Proofs of Distributed Algorithms: An Exercise

K. Mani Chandy*
The California Institute of Technology
Pasadena, California 91125
(818) 356-6559
mani@vlsi.caltech.edu

Jayadev Misra†
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

1 Introduction

It is generally assumed that formal proofs of programs are considerably longer and more tedious than their informal counterparts. Informal proofs employ a form of common sense reasoning whereby "obvious" facts are often omitted and the proof steps rely upon the intuition of the reader. Typically informal proofs are operational; arguments consist of the properties of program executions as they unfold over time.

Our goal in this paper is to suggest, by means of an example, that formal proofs can be made as concise as the informal ones. This argument rests upon two observations: (1) informal proofs tend to be long and difficult (in addition to being error-prone) when there are many interleaved execution sequences to consider, as is the case in multiprocess programs, and (2) formal proofs can be made concise by employing a logic that is appropriate for the problem domain and whose operators possess a number of useful properties that can be exploited in proofs.

In recent years, we have developed a programming and proof theory, called UNITY, Chandy and Misra [1988]. Our experience in using the UNITY proof theory on a wide range of problems has led us to believe that formal proofs need not be outrageously long or tedious. In this paper, we apply the UNITY proof theory to a problem in distributed computing—termination detection. We specify the problem and develop a correctness proof of a solution without relying upon the operational aspects of program execution. Use of our logic allows us to eliminate arguments about a program's execution sequences. We believe strongly that formal proofs cannot be made concise as long as they mimic the arguments in the informal proofs.

Most of this paper is about UNITY theory and specification of message communicating processes; only sections 4 and 6 contain the proof of the termination detection algorithm. The paper is self-contained: no familiarity with UNITY or termination detection is assumed.

*The work by K. Mani Chandy reported here is partially supported by Office of Naval Research Contract N00014–86–K–0763.

†Jayadev Misra's contribution is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658–065 and by support from the Office of Naval Research Contract N00014–90–J–1640.

1.1 Termination Detection

Here, we give a brief informal overview of the termination detection problem and its proposed solution.

We are given a finite set of processes that communicate by messages. A process sends a message to another process by depositing it in a channel that is directed from the former to the latter; the message is received after an arbitrary, but finite, delay and is then removed from the channel. A process is either *idle* or *nonidle*. An idle process remains idle until it receives a message, and an idle process does not send messages. A nonidle process may become idle autonomously. The system of processes is said to be *terminated* when all processes are idle and all channels are empty, because all processes will remain idle and all channels will remain empty from then on. The problem of termination detection is for some process to ascertain that the system has terminated; in order to do so, processes carry out some additional termination detection computation along with their basic computation.

The following algorithm may be employed to detect termination. From time to time, each process records its state (idle or nonidle), the number of messages it has received along each of its incoming channels and the number of messages it has sent along each of its outgoing channels. Different processes may record these values at different times. Clearly, the recorded values become obsolete if a process sends or receives a message or changes its state. Remarkably, though, the system is terminated if the recorded state of each process is idle and for each channel the recorded number of messages sent is equal to the recorded number of messages received. Therefore the synchronous computation required to detect termination can be replaced by asynchronous recordings and computations.

As a trivial optimization note that processes need not do any recording as long as they are nonidle—because the recorded state of a process will then be nonidle and hence the termination condition would not be met—and therefore only the number of messages sent and received along incident channels need be recorded by a process when it is idle. The nondeterminism inherent in this solution—there is no restriction on *when* a process records—makes it possible to develop a number of different algorithms from the one sketched above by specifying the order of recording. We outline two algorithms below which are obtained by restricting the order of recordings.

The detection algorithm may employ a single token. Computation involving the token is separate from the given underlying computation; thus, idle processes in the underlying computation may send and receive the token. The token visits the processes in some fixed order and it carries all the recorded information (recorded states of all processes and the recorded number of message sends and receives for each channel). A process does the recording sometime after it receives the token; it then updates the recorded information in the token appropriately and sends the token to the next process. Termination is detected by the token (or the process holding the token or by a prespecified "detector" process) if the recorded information shows each process to be idle and each channel to be empty. Note that the introduction of the token is merely an artifact for restricting the order in which the recordings are made. Correctness of this solution follows from the correctness (yet to be shown) of the original nondeterministic solution.

Another strategy is to introduce a special process, *detector*, that sends messages to all other processes asking them to record and send it the recorded information. The detector can declare termination based on the information it receives from the processes. The order in which the detector queries the processes is irrelevant for correctness.

Now we sketch an informal proof of correctness of the proposed solution. This proof relies on the reader's intuition about how a message communicating system operates. The proof has the flavor of a typical informal proof of a distributed algorithm; it is not necessary to read this proof for understanding the main ideas—formal proofs in UNITY—of this paper.

Let the recorded state of each process be idle and the recorded number of messages sent and received be equal for each channel. We show that the system is then terminated. Assume the

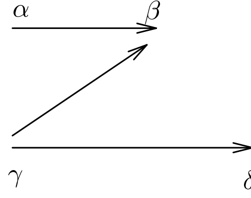

Figure 1: Relationships among certain time instants in the recording algorithm

contrary. Then there is a process, i, that became nonidle after it last recorded its state. Process i can become nonidle only by receiving a message, say message m from process j. Consider the following events and the times at which the events occur.

time		event
α	:	process i last records its state (idle) and the message counts
β	:	process i becomes nonidle (upon receiving message m)
γ	:	process j sends message m
δ	:	process j records its state (idle) and the message counts

Without loss in generality let i be the first process to become nonidle after its last recording. We have $\alpha < \beta$; also, $\gamma < \beta$ because message m is sent at γ and received at β. Furthermore, $\gamma < \delta$. This is because (1) $\gamma \neq \delta$, since process j is idle at δ and nonidle at γ, and (2) if $\delta < \gamma$ then process j becomes nonidle at γ (or earlier) after its last recording (at δ); hence it becomes nonidle before β ($\gamma < \beta$) thus contradicting our choice of process i.

The diagram in Figure 1 depicts the relationships among α, β, γ, δ schematically: An arrow is drawn from a time instant to another if the former precedes the latter.

Let,

f = the number of messages sent by j to i as recorded by j (at time δ)
g = the number of messages received by i from j as recorded by i (at time α)

Since message m was received at β, and $\beta > \alpha$, this message is not included in the count g. Since f is recorded at δ and $\gamma < \delta$ message m is included in the count f. Therefore, $f \neq g$. This contradicts our assumption that the recorded number of messages sent and received for every channel are equal.

A number of implicit assumptions have crept into the informal proof. Presumably no process can perform a recording at the same time that it sends or receives a message. Similarly, it is assumed that no process can receive a message unless it has been sent earlier, i.e., sending and receiving of one message cannot be simultaneous. Furthermore, the last part of the proof assumes that messages are received in the order sent. The arguments dealing with time—events happening before or after some point in time—reflect the way we understand the program execution to unfold over time. One reason for constructing a formal proof is to make all assumptions explicit. Another reason is to replace arguments about unfolding computations—a temporal entity—by arguments about the program text—a nontemporal entity.

The formal proof that we propose in this paper is completely different in character. We do not argue about the program execution. Our arguments are based on the specifications and texts of programs and not what effect these programs have when executed on computers. We prove that the following is invariant for any set of processes X: if

1. All processes in X have been recorded idle,
2. For all internal channels of X (i.e., those directed between processes in X) equal number of sends and receives have been recorded, and

3. For all incoming channels to X (from processes outside X) the recorded number of receives equal the actual number of receives

then

1. All processes in X are idle,
2. For all internal channels of X the number of receives and sends are equal, and
3. For all outgoing channels from X (to processes outside X) the recorded number of sends equal the actual number of sends.

Letting X be the set of all processes in the above invariant—the condition (3) in the antecedent of the invariant is then vacuously *true*—proves the desired theorem.

The problem of termination detection, and a solution for it, first appeared in Francez [1986]. The algorithm sketched above was invented by Chandy [1983] and independently by Helary et al [1987]. A description of Chandy's algorithm appears in Misra [1986]. Proofs appear in Chandy [1987], Dev Kumar [1987], and Hesselink [1987]. Nondeterminism, first postulated by Dijkstra [1976], is at the core of UNITY programming. UNITY logic is deeply influenced by temporal logic (Pnueli [1977] and Owicki and Lamport [1982]).

2 A Brief Introduction to UNITY

In this section, we describe those aspects of UNITY that are essential for understanding this paper; for a full description see Chandy and Misra [1988], Chapters 2, 3, and 7 in particular. First we give a brief overview of the operational behavior of UNITY programs. We do not describe the syntax of UNITY programs in this paper; a program is given in section 3.4 whose syntax is explained with reference to that program alone.

2.1 Operational Descriptions of UNITY Programs

A UNITY program has a set of variables; initial values of (some of) these variables may be specified in the program. The body of the program consists of a finite set of multiple assignment statements. A program execution starts in a state where the initial values of the variables are as specified. In each step of the execution a statement from the program body is selected for execution. Statements are selected arbitrarily with the restriction that in an infinite execution (i.e., an execution with an infinite number of steps) each statement is executed infinitely often.

Notions of process, channel, and message communication are not part of the UNITY theory. A UNITY program's variables and statements may be partitioned in various ways for execution on multiple processors. Such a partitioning does not affect the correctness of the program. Therefore we deal with correctness issues by ignoring the question of implementation of the program as a set of communicating processes at the outset.

2.2 The Logical Operator *unless*

Notational Convention Throughout this paper p, q, r denote arbitrary predicates that may name program variables, bound variables, and free variables (free variables are those that are neither program variables nor bound variables), and F, G, arbitrary UNITY programs. All formulae are (implicitly) quantified universally over all free variables appearing in them.

Safety properties are expressed by a logical operator, *unless*. For a program F,

$$\frac{\langle \forall\ t\ :\ t \text{ is a statement in } F\ ::\ \{p \wedge \neg q\}\ \ t\ \ \{p \vee q\}\rangle}{p\ \ \mathit{unless}\ \ q\quad \text{in } F}$$

This inference rule should be read as follows: p *unless* q in F can be inferred given that for every statement t in F if $p \wedge \neg q$ holds prior to the execution of t then $p \vee q$ holds upon completion of the execution of t. (We assume that execution of every statement always terminates.)

Note The form of quantification shown in the above definition appears several times in the paper. The notation $\{p\}\ s\ \{q\}$ is from Hoare [1969].

From p *unless* q in F we may deduce that if p holds at any point during an execution of F it continues to hold at least as long as q does not hold. To see this suppose that p holds at some point during an execution. If q holds then the above claim is valid. If q does not hold, i.e., $p \wedge \neg q$ holds, then execution of any statement establishes $p \vee q$ in the next step. If $\neg q$ holds in the next step, then $p \wedge \neg q$ holds and the same argument can be repeated; if q holds then the claim is seen to be valid.

We define two more concepts using *unless*. In the following, "initially p in F" means that p follows from the initial conditions of program F.

$$
\begin{array}{lcl}
p \text{ stable in } F & \equiv & p \ \ unless \ \ false \ \ \text{ in } F \\
p \text{ invariant in } F & \equiv & (\text{initially } p \text{ in } F) \ \wedge \ (p \text{ stable in } F)
\end{array}
$$

From the definition, p stable in F means that for all statements t in F,

$$\{p\} \ \ t \ \ \{p\}$$

Thus, once p is *true* it remains *true*. An invariant is initially *true* and remains *true* throughout any execution of the program.

Notational Convention The program name is omitted from a property if it is clear from the context.

2.3 Derived Rules About *unless*

For formal proofs in this paper, we do not rely on the intuitive meaning of *unless*; instead we use the following derived rules, proofs of which may be found in Section 3.6.1 of Chandy and Misra [1988]; also see Misra [1990].

1. Consequence weakening

$$\frac{p \ unless \ q \ , \ q \ \ \Rightarrow \ \ r}{p \ unless \ r}$$

2. Conjunction and disjunction

$$\frac{p \ unless \ q \ , \ p' \ unless \ q'}{\begin{array}{ll} p \ \wedge \ p' \ unless \ (p \ \wedge \ q') \ \vee \ (p' \ \wedge \ q) \ \vee \ (q \ \wedge \ q') & \{\text{conjunction}\} \ , \\ p \ \vee \ p' \ unless \ (\neg p \ \wedge \ q') \ \vee \ (\neg p' \ \wedge \ q) \ \vee \ (q \ \wedge \ q') & \{\text{disjunction}\} \end{array}}$$

Simpler forms of conjunction and disjunction are often useful; these are obtained from the above rule by weakening the consequence to $q \ \vee \ q'$ in both cases:

3. Simple conjunction and simple disjunction

$$\frac{p \ unless \ q \ , \ p' \ unless \ q'}{\begin{array}{ll} p \ \wedge \ p' \ unless \ q \ \vee \ q' & \{\text{simple conjunction}\} \\ p \ \vee \ p' \ unless \ q \ \vee \ q' & \{\text{simple disjunction}\} \end{array}}$$

The following rule generalizes the conjunction and disjunction rules to an arbitrary—perhaps infinite—number of *unless*es; for a proof, see Misra [1988]. In the following, m is quantified over some arbitrary set.

4. General conjunction

$$\frac{\langle \forall\ m\ ::\ p.m\ \textit{unless}\ q.m \rangle}{\langle \forall\ m\ ::\ p.m \rangle\ \textit{unless}\ \langle \forall\ m\ ::\ p.m \vee q.m \rangle\ \ \wedge\ \langle \exists\ m\ ::\ q.m \rangle}$$

5. General disjunction

$$\frac{\langle \forall\ m\ ::\ p.m\ \textit{unless}\ q.m \rangle}{\langle \exists\ m\ ::\ p.m \rangle\ \textit{unless}\ \langle \forall\ m\ ::\ \neg p.m \vee q.m \rangle\ \wedge\ \langle \exists\ m\ ::\ q.m \rangle}$$

The following rule is a corollary of general disjunction:

6. Free variable elimination

$$\frac{p\ \wedge\ x = k\ \ \textit{unless}\ \ q}{p\ \ \textit{unless}\ \ q}$$

where x is a set of program variables and k is free.

Axiom (Substitution) The substitution axiom allows us to replace any invariant by *true*, and vice-versa, in any predicate occurring in a property. Thus, given that I is invariant, we may conclude from

$$p\ \wedge\ I\ \textit{unless}\ q$$

that

$$p\ \textit{unless}\ q$$

2.4 Program Composition by *union*

As in other programming theories, it is often convenient to view or design a UNITY program as a composition of several program components. In this paper, we consider a particularly simple kind of program composition: *union*. The union of programs F, G denoted by $F\ [\!]\ G$, is obtained by combining the appropriate portions of F and G together; in particular, the variables of $F\ [\!]\ G$ are the ones in F or G, a variable is initialized to a value as prescribed in F or in G (we assume that if the initial value of a variable is prescribed in both F and G then these initial values are identical), and the set of statements in the body of $F\ [\!]\ G$ is the union of the corresponding sets of F and G.

The union operation is often referred to as "parallel composition." This is the primary structuring mechanism for building networks of processes that communicate by messages or shared variables; see Chandy and Misra [1988] for details.

We describe one part of the union theorem which is fundamental for the study of the union operation; for the proof of this theorem, see Section 7.2.1 of Chandy and Misra [1988] and Misra [1990].

Union Theorem (Safety)

$$p\ \ \textit{unless}\ \ q\ \ \text{in}\ F\ \ [\!]\ \ G\ =\ (p\ \ \textit{unless}\ \ q\ \text{in}\ F)\ \wedge\ (p\ \ \textit{unless}\ \ q\ \text{in}\ G)$$

Corollary 1

$$p \text{ is stable in } F \;[\!]\; G \;=\; (p \text{ is stable in } F) \;\wedge\; (p \text{ is stable in } G)$$

Corollary 2

$$\frac{p \text{ is stable in } F \;,\; p \text{ is invariant in } G}{p \text{ is invariant in } F \;[\!]\; G}$$

A particularly useful observation about the stability of a predicate p is that p is stable in F if p mentions no variable that can be changed in F.

Note For a composite program, the substitution axiom can be applied only with an invariant of the composite program. Thus it is illegal to: deduce p *unless* q in F using the substitution axiom with an invariant of F, deduce p *unless* q in G using the substitution axiom with an invariant of G and then deduce p *unless* q in $F \;[\!]\; G$ applying the union theorem. Such a deduction is valid provided the substitution axiom is applied in each case with an invariant of $F \;[\!]\; G$.

3 Problem Description

Let program D denote the programs for the given set of message communicating processes. Properties of D are described in section 3.2 (some notations are introduced in section 3.1 to facilitate this description). We view the problem of termination detection as designing a recording program R such that in the composite program $D \;[\!]\; R$ some predicate p holds only if D is terminated; in our case, program R contains statements to record the states of each process and the number of messages it sends/receives along the channels incident on it, and predicate p states that the recorded states of all processes are idle and all channels are empty. Note that R only reads but does not write into the variables of D.

3.1 Notation

We use symbols i, j for processes and W, X, Y for sets of processes. The set of all processes (assumed finite and nonempty) is denoted by Z. The symbol $\overline{X}$ denotes the complement of X, i.e., the set of processes not in X. Let $q.i$ be *true* iff process i is idle; $q.X$ is the conjunction of $q.i$, for all i in X. Symbol c is used to denote a channel; $r.c$ and $s.c$ are the number of messages received along c and sent along c, respectively. The set of channels directed from processes in X to processes in Y will be denoted by XY. In particular, XZ is the set of all outgoing channels from the processes in X (including those directed between the processes in X) and ZX is the set of all incoming channels to the processes in X (including those directed between the processes in X). Hence, the set of outgoing channels of process i is denoted by iZ and the incoming ones by Zi; assume that no channel is directed from a process to itself, i.e., ii is empty for all i.

A particularly useful notational abbreviation for a predicate over a set of channels is $p.XY$, where p is of the form $(s = r)$, $(s \geq r)$ or $(s > r)$. These are defined as follows.

$$\begin{array}{lcl}
(s = r).XY & \equiv & \langle \forall\, c \;:\; c \text{ in } XY \;::\; s.c = r.c \rangle \\
(s \geq r).XY & \equiv & \langle \forall\, c \;:\; c \text{ in } XY \;::\; s.c \geq r.c \rangle \\
(s > r).XY & \equiv & (s \geq r).XY \;\wedge\; \neg(s = r).XY
\end{array}$$

Observe that for $p.XY$ of the first or the second form,

$$\begin{array}{l}
p.XY \;=\; \mathit{true} \quad \text{if} \quad X \text{ or } Y \text{ is empty} \\
p.(X \cup X')(Y \cup Y') = p.XY \;\wedge\; p.XY' \;\wedge\; p.X'Y \;\wedge\; p.YY'
\end{array}$$

In particular,

$$\begin{array}{lcl} p.XZ & = & p.XX \;\wedge\; p.X\overline{X} \\ p.ZX & = & p.XX \;\wedge\; p.\overline{X}X \end{array}$$

For $p.XY$ of the third form,

$$\begin{array}{lcll} p.XY & = & \mathit{false} & \text{if} \quad X \text{ or } Y \text{ is empty} \\ p.XZ & \Rightarrow & p.XX \;\vee\; p.X\overline{X} & \\ p.ZX & \Rightarrow & p.XX \;\vee\; p.\overline{X}X & \end{array}$$

Throughout this paper $L.XY$ denotes a free variable (of type, set of integers) which has one integer corresponding to each channel in XY.

3.2 Specification of Program D

Program D has the following properties. The number of messages sent along any channel is at least the number received, and both of these are nonnegative (D1). The number of messages received along a channel is nondecreasing; similarly the number of messages sent (D2). An idle process remains idle as long as it does not receive a message (D3) (it may stay idle after receiving a message). An idle process does not send messages (D4).

We reiterate that the notions of process, channel, message, etc. are outside the UNITY theory. Thus, program D manipulates the variables $q.i$, $r.c$, $s.c$, for all i and c, without assigning them meanings. The restrictions that we put on D for manipulations of these variables reflect the nature of a message communicating system; this has been described informally above, and is described formally next.

In (D1–D4) the properties are of program D. The symbols m, n denote arbitrary integer constants.

Property D1

$s.c \geq r.c \geq 0$ invariant

Property D2

$r.c \geq m$ stable

$s.c \geq n$ stable

Property D3

$q.i \;\wedge\; (r = L).Zi \;\mathit{unless}\; (r > L).Zi$

Property D4

$q.i \;\wedge\; (s = L).iZ \;\; \mathit{unless} \;\; \neg q.i \;\wedge\; (s = L).iZ$

The reader may understand D3,D4 better by writing them out in terms of pre- and post-conditions, using the definition of *unless*. For instance, D4 says that for any statement t in D,

$\{q.i \;\wedge\; (s = L).iZ\} \;\; t \;\; \{(s = L).iZ\}$,

That is, execution of a statement never causes an idle process to send a message.

Observation The variables $q.i$, $r.c$, $s.c$ are *local* to program D, i.e., they cannot be modified in any other program G. Therefore, using the union theorem and its corollaries, properties (D1–D4) are also properties of $D \parallel G$, for any G.

Note We have not specified that the channels be first-in-first-out, nor that every message be delivered eventually. These are not required for the correctness of the proposed termination detection algorithm. Also, several processes may receive and/or send messages in one step; however, from D4, a process may not receive a message, become nonidle, and send a message, all in one step. This restriction prevents cyclical dependence among sends and receives as in the following scenario. Idle processes A, B receive, become nonidle, and send messages, all in one step, where the message m sent by A is received by B which causes it to send m' to A which caused A to send m in the first place.

3.3 Some Derived Properties of Program D

In this section we derive some properties of program D, from D1–D4. The first property, D5, roughly says that for any set of processes X: if all processes in X are idle and for all internal channels of X (i.e., those between processes in X) number of sends and receives are equal, then they remain so and no message is sent by any process in X until a message is received by a process in X from outside X. Formally,

Property D5

$$\begin{array}{l} q.X \ \wedge \ (r = L).ZX \ \wedge \ (s = L).XZ \ \ \textit{unless} \\ \qquad (r > L).\overline{X}X \ \wedge \ (s = L).X\overline{X} \quad \text{in } D \end{array}$$

The result is proven by taking the conjunction of D3,D4 and then applying the general conjunction rule over all i in X. In the following proof, justifications for proof steps are enclosed within braces.

Proof Applying conjunction to D3 and D4,

$$q.i \ \wedge \ (r = L).Zi \ \wedge \ (s = L).iZ \ \ \textit{unless} \ \ (r > L).Zi \ \wedge \ (s = L).iZ$$

Applying conjunction over all i in X,

$$\begin{array}{lll} \text{lhs} & \equiv & \langle \forall\, i \ : \ i \in X \ :: \ q.i \ \wedge \ (r = L).Zi \ \wedge \ (s = L).iZ \rangle \\ & \equiv & q.X \ \wedge \ (r = L).ZX \ \wedge \ (s = L).XZ \end{array}$$

The right-hand side has two conjuncts. The first one is

$$\begin{array}{ll} \equiv & \langle \forall\, i \ : \ i \in X \ :: [q.i \ \wedge \ (r = L).Zi \ \wedge \ (s = L).iZ] \ \vee \\ & \qquad\qquad [(r > L).Zi \ \wedge \ (s = L).iZ] \rangle \\ \Rightarrow & (r \geq L).ZX \ \wedge \ (s = L).XZ \end{array}$$

and the second one is

$$\begin{array}{ll} \equiv & \langle \exists\, i \ : \ i \in X \ :: \ (r > L).Zi \ \wedge \ (s = L).iZ \rangle \\ \Rightarrow & \langle \exists\, i \ : \ i \in X \ :: \ (r > L).Zi \rangle \end{array}$$

Hence

$$
\begin{array}{lll}
\text{rhs} & \Rightarrow & (r \geq L).ZX \;\wedge\; (s = L).XZ \;\wedge\; \langle \exists\, i \;:\; i \in X \;::\; (r > L).Zi \rangle \\
& \Rightarrow & (r > L).ZX \;\wedge\; (s = L).XZ \\
& \Rightarrow & \{(r > L).ZX \Rightarrow (r > L).XX \;\vee\; (r > L).\overline{X}X \\
& & \;(s = L).XZ \;\Rightarrow\; (s = L).XX \;\Rightarrow\; \neg(r > L).XX \\
& & \;\text{(applying the substitution axiom; see the observation in Section 3.2)} \\
& & \;(s = L).XZ \;\Rightarrow\; (s = L).X\overline{X}\} \\
& & (r > L).\overline{X}X \;\wedge\; (s = L).X\overline{X}
\end{array}
$$
□

Note The left-hand side of D5 implies that the number of sends and receives are equal for all internal channels of X.

Note The operational argument for proving D5 is to assume that all processes in X are idle, sends and receives are equal for all internal channels of X and no message will be received by any process in X from outside X, and then show that all processes in X remain idle and no message will be sent by any process in X. The typical proof assumes the contrary—some process in X becomes nonidle—and derives a contradiction by noting that this process must have received a message from some process in X (because it is assumed that processes outside X do not send messages) which must be nonidle when it sends the message and therefore there is no first process in X that becomes nonidle. This temporal reasoning is completely avoided in our formalism.

Termination Define predicate T as follows.

$$T \;\equiv\; q.Z \;\wedge\; (s = r).ZZ$$

Program D is said to be *terminated* if T holds, i.e., all processes are idle and for each channel the number of message sends and receives are equal. It is not obvious that T is stable nor that message transmissions cease once T holds. (Stability of T does not guarantee that message transmissions cease once T holds; both $s.c, r.c$ may change simultaneously, for some c, while preserving $s.c = r.c$, and hence preserving T.)

Property D6

$T \;\wedge\; (s = L).ZZ$ stable in D

Proof Replace X by Z in D5. Note that the term $(r > L).\overline{Z}Z$ in the right-hand side of D5 is *false*, because $\overline{Z}$ is empty. □

Property D7

T stable in D

Proof Eliminating free variable $L.ZZ$ in D6. □

Summary of Properties of D

D1. $s.c \geq r.c \geq 0$ invariant

D2. $r.c \geq m$ stable,
$s.c \geq n$ stable

D3. $q.i \;\wedge\; (r = L).Zi$ *unless* $(r > L).Zi$

D4. $q.i \;\wedge\; (s = L).iZ$ *unless* $\neg q.i \;\wedge\; (s = L).iZ$

Derived Properties

D5. $q.X \ \wedge \ (r = L).ZX \ \wedge \ (s = L).XZ \ \ unless \ \ (r > L).\overline{X}X \ \wedge \ (s = L).X\overline{X}$

D6. $T \ \wedge \ (s = L).ZZ$ stable

D7. T stable

3.4 The Recording Program, R

Program R, given below, is used to record the values of variables of D. We introduce the following local variables of R, $vq.i, vr.c, vs.c$, in which variables $q.i, r.c, s.c$ are recorded, respectively, for all i and c. Initially the recorded values are $vq.i = false$, $vr.c = 0$, $vs.c = 0$, for all i and c. For each process i there is a statement in the program body—i.e., in the assign-section of the program—to assign simultaneously

1. $q.i$ to $vq.i$,
2. $r.c$ to $vr.c$, for all channels c incoming to i, and
3. $s.c$ to $vs.c$, for all channels c outgoing from i.

We give a very brief and incomplete description of UNITY syntax for explaining program R. The symbol $[\!]$ is used to separate statements in the assign-section and equations in the initially-section; the symbol $\|$ is used to separate the components of a single assignment. Angled brackets, $\langle$ and $\rangle$, denote quantification. Interpret

initially $\langle [\!] \ i \ \ :: \ \ vq.i = false \rangle$

to mean that for each i (where $i \in Z$) equation $vq.i = false$ holds initially in the program. Similarly, the quantified statement in the assign-section is to be interpreted as, for each i there is a single statement in the program; this single statement (for any i) consists of three components which must be executed simultaneously:

1. $vq.i \ := \ q.i$,
2. $\langle \| \ c \ : \ c \ \in \ Zi \ \ :: \ \ vr.c \ := \ r.c \rangle$, and
3. $\langle \| \ c \ : \ c \ \in \ iZ \ \ :: \ \ vs.c \ := \ s.c \rangle$.

The component $\langle \| \ c \ : \ c \ \in \ Zi \ \ :: \ \ vr.c \ := \ r.c \rangle$ is to be interpreted as: For all c, where c is in Zi, perform the assignment $vr.c \ := \ r.c$, simultaneously, and similarly for the last component.

Program R

initially

$$
\begin{array}{l}
\quad \langle [\!] \ i \ \ :: \ \ vq.i = false \rangle \\
[\!] \ \langle [\!] \ c \ \ :: \ \ vs.c, vr.c = 0, 0 \rangle
\end{array}
$$

assign

$$
\begin{array}{l}
\langle [\!] \ i \ \ :: \\
\quad\quad vq.i \ := \ q.i \\
\quad \| \langle \| \ c \ : \ c \ \in \ Zi \ \ :: \ \ vr.c \ := \ r.c \rangle \\
\quad \| \langle \| \ c \ : \ c \ \in \ iZ \ \ :: \ \ vs.c \ := \ s.c \rangle
\end{array}
$$

$\rangle$

end $\{R\}$

3.5 The Termination Detection Theorem

We have previously defined program D to be terminated if $q.Z \wedge (s = r).ZZ$, holds. The termination detection theorem says that D is terminated if the above condition, with q, s, r replaced by vq, vs, vr, holds.

Termination Detection Theorem

$$vq.Z \;\wedge\; (vs = vr).ZZ \;\;\Rightarrow\;\; q.Z \;\wedge\; (s = r).ZZ \qquad \text{is invariant in } D \;[\!]\; R$$

4 Proof of the Termination Detection Theorem

4.1 Informal Outline of the Proof

Consider some point during the execution of program $D \;[\!]\; R$ when, for some i, $vq.i \wedge (vr = r).Zi$ holds. From $vq.i$, we can claim that $q.i$ was *true* (process i was idle) when the last recording was made for process i. From $(vr = r).Zi$, we can claim that process i has received no message since the last recording. Therefore,

1. Process i is still idle, i.e., $q.i$ is *true*.
2. Process i has sent no message since the last recording, i.e., $(vs = s).iZ$ holds.

That is, we claim that

$$vq.i \;\wedge\; (vr = r).Zi \;\;\Rightarrow\;\; q.i \;\wedge\; (vs = s).iZ \quad \text{invariant} \quad \text{in } D \;[\!]\; R$$

We consider a generalization of the above property with i replaced by an arbitrary set X. How do we define the idleness of a set of processes X? That all processes in X are idle and all internal channels of X are empty. Thus, $q.i$ is generalized to $q.X \wedge (s = r).XX$. The appropriate generalization of $vq.i$ is $vq.X \wedge (vs = vr).XX$. Hence we postulate, for all X,

Property DR1

$$\begin{array}{l} vq.X \;\wedge\; (vs = vr).XX \;\wedge\; (vr = r).\overline{X}X \;\;\Rightarrow \\ \qquad q.X \;\wedge\; (s = r).XX \;\wedge\; (vs = s).X\overline{X} \\ \text{invariant in } D \;[\!]\; R \end{array}$$

The termination detection theorem follows from DR1 by setting X to Z (since $\overline{Z}$ is the empty set, $(vr = r).\overline{Z}Z = true$).

In the remaining parts of Section 4, we prove DR1. Using the Corollary 2 of the union theorem (section 3.4), we undertake to show that the proposition in DR1 is stable in D and invariant in R:

Property DR2

$$\begin{array}{l} \langle \forall\; X \;:: \\ \quad vq.X \;\wedge\; (vs = vr).XX \;\wedge\; (vr = r).\overline{X}X \;\;\Rightarrow \\ \qquad q.X \;\wedge\; (s = r).XX \;\wedge\; (vs = s).X\overline{X} \\ \rangle \\ \text{stable in } D \end{array}$$

Property DR3

$\langle \forall\ X\ ::$
$vq.X\ \wedge\ (vs = vr).XX\ \wedge\ (vr = r).\overline{X}X\ \ \Rightarrow$
$q.X\ \wedge\ (s = r).XX\ \wedge\ (vs = s).X\overline{X}$
$\rangle$
invariant in R

Proofs of DR2,DR3 are given in sections 4.2, 4.3, respectively. Property DR4, given below, is used in the proofs of DR2,DR3; we leave the (rather trivial) proof of DR4 to the reader; for the proof use Corollary 2 of the union theorem and that initially $s.c \geq vs.c$ and $r.c \geq vr.c$ in $D\ [\!]\ R$.

Property DR4

$s.c \geq vs.c \geq 0$ invariant in $D\ [\!]\ R$
$r.c \geq vr.c \geq 0$ invariant in $D\ [\!]\ R$.

4.2 Proof of DR2

The proof of DR2 uses D2 (see section 3.2), D5 (see section 3.3), DR4 (section 4.1), properties of *unless* (see section 2.3) and the fact that vq, vs, vr are constants in D. In the following proof all properties are of Program D. From D5,

$$q.X\ \wedge\ (r = L).ZX\ \wedge\ (s = L).XZ\ \ \textit{unless}\ \ (r > L).\overline{X}X\ \wedge\ (s = L).X\overline{X}$$

Rewrite the left-hand side using

$$(r = L).ZX\ \equiv\ (r = L).XX\ \wedge\ (r = L).\overline{X}X$$

$$(s = L).XZ\ \equiv\ (s = L).XX\ \wedge\ (s = L).X\overline{X}$$

Weaken the right-hand side to $(r > L).\overline{X}X$.

$$q.X\ \wedge\ (s = r).XX\ \wedge\ (r = L).XX\ \wedge\ (r = L).\overline{X}X\ \wedge\ (s = L).X\overline{X}$$
$$\textit{unless}\ \ (r > L).\overline{X}X$$

Eliminate free variables $L.XX$ and hence the term $(r = L).XX$ from the left-hand side.

$$q.X\ \wedge\ (s = r).XX\ \wedge\ (r = L).\overline{X}X\ \wedge\ (s = L).X\overline{X}\ \ \textit{unless}\ \ (r > L).\overline{X}X$$

From D2, $(r > L).\overline{X}X$ stable. Applying disjunction with the preceding,

$$(r > L).\overline{X}X\ \vee\ [q.X\ \wedge\ (s = r).XX\ \wedge\ (r = L).\overline{X}X\ \wedge\ (s = L).X\overline{X}]\ \text{stable}$$

Replace $L.\overline{X}X$ by $vr.\overline{X}X$ and $L.X\overline{X}$ by $vs.X\overline{X}$, respectively. The latter lists are constants in D and hence this instantiation is permissible.

$$(r > vr).\overline{X}X\ \vee\ [q.X\ \wedge\ (s = r).XX\ \wedge\ (r = vr).\overline{X}X\ \wedge\ (s = vs).X\overline{X}]\ \text{stable}$$

Since vq, vs, vr are constants in D, $\neg[vq.X\ \wedge\ (vs = vr).XX]$ is stable. Taking simple disjunction with the above and rewriting the expression,

$$[vq.X\ \wedge\ (vs = vr).XX\ \wedge\ \neg(r > vr).\overline{X}X]\ \ \Rightarrow$$
$$[q.X\ \wedge\ (s = r).XX\ \wedge\ (r = vr).\overline{X}X\ \wedge\ (s = vs).X\overline{X}]\ \text{stable}$$

From DR4 (Section 4.1), $(r \geq vr).\overline{X}X$. Hence, $\neg(r > vr).\overline{X}X\ \equiv\ (r = vr).\overline{X}X$. Using the substitution axiom replace this term in the antecedent. Also, the term $(r = vr).\overline{X}X$ can be dropped from the consequent of the implication:

$$[vq.X\ \wedge\ (vs = vr).XX\ \wedge\ (r = vr).\overline{X}X]\ \Rightarrow$$
$$[q.X\ \wedge\ (s = r).XX\ \wedge\ (s = vs).X\overline{X}]\ \text{stable}\ \ \ \Box$$

4.3 Proof of DR3

DR3 is of the form

$$\langle \forall\ Y\ \ ::\ \ vp.Y\ \Rightarrow\ p.Y \rangle \text{ invariant in } R$$

where

$$vp.Y \quad \equiv \quad vq.Y\ \wedge\ (vs = vr).YY\ \wedge\ (vr = r).\overline{Y}Y$$

and

$$p.Y \quad \equiv \quad q.Y\ \wedge\ (s = r).YY\ \wedge\ (vs = s).Y\overline{Y}$$

All properties in the rest of this section, 4.3, are of Program R. The proof of DR3 follows directly by using the "assignment axiom" because all statements in R are assignments. For completeness, we give this proof in some detail; however, the proof is easy and the reader is encouraged to construct the proof.

From the initial condition of R, each $vq.Y$ is false and hence, so is $vp.Y$. Thus $\langle \forall\ Y\ ::\ vp.Y\ \Rightarrow\ p.Y \rangle$ holds initially. Next, we prove the stability of $\langle \forall\ Y\ ::\ vp.Y\ \Rightarrow\ p.Y \rangle$. Consider any statement, $t.j$, that records for process j. We have to show that

$$\{\langle \forall\ Y\ ::\ vp.Y\ \Rightarrow\ p.Y \rangle\}\ \ t.j\ \ \{\langle \forall\ Y\ ::\ vp.Y\ \Rightarrow\ p.Y \rangle\}$$

We prove the postcondition for an arbitrary set of processes X. We apply the *assignment axiom* to compute a precondition from $vp.X\ \Rightarrow\ p.X$ by replacing in it all occurrences of $vq.j$ by $q.j$, $vr.c$ by $r.c$ for all c incoming to j and $vs.c$ by $s.c$ for all c outgoing from j.

If $j \notin X$ then the precondition as computed above is $vp.X \Rightarrow p.X$ and hence the proof is trivial. Therefore, consider the case where $j \in X$. Let $W = X - \{j\}$. We have, before execution of t_j (where $A \equiv vp.W$ and $B \equiv p.W$),

$$A\ \Rightarrow\ B$$

where

$$\begin{array}{lll} A & \equiv & vq.W\ \wedge\ (vs = vr).WW\ \wedge\ (vr = r).\overline{W}W \\ B & \equiv & q.W\ \wedge\ (s = r).WW\ \wedge\ (vs = s).W\overline{W} \end{array}$$

The precondition computed from $vp.X \Rightarrow p.X$ by applying the assignment axiom is

$$U\ \Rightarrow\ V$$

where

$$\begin{array}{lll} U & \equiv & [vq.W\ \wedge\ q.j]\ \wedge\ [(vs = vr).WW\ \wedge\ (s = vr).jW\ \wedge\ (vs = r).Wj] \\ & & \wedge\ [(vr = r).\overline{X}W\ \wedge\ (r = r).\overline{X}j] \\ V & \equiv & q.X\ \wedge\ (s = r).XX\ \wedge\ [(vs = s).W\overline{X}\ \wedge\ (s = s).j\overline{X}] \end{array}$$

We will show that

$$(A\ \ \Rightarrow\ \ B)\ \ \Rightarrow\ \ (U\ \ \Rightarrow\ \ V)$$

i.e., that

$$[U\ \wedge\ (A\ \ \Rightarrow\ \ B)]\ \ \Rightarrow\ \ V$$

We prove this in two steps:

1. $U \quad \Rightarrow \quad A$ (Lemma 2), and
2. $U \ \wedge \ B \quad \Rightarrow \quad V$ (Lemma 3).

Lemma 1

$U \quad \Rightarrow \quad (s = r = vr).jW$

Read $(s = r = vr).jw$ as $(s = r).jw \ \wedge \ (r = vr).jw\}$

Proof

$(s \geq r \geq vr).jW$	, from D1 and the substitution axiom used with DR4
$(s = vr).jW$	, from U
$(s = r = vr).jW$	, from the two previous steps

□

Lemma 2

$U \quad \Rightarrow \quad A$

Proof All conjuncts in A follow trivially from U except $(vr = r).\overline{W}W$. This follows from

$(vr = r).\overline{X}W$	, from U
$(vr = r).jW$	, from U using Lemma 1
$(vr = r).(\overline{X} \ \cup \ \{j\})W$	, from the two previous steps
$(vr = r).\overline{W}W$	, $\overline{W} = \overline{X} \ \cup \ \{j\}$

□

Lemma 3

$U \ \wedge \ B \quad \Rightarrow \quad V$

Proof We prove each conjunct of V in a separate sublemma.

Sublemma 1 $q.X$

$q.W$	, from B
$q.j$	, from U
$q.X$	, from the two previous steps because $X = W \ \cup \ \{j\}$

Sublemma 2 $(s = r).XX$

$(vs = s).W\overline{W}$	, from B
$(vs = s).Wj$	, from the preceding step because $j \in \overline{W}$
$(vs = r).Wj$	, from U
$(s = r).Wj$	, from the two previous steps
$(s = r).WW$	, from B
$(s = r).jW$	, from U using Lemma 1
$(s = r).XX$	, from the three previous steps because $X = W \ \cup \ \{j\}$

Sublemma 3 $(vs = s).W\overline{X}$

$(vs = s).W\overline{W}$, from B
$(vs = s).W\overline{X}$, from the preceding step because $\overline{X} \subseteq \overline{W}$ $\square$

5 UNITY Logic: Progress

Two logical operators, *ensures* and *leads-to* (also written as $\mapsto$), defined on pairs of predicates, are used to prove progress properties of UNITY programs. Briefly, p *ensures* q says that p remains *true* as long as q is not *true* (i.e., p *unless* q) and there is a statement in the program whose execution establishes q, starting in any state that satisfies $p \wedge \neg q$. The operator *leads-to* is the transitive, disjunctive closure of *ensures*: $p \mapsto q$ means that if p holds at any point during the program execution then q holds eventually.

In most cases we are interested in establishing *leads-to* properties. The reason we introduce *ensures* is two-fold:

1. The only way to establish *leads-to* is by using a set of ensures properties (*ensures* serves as the basis of induction for defining *leads-to*).
2. It is possible to formulate a union theorem, akin to the union theorem of section 2.4, for *ensures* but there is no analogous theorem for *leads-to*.

5.1 The Logical Operator *ensures*

For a program F, we define the operator *ensures* by the following inference rule.

$$\frac{p \;\; unless \;\; q \text{ in } F \;\wedge\; \langle \exists\, t \;:\; t \text{ is a statement in } F \;\; :: \;\; \{p \wedge \neg q\} \;\; t \;\; \{q\} \rangle}{p \;\; ensures \;\; q \text{ in } F}$$

From the hypothesis of the inference rule it follows that once p is *true* p remains *true* as long as q is not—from p *unless* q—and there is a statement t in F whose execution with precondition $p \wedge \neg q$ establishes q. From our fairness assumption, the statement t is executed sometime after p becomes *true* and hence, once p is *true* q is established eventually. Analogous to the union theorem of section 2.4 we have,

Union Theorem (Progress)

$$\begin{aligned} p \;\; ensures \;\; q \text{ in } F \;\; [\!]\;\; G = \; & (p \; unless \; q \text{ in } F \;\wedge\; p \; ensures \; q \text{ in } G) \;\vee \\ & (p \; ensures \; q \text{ in } F \;\wedge\; p \; unless \; q \text{ in } G) \end{aligned}$$

Corollary 3

$$\frac{p \text{ is stable in } F \;,\; p \; ensures \; q \text{ in } G}{p \; ensures \; q \text{ in } F \;\; [\!] \;\; G}$$

5.2 The Logical Operator leads-to

For a program F, p *leads-to* q, typically written as $p \mapsto q$, is defined as the strongest relation satisfying the following (i.e., $p \mapsto q$ can be deduced only by applying the following rules). The program name, F, is omitted in the following discussion.

1. $\dfrac{p \; ensures \; q}{p \mapsto q}$

2. (Transitivity)

$$\frac{p \;\mapsto\; q \;,\; q \;\mapsto\; r}{p \;\mapsto\; r}$$

3. (Disjunction) Let $p.m$ be a predicate with a free variable m that ranges over any arbitrary set W and does not occur free in q.

$$\frac{\langle \forall\; m \quad :: \quad p.m \;\mapsto\; q \rangle}{\langle \exists\; m \quad :: \quad p.m \rangle \;\mapsto\; q}$$

We state two results in connection with *leads-to* that are used in this paper.

1. (Implication Theorem)

$$\frac{p \quad \Rightarrow \quad q}{p \quad \mapsto \quad q}$$

2. (Completion Theorem; special case) In the following, m is quantified over any finite set:

$$\frac{\begin{array}{c}\langle \forall\; m \quad :: \quad p.m \;\mapsto\; q.m \rangle, \\ \langle \forall\; m \quad :: \quad q.m \text{ stable} \rangle\end{array}}{\langle \forall\; m \quad :: \quad p.m \rangle \quad \mapsto \quad \langle \forall\; m \quad :: \quad q.m \rangle}$$

6 Progress of the Termination Detection Algorithm

We showed in Section 4 that

$$vq.Z \;\wedge\; (vs = vr).ZZ \quad \Rightarrow \quad T \quad \text{in } D \;\|\; R$$

where T, as defined in Section 3.3, is $q.Z \;\wedge\; (s = r).ZZ$. Thus it is safe to report termination if the left side of the above implication holds. We now show that

Property DR5

$$T \quad \mapsto \quad vq.Z \;\wedge\; (vs = vr).ZZ \quad \text{in } D \;\|\; R$$

That is, $vq.Z \;\wedge\; (vs = vr).ZZ$ becomes *true* within a finite time of termination.

We begin with a preliminary result. Define $u.i$ as follows:

$$u.i \quad \equiv \quad (vq.i = q.i) \;\wedge\; (vr = r).Zi \;\wedge\; (vs = s).iZ$$

Property D8

$$T \;\wedge\; u.i \text{ stable in } D$$

Proof All properties in the following proof refer to D. Let $Y = Z - \{i\}$.

$T \wedge (s = L).ZZ$ stable
  , from D6 (Section 3.3)
$T \wedge (s = L).Zi \wedge (s = L).iZ \wedge (s = L).YY$ stable
  , from the above: $Z = Y \cup \{i\}$
$T \wedge (s = L).Zi \wedge (s = L).iZ$ stable
  , eliminating free variable $L.YY$
$vq.i$ stable
  , $vq.i$ is constant in D
$T \wedge vq.i \wedge (s = L).Zi \wedge (s = L).iZ$ stable
  , simple conjunction of the previous two
$T \wedge (vq.i = q.i) \wedge (r = L).Zi \wedge (s = L).iZ$ stable
  , $T \wedge vq.i \equiv T \wedge (vq.i = q.i)$; $T \wedge (s = L).Zi \equiv$
  $T \wedge (r = L).Zi$
$T \wedge (vq.i = q.i) \wedge (r = vr).Zi \wedge (s = vs).iZ$ stable
  , replacing $L.Zi, L.iZ$ by $vr.Zi, vs.iZ$, which are constants in D. □

Now we can present the proof of Property DR5.

Proof

T *ensures* $T \wedge u.i$ in R
  , from the text of R
T *ensures* $T \wedge u.i$ in $D \,[\!]\, R$
  , applying Corollary 3 on D7 (T stable in D) and the preceding
$T \mapsto T \wedge u.i$ in $D \,[\!]\, R$
  , applying the definition of *leads-to* to the preceding (1)
T stable in R
  , all variables in T are constants in R
$u.i$ stable in R
  , from the text of R
$T \wedge u.i$ stable in R
  , simple conjunction on the preceding two
$T \wedge u.i$ stable in $D \,[\!]\, R$
  , applying Corollary 1 on D8 ($T \wedge u.i$ stable in D) and the preceding
$T \mapsto \langle \forall\, i \;::\; T \wedge u.i \rangle$
  , completion theorem on (1) and the preceding
$\langle \forall\, i \;::\; T \wedge u.i \rangle \equiv T \wedge \langle \forall\, i \;::\; u.i \rangle$
  , predicate calculus
$T \wedge \langle \forall\, i \;::\; u.i \rangle \Rightarrow vq.Z \wedge (vs = vr).ZZ$
  , from the definitions of T and $u.i$
$T \mapsto vq.Z \wedge (vs = vr).ZZ$
  , using transitivity and implication on the previous three □

7 Discussion

7.1 Why Bother with Formal Specifications?

It may be argued that since we could construct an informal proof of the correctness of the termination detection algorithm—see Section 1—without defining the properties of D precisely, formal

specifications are merely “icing on the cake.” This argument is often valid because formal specifications are rarely used in a constructive manner to derive other properties of programs, or aid in program designs. There seems to be little reason for formalism, except to avoid ambiguity, if the goal is merely to reach agreement among a group of designers.

Once we start using the specification for deductions and program designs, however, informal specifications doom us to common sense reasoning, a costly and error-prone procedure. Formal specifications force us to state (1) not too much, because then we cannot manipulate them as effectively and the specification will not apply to a broad range of systems, and (2) not too little, because then we cannot deduce the properties we want to hold. Thus, while properties D1 and D2 of program D—that the number of receives along a channel never exceeds the number of sends, and that numbers of sends and receives along a channel never decrease—would be taken for granted in informal discussion, we are forced to write them out in a formal specification. Most informal proofs would assume D5, D6, D7 as properties of program D; we believe that it is interesting in its own right to prove these from the simpler properties, D1–D4.

7.2 Proof Length

It should be clear by now that our formal proof consists of a relatively small number of proof steps; most of the proof consists of explaining the notation and relating the proof to its informal counterpart. The short informal proof, given in Section 1, suffers from many deficiencies; among them, what can be assumed about message communication systems—e.g., can a message be sent and received simultaneously, is FIFO order on channels required, etc.—and what are the precise steps of the recording algorithm. The great virtue of informal reasoning (besides being easily accessible and hence, more democratic) is that reasoning is carried out with a description that is at a far higher level than what is available in traditional programming languages; thus, we can informally talk about “search the list from left to right” as a single program step. In UNITY we have attempted to combine this flavor of high level description with precision of a formal language.

7.3 Proof Structure

Some features of the UNITY-style proof are worth noting. First, we associate properties with *programs*, not program points. Proofs of program properties are, therefore, carried out in the style of formal proofs in mathematics, outside the program text. The number of references to the program text in a proof is quite small: in the current proof, there is one reference to the text of R in section 4.3 (application of the assignment axiom) and two references in the proof of DR5, in section 6.

Second, associating properties with programs makes it simpler to construct “compositional” proofs whereby properties of a composite program, say $F \parallel G$, are deduced from the properties of its components, F and G. This is a particularly attractive feature because it allows us to work with specifications in the absence of code; see, for instance, the way in which the specification of D (in the absence of the code of D) was used in the current proof. This is in contrast to proofs by noninterference, as advocated in Owicki and Gries [1976], where proofs are intimately tied to program codes.

Third, the success of a formal system relies crucially on a rich body of derived rules that can be exploited effectively in practice. We have stated a few derived rules about our logical operators—*unless*, *ensures*, *leads-to*—in this paper, and we have applied these effectively in the proof.

Fourth, as we have said earlier, the UNITY theory does not include “process” as a basic construct. This is a deliberate decision. One outcome of this decision is that we are able to study program composition as a concept orthogonal to composition of processes. For instance, we have viewed the overall program as the union of two programs, D and R, and we partitioned the proof obligation suitably between these two programs. Program D represents the basic computations of *all* processes; program R represents the recording actions of *all* processes. Clearly, $D \parallel R$ can be implemented

on a set of communicating processes, but it is considerably more difficult to partition the proof obligation among these processes, or to construct the proof in a manner independent of the schedule for recording states and message counts.

Acknowledgments

We are grateful to the Austin Tuesday Afternoon Club, under the guidance of Edsger W. Dijkstra, for initiating discussions about the proof of this problem, to Ernie Cohen, Wim Hesselink, and Devendra Kumar for showing us alternative proofs, to Wim Feijen for many constructive criticisms during the development of this proof, and to the participants in the Workshop on "Concurrency in Hardware and Software" (Workshop Director, Alain J. Martin), La Jolla, California, February 22–26, 1988, for comments on this proof. We are indebted to Alan Fekete, Jürg Gutknecht, Bengt Jonsson, and Martin Rem for their comments on the first draft; particular thanks to Nissim Francez, C. A. R. Hoare and J. R. Rao for their insightful comments.

References

1. Chandy, K. Mani [1983]. Unpublished notes, 1983.
2. Chandy, K. Mani [1987]. "A Theorem on Termination of Distributed Systems," TR-87-09, March 1987, Dept. of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712–1188, 1987.
3. Chandy, K. Mani, and Jayadev Misra [1988]. *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts, 1988.
4. Dijkstra, E. W. [1976]. *A Discipline of Programming*, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.
5. Francez, N. [1986]. *Fairness*, New York: Springer-Verlag, 1986.
6. Helary, M., Jard, C., Plouzeau, N., and M. Raynal [1987]. "Detection of Stable Properties in Distributed Applications," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 125–136, 1987.
7. Hesselink, W. H. H. [1987]. "Chandy's theorem on termination detection," Wim H. Hesselink, WHH4, Austin, Texas, January 21, 1987.
8. Hoare, C. A. R. [1969]. "An Axiomatic Basis for Computer Programming," *C. ACM*, 12, pp. 576–580, 1969.
9. Kumar, Devendra [1987]. "Efficient Algorithms for Distributed Simulation and Related Problems," Ph.D. Thesis, U.T., 1987.
10. Manna, Z. and A. Pnueli, *The Temporal Logic of Reactive Systems*, Springer-Verlag, Berlin (to appear).
11. Misra, J. [1986]. "Distributed Discrete Event Simulation," (see the last paragraph of p.62 and the first paragraph of p.63), *Computing Surveys*, Vol. 18, No. 1, pp. 39–66, March 1986.
12. Misra, J. [1988]. "General Conjunction and Disjunction of *unless*," *Notes on UNITY 01-88*, The University of Texas at Austin, 1988.
13. Misra, J. [1990]. "Soundness of the Substitution Axiom," *Notes on UNITY 14-90*, The University of Texas at Austin, 1990.

14. Owicki, S., and David Gries [1976]. "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, 6:1, pp. 319–340, 1976.

15. Owicki, S., and Leslie Lamport [1982]. "Proving Liveness Properties of Concurrent Programs," *ACM TOPLAS*, 4:3, pp. 455–495, July 1982.