $[\forall\, p : p.\textit{inc}] \rightarrow \textit{checked} = P$

Refinement Step

We propose a stronger invariant by exploiting the "locality" of the relation *affects*.

$$\textit{Invariant}: \; w(\textit{checked}) \quad \textit{or}$$
$$[\exists \text{ unchecked } q, \text{ checked } p : (\textit{not } q.\textit{inc}) \quad \textit{and} \quad (q \textit{ affects } p)]$$

The reasons for the stronger invariant are as follows: Suppose $w(\textit{checked})$ does not hold. Consider the latest point d at which $w(\textit{checked})$ became *false*. From the definition of w, *not* $w(\textit{checked})$ means that there exists a nonquiescent checked process. A quiescent process becomes nonquiescent only if it has a nonquiescent

affector. Hence at d there exists an unchecked nonquiescent q, checked p, and q *affects* p. Intuition suggests that the process q that causes $w(checked)$ to become *false* should have its own inclusion condition set *false*. This argument leads us to propose the above stronger invariant.

Refinement Step

From the definition of w it follows that the invariant of the previous step is equivalent to

Invariant: [∀ checked p: $p.qui$] *or*

[∃ unchecked q, checked p: (*not* $q.inc$) *and* (q *affects* p)]

We propose to strengthen it to

Invariant I: [∀ checked p: $p.qui$ *and* $p.inc$] *or*

[∃ unchecked q, checked p: (*not* $q.inc$) *and* (q *affects* p)]

Our reasons for the stronger invariant are as follows. A process is added to *checked* only if its inclusion condition holds. If at some point in the computation the system is quiescent and a process' inclusion condition holds, then we expect it to continue to hold. A checked process changes its inclusion condition from *true* to *false* only when the system is nonquiescent, in which case (we design our algorithm so that) the second term in the disjunction holds.

What we are doing by strengthening the invariant is capturing intuitive, temporal, behavioral arguments by means of formal, invariant, system properties.

Pause to Review Stepwise Refinement

Before we proceed with stepwise refinement we pause to take stock of what we are doing. We have proceeded without concerning ourselves too much with the target architecture. For instance, we cannot implement checking-off statements directly on distributed architectures because *checked* is a global variable, and distributed architectures do not admit global variables. But that is not a serious concern at this level of program development; if a distributed system is a target architecture, then we shall concern ourselves *later* with implementing checking-off statements on that architecture.

Our understanding of the program is embodied in specifications at an appropriate level of detail and in a program skeleton. The skeleton takes the form of initial conditions and a set of (possibly nonimplementable) atomic statements.

The elaborated specifications can be used to develop apparently dissimilar algorithms. This suggests that the detailed specifications obtained in stepwise refinement are valuable quite apart from the algorithms.

Viewing a program as an initial condition and a set of atomic statements gives our methodology focus. We know that *all* statements must satisfy the *same* set of pre- and postconditions to ensure *safety*. Statements differ only in their contributions to *progress*. To derive a statement we postulate its contribution to progress (for instance, the purpose of checking-off statements is to increase the size of *checked*), and then to deduce the form of the statement from system-wide pre- and postconditions. In deriving a program we may find that our invariant needs strengthening. Thus the development of the program is an interplay

between system-wide safety properties and each individual statement whose purpose is to ensure some aspect of progress. The disadvantage of this approach is that by denying oneself the luxury of different contexts for different statements we require our global invariant to be strong enough to capture all contexts. (An apparent, but not real, disadvantage is that we must specify subsystems in terms of system-wide properties; compositional proofs are indeed possible, but there is insufficient space here to describe them.)

We can continue the refinement for a variety of target architectures; however, in the interest of brevity, we limit ourselves to only one: static, fault-free distributed systems with asynchronous sending/receiving of messages and point-to-point, directed, first-in-first-out channels with unbounded buffers. By "static" and "fault-free" we mean that processes and channels in the underlying system are given: they are not created, nor do they disappear or fail. A channel is directed from precisely one process to precisely one process. There are no restrictions on when processes send messages. The only restriction on when a process may receive a message along a channel is the obvious one: the channel must contain a message. (Of course, we are obliged to prove that the number of messages in each channel is indeed bounded—but it is helpful to separate concerns: assume a simple protocol with unbounded buffers and postpone proofs about bounds). A channel is a shared variable between two processes in the sense that the process sending and the process receiving along a channel may change the state of the channel; however, neither process can determine the state of a channel directly. This aspect of channels makes detection problems in asynchronous distributed systems particularly interesting.

We leave to the reader the problems of refining the program for shared-variable concurrent systems, distributed systems with synchronous communication, and distributed systems with asynchronous multiway channels (connecting many processes to many processes).

3. REFINEMENT FOR DISTRIBUTED SYSTEM ARCHITECTURE

We now continue refinement for the distributed system architecture described in the previous section.

Each process and each message has a boolean attribute: *stable.* A stable process can become *unstable* (i.e., *not stable*) only by receiving an unstable message. All messages sent by stable processes are stable. The problem is to detect the (preserved) property W, given by

$$W \equiv \text{all processes and all messages in all channels are stable.}$$

We define process quiescence so as to ensure $W \equiv [\forall\, p : p.qui]$. We propose

$$p.qui \equiv p.stable \quad and \quad [\forall\, p\text{'s input channels } c : c.stable]$$

where

$$c.stable \equiv c \text{ contains no unstable message}$$

(*Note:* We could have used other definitions for $p.qui$, for instance,

$$p.qui \equiv p.stable \quad and \quad [\forall\, p\text{'s output channels } c : c.stable]$$

Different definitions lead to slightly different programs.)

In a distributed system q *affects* p means there is a channel from q to p. Neither p nor q can access the state of channel (q, p) directly; q and p must cooperate to determine the state of channel (q, p). The algorithms differ in how the cooperation is achieved.

Marker Algorithm

Cooperation between q and p to determine the state of channel (q, p) is achieved by q sending p a special message, which has no effect on the underlying computation; this message is called a *marker*. For channel c from q to p, process q maintains a local variable $c.sm$ (for *s*end *m*arker) and p maintains local variable $c.rm$ (for *r*eceive *m*arker), with the following meaning. Variable $c.sm$ takes on values *pre*, *pos*, and *neg*—where its value is *pre* means the marker has not been sent along c; its value is *pos* means the marker has been sent along c, and all postmarker messages sent along c are stable; and its value is *neg* means the marker has been sent along c, and an unstable postmarker message has been sent along c. Variable $c.rm$ is boolean—where $c.rm$ holds means the marker has been received along c. These arguments lead us to postulate invariant:

$$(c.rm \text{ and } c.sm \neq neg) \Rightarrow c.sm = pos \text{ and } c.stable$$

This gives us a clue about the inclusion condition: we propose that

$$p.inc \equiv p.stable \quad and$$
$$[\forall\ p\text{'s incoming channels } c\text{: } c.rm] \quad and$$
$$[\forall\ p\text{'s outoing channels } c\text{: } c.sm \neq neg]$$

We now postulate invariants for the marker algorithm. Using Invariant I, definitions of $p.qui$ and $p.inc$, we postulate

Invariant K: $[\forall\ p \text{ in } checked : p.stable]$ *and*

$$[\forall \text{ channels } c \text{ to checked processes: } c.rm] \quad and$$
$$[\forall \text{ channels } c \text{ to, or from, checked processes: } c.sm \neq neg]$$

or

$$[\exists \text{ channels } c \text{ from an unchecked to a checked process: } c.sm = neg]$$

Also, from our description about the movement of markers, we postulate an invariant relating, for each channel c, $c.rm$, $c.sm$, and marker in c. Let $c.num$ be the number of markers in c.

Invariant L:

$$(c.num \leq 1) \text{ and not } (c.rm \text{ and } c.num = 1)$$

and

$$(c.sm = pre) \Rightarrow (c.num = 0 \text{ and not } c.rm)$$

and

$$c.sm = pos \Rightarrow [\forall \text{ messages } m \text{ in } c\text{: } m \text{ is stable or there is a marker following } m \text{ in } c]$$

Invariants K and L imply invariant I.

We now postulate the progress conditions, taking into account marker transmission and the values of *c.rm*, *c.sm*. The first two progress conditions, given below, are easy to see. The next two describe progress with reference to *c.sm* and *c.rm*.

Progress Conditions for the Marker Algorithm

For all channels c:

(1) $c.sm = pre \rightarrow c.sm = pos$ *and* $c.num = 1$,
(2) $c.num = 1 \rightarrow c.rm$,
(3) $W \rightarrow W$ *and* $[\forall\ c : c.sm = pos$ *and* $c.rm]$,
(4) W *and* $[\forall\ c : c.sm = pos$ *and* $c.rm] \rightarrow checked = P$.

Progress conditions require that if W holds, then no *c.sm* remains *neg* forever. Therefore, we add an additional progress condition to guarantee that *c.sm* which is *neg* will be set to *pre* within finite time.
For all channels c:

(5) $c.sm = neg \rightarrow c.sm = pre$

The marker algorithm follows from the invariant and progress condition. Each statement implements a progress condition. Each statement must also preserve invariants. For instance, changing *c.sm* from *neg* to *pre* (see progress condition 5) when c is from an unchecked to a checked process may violate invariant K; we preserve this invariant by setting *checked* to *empty*.

The Marker Algorithm

Initially: $checked = empty$,

$[\forall\ c : c.rm = false$ *and* $c.sm = pre]$

Set of Statements.

Marker sending along c:

if $c.sm = pre$ **then begin** send *marker* along $c \parallel c.sm := pos$ **end**

Upon receiving marker along c:

if marker is received along c **then** $c.rm := true$

Upon sending unstable message along c:

if $c.sm = pos$ *and* unstable message sent along c **then** $c.sm := neg$

Reinitializing c:

if $c.sm = neg$ *and* $c.rm$ *and* c is from an unchecked to an unchecked process
then begin $c.sm := pre \parallel c.rm := false$ **end**

Reinitializing c and *checked*:

if $c.sm = neg$ *and* $c.rm$ *and* c is from an unchecked to a checked process
then begin $checked := empty \parallel c.sm := pre \parallel c.rm := false$ **end**

Adding q to *checked*:

if *q.stable and*

[∀ input channels c of q: *c.rm*] *and*
[∀ output channels c of q: $c.sm \neq neg$]

then *checked* := $\{q\} \cup$ *checked*

Refinement Step

We now have the program in hand, except for the distributed implementation of *checked*. Also, variables (*c.sm*, *c.rm*) of different processes appear in a single statement. To implement a global variable on a concurrent system we need only ensure that every process needing to access the variable does so in finite time, at most one process accesses the variable at any time, and atomicity constraints are preserved. In this instance we are dealing with only one global variable—*checked*—and hence the problem reduces to that of mutual exclusion. An obvious way of implementing mutual exclusion is to have a single token in the system at all times, and to allow a process to execute its critical section (i.e., access *checked*) only upon holding the token. The information *checked* is carried by the token.

We assume that there are n processes indexed i where $0 \leq i < n$. We employ a boolean variable *holdstoken* with each process, where $i.holdstoken \equiv i$ holds the token.

We now give the invariant and progress conditions for the token-passing algorithm given below. We weaken invariant K to

Invariant K:

[∀ p in *checked*: *p.stable*] *and*
[∀ c between checked processes: *c.stable*] *and*
[∀ c from checked processes: $c.sm = pos$]

or

[∃ c from unchecked to checked process: $c.sm = neg$]

Invariant L is as before.

Define "token is between j, k" to mean "token is in channel $(i, i + 1 \bmod n)$" or $(i + 1).holdstoken$ for some i in $\{j, j + 1 \bmod n, \ldots, k - 1 \bmod n\}$. Invariant M describes the properties of markers and *c.rm* with respect to the position of the token. For all channels (j, k):

Invariant M: Token is between $j, k \equiv (j, k).rm$ *or* $(j, k).num = 1$.

Progress conditions describe how processes are added to *checked* as the token moves.

Progress Condition. For all sets of channels C and all processes i:

W *and* $i.holdstoken$ *and* [∀ c in C: $c.sm = pos$] →
W *and* $(i + 1).holdstoken$ *and* [∀ c in C': $c.sm = pos$]

where $C' = C \cup \{c \mid c$ is an output channel of process $i\}$.

Progress Condition. For all sets of processes Q and all processes i:

$$W \text{ and } i.holdstoken \text{ and } [\forall\, c : c.sm = pos] \text{ and } [\forall\, q \text{ in } Q : q \in checked] \rightarrow$$
$$W \text{ and } (i+1).holdstoken \text{ and } [\forall\, c : c.sm = pos] \text{ and } [\forall\, q \text{ in } Q' : q \in checked]$$

where $Q' = Q \cup \{i\}$.

From the first progress condition:

$$W \rightarrow W \text{ and } [\forall\, c : c.sm = pos]$$

From the second progress condition:

$$W \text{ and } [\forall\, c : c.sm = pos] \rightarrow W \text{ and } checked = P$$

Hence $W \rightarrow checked = P$.

We now give the algorithm for a process. Note that this is the first time that we have written our algorithm in terms of statements in component processes. Up to this point we have presented our algorithm as a set of statements and ignored questions of how the set is partitioned among component processes.

Algorithm for Process i, $0 \leq i < n$:

if *i.holdstoken and i.stable and* [∀ input channels c of i: *c.rm*]
then begin
send token to $(i + 1)$ *mod n* with *checked* as follows:
if [∀ channels (i, j) to checked j: $(i, j).sm \neq neg$]
then *checked* := *checked* ∪ $\{i\}$ **else** *checked* := *empty*
‖ *i.holdstoken* := *false*
‖ [for all input channels c of i: *c.rm* := *false*]
‖ [for all output channels c of i: send *marker* along c ‖ *c.sm* := *pos*]
end

Upon process i receiving the token:

if i receives the token **then** *i.holdstoken* := *true*

Upon process i receiving marker along c:

if marker is received along c **then** *c.rm* := *true*

Upon process i sending unstable message along c:

if *c.sm* = *pos and* unstable message sent along c **then** *c.sm* = *neg*

Deriving Initial Conditions for the Algorithm

Invariant K is ensured by having *checked* initially *empty.* Invariant L is ensured initially by having for all c: $(c.sm = neg)$ and $c.num = 0$. Invariant M is ensured by choosing any initial position of the token and then having $(j, k).rm \equiv$ the token is between (j, k). We choose to place the token at process 0 initially. Initially:

—$i.holdstoken \equiv (i = 0)$ {token is at process 0};
—for a channel c from a process i to a process j, for all i, j: $c.rm \equiv (i > j)$;
—no channel contains a marker (i.e., for all c: $c.num = 0$);
—for all channels c: $c.sm = neg$;
—*checked* = *empty.*

Note. In the algorithm we require that each process send the token to the next process in a cycle. We do not require that there be a channel from each process to the next one in the cycle. The token can be sent from one process to another via intermediate processes.

Refinement Steps for Optimization

We now refine the program to improve its efficiency. This final refinement also results in a simpler program.

Reducing the Volume of Information Carried by the Token. Recall that the token carries the value of *checked*. The volume of information carried by the token may be large because *checked* may contain many processes; we now seek ways of reducing this volume. By sending the token in a cycle, numbering processes $0..(n-1)$ so that the token is passed from process i to $(i+1)$ *mod* n, and postulating that *checked* consists of a sequence of processes ending in the process holding the token, we can determine *checked* by keeping track of the identity of the first process in the sequence. Therefore, we postulate that the token has a field *init* containing the identity of a process and having the following meaning. In the interval between the token leaving j and its processing by $(j+1)$ *mod* n:

$$init = j \equiv checked = empty \quad and$$
$$init \neq j \equiv checked = \{(init + 1) \bmod n, \ldots, j\}$$

Keeping Track of Output Channels. The only purpose of *c.sm* is to determine if *c.sm* = *neg* for a channel *c to a checked process.* The sequential numbering of processes allows us to implement *c.sm* for all output channels *c* of a process by means of a single local variable, *farthest* (for farthest negative), of the process, where *i.farthest* is the index of the process farthest from *i*, for which *c.sm* = *neg*. The sequence of processes ranked farther from i is

$$(i+1) \bmod n, (i+2) \bmod n, \ldots, (i-1) \bmod n$$

When the token is at i, the statement "there is a negative channel from i to a *checked* process" means *i.farthest* is in $(init + 1) \bmod n, \ldots, (i-1) \bmod n$; in particular, *i.farthest* = i means all of process i's output channels are positive.

Keeping Track of Input Channels. The variables *c.rm* are used only to determine if a process has received markers on all its input channels. This observation allows us to replace variables *c.rm* by a count *nmr* for each process where *nmr* is the number of markers received by the process (since the token last left the process). In particular, *nmr* = number of input channels, means a marker has been received along each input channel since the token last left the process.

We leave the derivation of the optimized program using *init*, *farthest*, *nmr* in place of *checked*, *sm*, *rm* to the reader.

A Note on Optimization

In the algorithms we have given, if there is a channel from an unchecked q to checked p such that *not q.inc*, then when we change the value of *q.inc* we reinitialize the algorithm by setting *checked* to empty, thus maintaining the invariant: $w(checked)$ or there exists a channel from an unchecked to a checked

process where the sender along the channel does not satisfy the inclusion condition. It is not always necessary to set *checked* to empty when the value of *q.inc* is changed. It is sufficient to remove from *checked* the set of processes that have received unstable messages from q and that may in turn have sent unstable messages to other checked processes, and so on. The algorithm for removing all processes from *checked* that may have been (indirectly) affected by q's sending of an unstable message is straightforward. We invite the reader to develop such an algorithm using the invariant as the foundation of the development.

4. EXTENSIONS

Acknowledgment Algorithm

Our primary purpose in presenting another algorithm is to show how *stepwise refinement leads to a class of solutions.* A secondary purpose is that the algorithm we present is efficient, particularly in distributed systems in which all messages are acknowledged. Messages may be acknowledged (ack'd) for several reasons; for instance, the communication protocol may be based on acks. If the system on which we are to impose a detection algorithm acks messages, we may as well employ the acks to our advantage. We shall not develop the algorithm in detail but merely present an outline. The algorithm is based on invariant I.

Each ack has two attributes *checked* and *stable* where checked processes send checked acks, unchecked processes send unchecked acks, and acks for stable messages are stable acks and acks for unstable messages are unstable acks. We define *q.inc* as

q.inc ≡ *q.stable and*
acks have been received by q for all unstable messages sent by q *and*
all acks received by q for unstable messages sent by q are unchecked acks.

The algorithm is reinitialized (i.e., *checked* is set to *empty*) if an unchecked process receives a checked, unstable ack. We leave the derivation of elaborated specifications and the program to the reader.

Comparison with Other Algorithms

Our algorithms differ from most others in one important aspect. In most other algorithms, if a nonquiescent process is detected, the algorithm is reinitialized because the presence of the nonquiescent process means that W does not hold. If the algorithm determines that W does not hold, then why not restart the algorithm? Efficiency suggests that the algorithm be continued rather than restarted where possible. Our algorithm is reinitialized *only* if an unchecked to checked channel is unstable. Algorithms based on global snapshots [7], and on overlapping intervals of observation at processes, are reinitialized if the snapshot or the observation shows that W does not hold.

5. CONCLUSION

Our model of programs (a set of statements) helped us to focus on the appropriate level of detail of architecture at each step of refinement. The model allows us to develop pieces of the program given only the invariant, *independent of other pieces.* This encouraged concentration of attention on one concern at a time. Our

model of a program (a set of multiple assignments) may appear unduly austere; however, our experience suggests that the model is adequate. Nondeterminism captures the essence of various forms of concurrent programming. An ongoing project, UNITY, has the goal of determining whether programs in diverse areas may be developed systematically by viewing them as sets of multiple assignment statements (and initial condition specifications).

The program was not derived in one pass as might be suggested here. We backtracked several times, and we derived two less efficient algorithms. Ideally, a description of program derivation should include a description of wrong turns, consequent backtracking, and what was learned from the mistakes. Backtracking is not described here in the interests of brevity.

ACKNOWLEDGMENTS

Our ideas about anthropomorphism and operational reasoning were sharpened by discussions and arguments with E. W. Dijkstra. We also owe him a debt of gratitude for his detailed criticism of earlier drafts. Comments from Hank Korth and Shmuel Katz are appreciated. Discussions at IFIP W.G. 2.3 helped in clarifying our ideas. We are especially grateful to the Austin Tuesday Afternoon Club for a careful reading of the manuscript, and to the referees for insightful comments on the first draft.

REFERENCES

1. Beeri, C., and Obermarck, R. A resource class independent deadlock detection algorithm. Res. Rep. RJ3077, IBM Research Laboratory, San Jose, Calif., May 1981.
2. Bracha, G., and Toueg, S. A distributed algorithm for generalized deadlock detection. Tech. Rep. TR 83-558, Cornell Univ., Ithaca, June 1983.
3. Chandy, K. M., and Misra, J. A distributed algorithm for detecting resource deadlocks in distributed systems. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Can., Aug. 1982), ACM, New York.
4. Chandy, K. M., Misra, J., and Haas, L. Distributed deadlock detection. *ACM Trans. Comput. Syst. 1*, 2 (May 1983), 144–156.
5. Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 63–75.
6. Chang, E. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Softw. Eng. SE-8*, 4 (July 1982), 391–401.
7. Cohen, S., and Lehmann, D. Dynamic systems and their distributed termination. *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Can., Aug., 1982), ACM, New York, 29–33.
8. Dijkstra, E. W., and Scholten, C. S. Termination detection for diffusing computations. *Inf. Process. Lett. 11*, 1 (Aug. 1980).
9. Dijkstra, E. W. Distributed termination detection revisited. EWD 828, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.
10. Dijkstra, E. W., Feijen, W. H. J., and Van Gasteren, A. J. M. Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett. 16* (1983), 217–219.
11. Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
12. Francez, N. Distributed termination. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan. 1980), 42–55.
13. Francez, N., Rodeh, M., and Sintzoff, M. Distributed termination with interval assertions. In the *Proceedings of Formalization of Programming Concepts* (Peninsula, Spain, Apr. 1981), *Lecture Notes in Computer Science 107*, Springer Verlag, New York.
14. Francez, N., and Rodeh, M. Achieving distributed termination without freezing. *IEEE Trans. Softw. Eng. SE-8*, 3 (May 1982), 287–292.

15. GLIGOR, V., AND SHATTUCK, S. On deadlock detection in distributed databases. *IEEE Trans. Softw. Eng. SE-6*, 5 (Sept. 1980).
16. GOUDA, M. Distributed state exploration for protocol validation. TR-185. Dept. of Computer Sciences, Univ. of Texas, Austin, Oct. 1981.
17. HAAS, L., AND MOHAN, C. A distributed deadlock detection algorithm for a resource-based system. Res. Rep. RJ3765, IBM Research Laboratory, San Jose, Calif., Jan. 1983.
18. HERMAN, T., AND CHANDY, K. M. A distributed procedure to detect AND/OR deadlock. Dept. of Computer Sciences, Univ. of Texas, Austin, Feb. 1983.
19. LAMPORT, L. An assertional correctness proof of a distributed algorithm. In *Science of Computer Programming, 2*, North-Holland, Amsterdam, 1982, 175–206.
20. MANNA, Z., AND PNUELI, A. How to cook a temporal proof system for your pet language. In *Symposium on Principles of Programming Languages* (Austin, Tex., 1983).
21. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed databases. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979).
22. MISRA, J., AND CHANDY, K. M. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst. 4*, 1 (Jan. 1982), 37–43.
23. MISRA, J. Detecting termination of distributed computations using markers. In *Proceedings of the ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing* (Montreal, Can., Aug. 17–19, 1983), ACM, New York.
24. OBERMARCK, R. Deadlock detection for all resource classes. Res. Rep. RJ2955, IBM Research Laboratory, San Jose, Calif., Oct. 1980.
25. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2, (June 1982), 187–208.

Received January 1985; revised November 1985; accepted November 1985