An Introduction to Assertional Reasoning for Concurrent Systems

A. UDAYA SHANKAR

Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742

This is a tutorial introduction to assertional reasoning based on temporal logic. The objective is to provide a working familiarity with the technique. We use a simple system model and a simple proof system, and we keep to a minimum the treatment of issues such as soundness, completeness, compositionality, and abstraction. We model a concurrent system by a state transition system and fairness requirements. We reason about such systems using Hoare logic and a subset of linear-time temporal logic, specifically, invariant assertions and leads-to assertions. We apply the method to several examples.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.1 [**Logics and Meanings of Programs**]· Specifying and Verifying and Reasoning about Programs—*assertions*; *invariants*; *logics of programs*; *pre- and post-conditions*

General Terms: Verification

Additional Key Words and Phrases: Assertional reasoning, generation of preconditions, Hoare logic, invariants, leads-to, progress properties, safety properties, state transition systems, weakest preconditions

1. INTRODUCTION

A concurrent system consists of several processes that execute simultaneously and interact with each other *during* the course of their execution via shared variables or message passing. This is unlike a sequential system consisting of a single process that interacts with its environment only at the start and the end of its execution. Indeed, many concurrent systems are useful only while executing, for example, operating systems, communication networks, etc. Various approaches have recently been proposed to specify and analyze concurrent systems. One such approach is assertional reasoning based on temporal logic.[1]

Reasoning assertionally about programs is not new. Floyd [1967] introduced assertional reasoning for sequential programs, and Hoare formulated this as a logic, namely, Hoare logic [Hoare

[1] Other approaches, such as CCS [Milner 1989] and CSP [Hoare 1985], will not be covered in this tutorial.

This work was supported in part by National Science Foundation grant NCR-890450.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its data appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish requires a fee and/or specific permission.
© 1993 ACM 0360-0300/93/0900-0225 $01.50

CONTENTS

1969; Apt 1981]. Dijkstra [1976] extended this to nondeterministic sequential programs with guarded commands and introduced weakest preconditions. Reasoning about concurrent programs involves several extensions. First, because the processes of a concurrent program interact (and interfere) with each other, it is necessary to assume some level of atomicity in their interaction [Dijkstra 1965]. Second, the properties of interest for concurrent programs are more complex than for sequential programs; we are interested, for example, in infinite executions where every nonblocked process eventually executes some statement, where every request is eventually satisfied, etc. [Pnueli 1977]. Invariants and termination suffice for sequential programs, but not for concurrent programs.

Pnueli [1977; 1979] pioneered the use of temporal logic for reasoning formally about the properties of concurrent systems. Since then, various assertional methods based on temporal logic formalisms have been proposed, for example, Manna and Pnueli [1984; 1992], Owicki and Lamport [1982], Lamport [1983b; 1989; 1991], Chandy and Misra [1986; 1988], Back and Kurki-Suonio [1983; 1988], Lynch and Tuttle [1987], Schneider and Andrews [1986], Lam and Shankar [1990]. In these methods, we reason about a concurrent system using two kinds of assertions, referred to as **safety assertions** and **progress assertions**[2] [Lamport 1977]. Informally, a safety assertion states that "nothing bad can happen" (e.g., variable x never exceeds 5), while a progress assertion states that "something good will eventually happen" (e.g., x will eventually become 5). (The distinction is made precise at the end of Section 3.2.) With these two kinds of assertions, we can reason about any property that holds for a concurrent system iff it holds for every possible execution of the system. This class of properties includes partial correctness, invariants, termination, deadlock freedom, livelock freedom, worst-case complexity, hard real-time properties, etc. It does not include properties that involve operations over all possible behaviors, such as average complexity, probability distribution of response time, etc.

Assertional reasoning can be done at a formal level using temporal logic proof systems. However, we emphasize that assertional reasoning is not just about proofs. More importantly, it is a convenient language for talking unambiguously about concurrent systems, so that

[2] Progress assertions are also referred to as liveness assertions The term "progress" is used in Chandy and Misra [1986; 1988].

one person's view of a concurrent system can be communicated to another without distortion.

The objective of this tutorial is to provide a working familiarity with assertional reasoning, so that the reader can apply it to concurrent and distributed systems of interest. Our approach is to use a simple state transition representation of concurrent systems and a simple proof system. We illustrate this approach on a variety of examples. We place minimum emphasis on theoretical issues such as models, semantics, soundness, completeness, etc., because we feel that these issues are concerned with the foundation, rather than the application, of assertional reasoning. We do not use a compositional model or proof system, because we feel that it would hinder, rather than help, the novice. We do, however, discuss compositional approaches in the conclusion.

The formalism we utilize is taken to a large extent from Lam and Shankar [1990] and Shankar and Lam [1987]. But the basic ideas are present in much of the recent work in assertional reasoning, including the temporal-logic approach of Manna and Pnueli [1984; 1992], UNITY [Chandy and Misra 1986; 1988], TLA [Lamport 1991], and I/O automata [Lynch and Tuttle 1987].

This tutorial is organized as follows. In Section 2, with the help of a mutual-exclusion algorithm, we informally introduce the basic notions of our approach, namely, state transition system, fairness requirements, invariant and leads-to assertions, and proof rules. In Sections 3 and 4, we present these notions formally and describe how our approach fits in with those of others. Section 3 deals with the system model and assertion language, and Section 4 deals with proof rules and methods to apply them. In Section 5, we present examples of concurrent systems and their analyses. In Section 6, we specialize the system model for distributed systems. In Section 7, we present examples of distributed systems and their analyses. In Section 8, we motivate compositional and refinement techniques and review some of the literature in this area.

2. BASIC NOTIONS

Let us consider a simple solution by Peterson [1981] to the mutual exclusion problem involving two processes, 1 and 2. Each process can access a "critical section" (which may represent, for example, access to a shared data structure). The purpose is to ensure that (1) while process i is accessing the critical section, the other process j does not access it also, and (2) if a process i wants to access the critical section, then process i is eventually allowed to access it. The first is a safety property, and the second is a progress property. Throughout this example, we use i and j to identify the processes where $i \neq j$.

The solution uses two boolean variables, $thinking_1$ and $thinking_2$, and a variable $turn$ that takes values from $\{1, 2\}$. $thinking_i = true$ means process i is not interested in accessing the critical section; it is initially true. $turn$ is used to resolve contention. If process i wants to enter the critical section, it sets $thinking_i$ to *false* and $turn$ to j, and it then checks the values of $thinking_j$ and $turn$. Process i enters the critical section only if $thinking_j = true$ or $turn = i$. When it leaves the critical section, it sets $thinking_i$ to true. Expressed in a traditional concurrent programming language, the solution is as follows; process 1 executes the first loop below; process 2 executes the second loop:[3]

```
cobegin
    repeat
        S1: $thinking_1 \leftarrow false$;
        S2: $turn \leftarrow 2$;
        S3: await $thinking_2 \vee turn = 1$;
            "critical section";
        S4: $thinking_1 \leftarrow true$;
    forever
```

[3]We have simplified Peterson's algorithm here by using await statements. As usual, we assume that the boolean guard of an await statement is evaluated atomically and that control goes to the next statement only if the guard is true.

```
||
  repeat
    R1. $thinking_2 \leftarrow false$,
    R2: $turn \leftarrow 1$;
    R3: await $thinking_1 \lor turn = 2$,
        "critical section";
    R4: $thinking_2 \leftarrow true$;
  forever
cobegin
```

Because we do not make any assumption about the relative speeds of the two processes, the algorithm has many possible executions. How can we prove that the algorithm is correct, i.e., each of its executions satisfies the safety and progress properties? The usual approach is to consider some example executions.

2.1 Safety

Consider the safety property. Suppose process 1 enters the critical section, say at time t_1. We want to show that process 2 is not in the critical section at the same time. At t_1, $thinking_2 \lor turn = 1$ held; otherwise process 1 would not have entered the critical section. There are two cases to consider:

(a) Suppose $thinking_2$ held. Then process 2 is not in the critical section, because process 2 sets $thinking_2$ to false before it attempts to enter the critical section, and sets $thinking_2$ to true only after it leaves the critical section.

(b) Suppose $\neg thinking_2 \land turn = 1$ held. Let us assume process 2 is in the critical section and reach a contradiction. Let $t_2 < t_1$ be the time process 2 entered the critical section. Thus, $thinking_1 \lor turn = 2$ held at t_2. If $thinking_1$ held at t_2, then at some time t_3 between t_2 and t_1, process 1 executed S2 setting *turn* to 2. Hence at t_1, *turn* should equal 2, which is a contradiction. If $\neg thinking_1 \land turn = 2$ held at t_2, then again *turn* should equal 2 at t_1, since process 1 cannot set *turn* to 1. Again we have a contradiction.

This kind of reasoning is referred to as operational (or behavioral) reasoning. The basic idea is to examine example execution sequences and to show that each case satisfies the desired property. In general, it is hard to be sure that we have exhausted all the possible executions. Operational reasoning is useful because it gives insight, but it is prone to errors.

Assertional reasoning works differently. Instead of examining different execution sequences, we successively formulate **assertions**. Each assertion states a property about *all* executions of the program. The final assertion implies the desired property. Each assertion in the succession is proved by applying a **proof rule**. A proof rule consists of a list of conditions and a conclusion, such that if the conditions are satisfied by the program, then we can infer that the conclusion is satisfied by the program. Furthermore, the soundness of a rule should be "obvious" and not depend on the particular program or property being examined.

In this tutorial, we use **invariant** assertions for expressing safety properties. An invariant assertion has the form *Invariant*(P), where P is a predicate (boolean expression) in the program variables. *Invariant*(P) states that at any instant during execution the values of the program variables satisfy P. A proof rule for invariance, informally stated, is the following: *Invariant*(P) holds if (i) P holds initially and (ii) every statement S of the program preserves P, i.e., executing S when P holds leaving P holding.

Let us prove the mutual-exclusion property assertionally. The property is expressed by *Invariant*(A_0), where:

$$A_0 \equiv \neg(\text{process 1 at S4} \land \text{process 2 at R4})$$

We start by considering the following predicates:

$$A_1 \equiv \text{process 1 at S2, S3, or S4} \Leftrightarrow \neg thinking_1$$

$$A_2 \equiv \text{process 2 at R2, R3, or R4} \Leftrightarrow \neg thinking_2$$

We can prove that *Invariant*(A_1) holds by applying the invariance proof rule described above. Here are the details: Initially A_1 holds, because process 1 is at S1 and $thinking_1$ is true. S1 preserves A_1 because it makes both sides of the $\Leftrightarrow$ true. S2 preserves A_1 because both sides are true before execution (since process 2 is at S2, and we can assume A_1), and S2 leaves both sides true. S3 preserves A_1 in the same way. S4 preserves A_1 because it makes both sides false. Every Ri preserves A_1 because it does not affect A_1.

The proof for $Invariant(A_2)$ is symmetric; simply interchange process 1 with process 2, and $thinking_1$ with $thinking_2$.

We now consider another pair of predicates:

$$A_3 \equiv \text{process 1 at S4} \Rightarrow thinking_2 \vee turn = 1 \vee \textit{process 2 at R2}$$

$$A_4 \equiv \text{process 2 at R4} \Rightarrow thinking_1 \vee turn = 2 \vee \textit{process 1 at S2}$$

$Invariant(A_3)$ holds by the invariance rule. (Initially, A_3 holds vacuously. S1, S2 and S4 establish A_3 vacuously by falsifying the antecedent. S3 preserves A_3 because it is executed only if $thinking_2 \vee turn = 1$. R1 preserves A_3 because it establishes one disjunct in A_3's consequent, namely, "process 2 at R2." R2 and R4 preserve A_3 in the same way; each of them establishes one disjunct in A_3's consequent.) $Invariant(A_4)$ follows by symmetry.

We have established that $Invariant(A_i)$ holds, for $i = 1,2,3,4$. It is easy to show that A_1, A_2, A_3, and A_4 imply A_0 by predicate logic. (Assume $\neg A_0$ and reach a contradiction: $\neg A_0$ and A_3 imply $thinking_2 \vee turn = 1$. $\neg A_0$ and A_4 imply $thinking_1 \vee turn = 2$. $\neg A_0$ and A_1 and A_2 imply $\neg thinking_1$ and $\neg thinking_2$. Thus, we have $turn = 1$ and $turn = 2$, which is a contradiction.) Hence $Invariant(A_0)$ holds.

The assertional argument differs from the previous operational argument in two important ways. First, properties are stated precisely. This is important for communication: We may state an assertion that is not valid, for example, *Invariant*(process 1 at S3 $\Rightarrow$ process 2 not at R3), but there is no ambiguity as to what it means. Second, the assertional proof is in terms of applications of proof rules. It can be checked by examining the individual statements of the algorithm without understanding the algorithm. This seems central to what a "proof" is all about. It may take an expert to invent an algorithm or a proof (whether operational or assertional), but it should not require an expert to check a proof. In fact, the assertional approach lends itself to mechanization, i.e., the process of checking a proof can be completely automated, and this motivates much of the work in theorem proving.

A common complaint about assertional proofs of concurrent algorithms is that they are too tedious or too difficult and that the insight gained in doing operational reasoning is often lost or buried among the tedious and "syntactic" details of assertional proofs. We disagree. It is concurrent algorithms, and not assertional proofs, that are difficult to understand.

An assertional proof can be made brief and insightful just like any other proof: by omitting steps and unnecessary detail; e.g., in the above proof, maybe the invariance of A_1 and A_2 is obvious, but that of A_3 and A_4 is not. If you think a property is obvious, then state it assertionally (so we know what you mean); assume it without proof; and use it to prove other properties. Proof rules can be formulated to allow such reasoning. An example is the following generalization of the invariance rule: $Invariant(P)$ holds if, for some predicate Q, (i) $Invariant(Q)$ holds, (ii) P holds initially, and (iii) for every statement S of the program, executing S when $P \wedge Q$ holds leaves P holding. This rule allows us to assume the invariance of Q, and we make use of it in proving $Invariant(P)$. (It simplifies to the former rule if $Q \equiv true$.)

2.2 Progress and Fairness

Let us give an operational proof of the progress property of the mutual exclusion algorithm. Suppose process 1 wants to enter the critical section. It sets $thinking_1$ to false, *turn* to 2, and checks for $thinking_2 \vee turn = 1$, say, at time t_1. If process 2 is not attempting to enter the critical section, then $thinking_2$ is true, and process 1 enters the critical section. If process 2 is also attempting to enter the critical section, then $thinking_2$ will be false, and the value of *turn* determines which process enters first. There are two cases to consider:

(a) If process 2 was the last process to assign *turn*, then *turn* equals 1 at t_1, and process 1 enters the critical section first.

(b) If process 1 was the last process to assign *turn*, then *turn* equals 2 at t_1. Process 2 enters the critical section and process 1 has to wait. Assuming that process 2 does not stay indefinitely in the critical section, at some time t_2 ($> t_1$) it sets $thinking_2$ to true. At this point, one of two things can happen:

(b1) Process 1 enters the critical section.

(b2) Process 1 is slower than process 2, and before it can execute S3 process 2 executes R1, say at time $t_3 > t_2$. Then process 2 is again blocked. However, process 2 will soon execute R2 setting *turn* to 1, at which point process 1 becomes unblocked. More importantly, it remains unblocked because process 2 can never set *turn* to 2. Thus process 1 eventually enters the critical section.

Clearly, there are more cases here than in the safety proof. But there is also a fundamental difference. In the case of safety, we do not care if a process takes forever to execute a statement. But in the case of progress, that is not true in general. Consider process 2. If it is thinking, we do not care if it ever executes R1. But if it does execute R1, then it must execute R2 in finite time; otherwise process 1 can wait forever (case b2). For the same reason, process 2 must also not delay indefinitely the execution of R3 and R4.

Because R3 is an *await* statement whose condition ($thinking_1 \vee turn = 2$) can be enabled and disabled by process 1, we need to be more precise in what we mean by "process 2 must not delay indefinitely the execution of R3." Assume process 2 is at R3.

- One interpretation is that if R3 remains continuously enabled, then process 2 eventually executes R3. We refer to this as "process 2 executes R3 with **weak fairness**."
- Another interpretation is that if process 2 is at R3 and R3 becomes enabled and disabled repeatedly,[4] then process 2 eventually executes R3 in one of its enabled periods. We refer to this as "process 2 executes R3 with **strong fairness**."

The notions of weak fairness and strong fairness were introduced in Lehmann et al. [1981]. A detailed treatment of these (and other kinds of fairness) can be found in Francez [1986] and Manna and Pnueli [1992]. Weak fairness for a statement is easy to implement,[5] whereas strong fairness is not [Sistla 1984; Apt et al. 1988]. It turns out that strong fairness implies weak fairness (see Section 3.1). The fairness terminology can be extended to statements like R1 and R4, simply by considering them to be always enabled. Weak fairness suffices for such statements because they cannot be disabled by another process.

To summarize, we want R2, R3, R4, S2, S3, and S4 to be executed with weak fairness. Note that R1 and S1 are the only statements that need not be executed with any fairness. Thus, if both processes are thinking, the system can stay in that state forever. In any other situation, the system is not "at rest" in

[4] This can happen if, for example, the program executed by process 1 is changed to *repeat* $turn \leftarrow 1$; $turn \leftarrow 2$ *forever*

[5] For example, in a multiprogrammed environment by simply using a FIFO queue for scheduling enabled statements.

that it will eventually execute some statement.

To reason assertionally about progress, we consider assertions of the form P *leads-to* Q, where P and Q are predicates in the program variables. It means that if at some instant the program variables satisfy P, then at some later instant they satisfy Q. How long do we have to wait? No *a priori* time bound is implied, but we do know that it must hold whenever the system reaches a state where it is at "rest," i.e., no process is at an enabled statement subject to fairness.

To prove assertions like P *leads-to* Q, we rely on two kinds of proof rules. The first kind is for inferring leads-to assertions from program specification and fairness requirements. An example is the **leads-to via** S rule, where S is a program statement that is executed with weak fairness: P *leads-to* Q holds *via* S if (i) whenever P holds, S can be executed and its execution results in Q holding and (ii) for every statement R other than S, if P holds and R can be executed, its execution results in $P \vee Q$ holding. Note that the leads-to via S rule requires us to examine every statement of the program, and not just S. This rule implies that once P holds, it will continue to hold (at least) until Q becomes true, and that there is at least one statement, S, which will eventually make Q hold (if no other statement makes it so).

The second kind of proof rule is for inferring leads-to assertions from other leads-to assertions. For example, P *leads-to* Q holds if P *leads-to* R and R *leads-to* Q hold, or if P *leads-to* $(Q \vee R)$ and R *leads-to* Q hold. Such rules are called **closure** rules.

Now let us give an assertional proof of progress for the mutual exclusion algorithm. Suppose process 1 wants to access the critical section. It executes S1 and S2, at which point the following boolean condition holds:

$X_1 \equiv \neg\, thinking_1 \wedge$ process 1 at S3 $\wedge\, turn = 2$

Thus the desired progress property can be expressed by:

$Y_1 \equiv X_1$ *leads-to* process 1 at S4

Suppose X_1 holds, and process 2 is at R1. What can happen next? S3 can be executed because $thinking_2$ is true (which follows from $Invariant(A_1)$ and process 2 being at R1); its execution results in process 1 being at S4. Or R1 can be executed, resulting in X_1 and process 2 at R2. No other statement can be executed. Because S3 has weak fairness, process 1 will eventually execute S3 unless process 2 executes R1. In fact, we have just given the details of applying the leads-to via S rule to establish

$Y_2 \equiv X_1 \wedge$ process 2 at R1 *leads-to* process 1 at S4 $\vee$ ($X_1 \wedge$ process 2 at R2) (via S3)

The tag "via S3" at the right indicates that Y_2 holds by the *leads-to* via S rule with S instantiated by S3. Here are some other assertions that hold by the leads-to via S rule:

$Y_3 \equiv X_1 \wedge$ process 2 at R2 *leads-to* $\neg\, thinking_1 \wedge$ process 1 at S3 $\wedge$ $turn = 1 \wedge$ process 2 at R3 (via R2)

$Y_4 \equiv \neg\, thinking_1 \wedge$ process 1 at S3 $\wedge$ $turn = 1 \wedge$ process 2 at R3 *leads-to* process 1 at S4 (via S3)

$Y_5 \equiv X_1 \wedge$ process 2 at R3 *leads-to* $X_1 \wedge$ process 2 at R4 (via R3)

$Y_6 \equiv X_1 \wedge$ process 2 at R4 *leads-to* $X_1 \wedge$ process 2 at R1 (via R4)

The desired result Y_1 follows by applying closure rules to Y_2 through Y_6:

$Y_7 \equiv X_1 \wedge$ process 2 at R2 *leads-to* process 1 at S4 (from Y_3 and Y_4)

$Y_8 \equiv X_1 \wedge$ process 2 at R1 *leads-to* process 1 at S4 (from Y_2 and Y_7)

$Y_9 \equiv X_1 \wedge$ process 2 at R4 *leads-to* process 1 at S4 (from Y_6 and Y_8)

$Y_{10} \equiv X_1 \wedge$ process 2 at R3 *leads-to* process 1 at S4 (from Y_5 and Y_9)

Finally, Y_1 follows from Y_7, Y_8, Y_9, and Y_{10}, because when X_1 holds, process 2 has to be at one of the statements R1, R2, R3, or R4. This completes our assertional proof of progress. As in the case of the safety proof, we point out that each step can be checked by a nonexpert.

2.3 State Transition Systems

In our proofs for the mutual exclusion algorithm, we have implicitly assumed that each statement is executed **atomically**. That is, once a process starts executing a statement, another process cannot influence the statement's execution or observe intermediate points of the execution. This means that if two statements, say Si and Rj, are executed concurrently, then the net effect is either that of Si followed by Rj, or Rj followed by Si. (This assumption was made in both the operational and assertional proofs.)

More generally, consider a system of processes concurrently executing on some architecture (hardware or software). To analyze such a system, we need a formal model of the system. Any formal model makes assumptions about the real world that ultimately have to be accepted on faith. Almost every model used for correctness analysis assumes that the execution of a concurrent system can be viewed in terms of events that can be considered atomic.[6] Some events represent activities internal to a process (e.g., read or write of a local variable), while the remaining events represent communications between processes (e.g., a message transfer, a read, or write of a shared variable).

The granularity of an event refers to the extent of the system that is affected by the event; for example, an event that reads an integer is more coarse-grained than one that reads a single bit. The **level of atomicity**, i.e., the granularity of events that we can consider atomic, depends on the architecture upon which the processes execute and the properties being analyzed. A well-known folk theorem is that any sequence of operations can be considered atomic if it contains only a single access to a single shared variable. Larger units can be treated as atomic by using synchronization constructs, such as awaits, semaphores, conditional regions, etc. In a message-passing system, it is common to treat each message send or reception as atomic. (See Lamport [1990] for a discussion of the folk theorem and extensions to it.)

Because of the atomicity assumptions, we can view a concurrent program as a **state transition system**. Simply associate with each process a control variable that indicates the atomic statement to be executed next by the process. Then the state of the program is defined by the values of the data variables and control variables. In any state, an atomic statement can be executed if and only if it is pointed to by a control variable and is enabled. Executing the statement results in a new state. Thus each statement execution corresponds to a state transition, and each execution of the concurrent program corresponds to a sequence of state transitions.

The distinction between data variables and control variables is natural in a concurrent program. Data variables are updated in assignment statements, whereas control variables are updated according to the control flow of the program statements. However, as far as understanding and analyzing the program is concerned, there is no fundamental difference between data and control variables. But dealing with control flow of program statements is notationally more cumbersome than dealing with assignment statements. So it is natural to consider a system model where control variables are treated just like data variables, i.e., updated in assignment statements. This is the approach taken in many recent works, for example, Back and Kurki-Suonio [1988], Lamport [1991], Chandry and Misra [1986; 1988], Manna and Pnueli [1984; 1992], Lynch and Tuttle [1987].[7]

[6] For a discussion, see Sections 2.1 and 2.2 in Manna and Pnueli [1992]

[7] An example of an approach that does not treat control variables just like data variables is Owicki and Lamport [1982]

This means that the state transition system can be defined by a set of **state variables** (corresponding to the data and control variables) and a set of **events** (corresponding to the atomic statements). Each event has an **enabling condition**, which is a predicate on the state variables, and an **action**, which updates the state variables. Thus, a concurrent system is described by a state transition system and fairness assumptions on the events. Typically, the state transition system is nondeterministic in that more than one event can be enabled in a system state.

This is illustrated with the mutual-exclusion algorithm. For $i = 1, 2$, let $control_i$ denote the control variable for process i. Let e_{Sk} denote the event corresponding to Sk, and let e_{Rk} denote the event corresponding to Rk. The concurrent system is specified as follows:

State variables and initial condition:

```
    thinking_1, thinking_2: boolean.
      Initially true.
    turn: {1,2}.
    control_1, control_2: {1,2,3,4}. Initially 1.
```

Events:

```
  e_S1
    enabled ≡ control_1 = 1
    action ≡ thinking_1 ← false;
             control_1 ← 2

  e_S2
    enabled ≡ control_1 = 2
    action ≡ turn ← 2; control_1 ← 3

  e_S3
    enabled ≡ control_1 = 3 ∧
             (thinking_2 ∨ turn = 1)
    action ≡ control_1 ← 4

  e_S4
    enabled ≡ control_1 = 4
    action ≡ thinking_1 ← true;
             control_1 ← 1

  e_R1
    enabled ≡ control_2 = 1
    action ≡ thinking_2 ← false;
             control_2 ← 2

  e_R2
    enabled ≡ control_2 = 2
    action ≡ turn ← 1; control_2 ← 3

  e_R3
    enabled ≡ control_2 = 3 ∧
             (thinking_1 ∨ turn = 2)
    action ≡ control_2 ← 4

  e_R4
    enabled ≡ control_2 = 4
    action ≡ thinking_2 ← true;
             control_2 ← 1
```

Fairness requirements:

Events e_{S2}, e_{S3}, e_{S4}, e_{R2}, e_{R3}, and e_{R4} have weak fairness

Although we have listed the events in the same order as in the program, they can be listed in any order. The desired safety property is expressed by *Invariant*$(\neg(control_1 = 4 \wedge control_2 = 4))$ and the desired progress property by $control_1 = 2$ *leads-to* $control_1 = 4$ and $control_2 = 2$ *leads-to* $control_2 = 4$.

The above example illustrates how to encode read, write, await, and repeat-forever statements in state transition notation. Other kinds of statements can be encoded similarly. For example,

```
S1: if B then S2: ⟨statement⟩
          else S3: ⟨statement⟩
```

becomes the event

```
e_S1
  enabled ≡ control = S1
  action ≡ if B then control ← S2
                else control ← S3
```

if we assume that B is checked atomically. If instead $B \equiv C \wedge D$, where C and D are atomically evaluated in order, then we insert a control point, say S1a, between C and D, and model S1 by two events: one event has enabling condition $control =$ S1, and its action sets *control* to S1a or S3 depending on the value of C; the other event has enabling condition $control =$ S1a, and its action sets *control* to S2 or S3 depending on the value of D.

For another example, consider "S1: $P(sem)$; S2: ⟨statement⟩," where *sem* is a semaphore. S1 can be modeled by an event with enabling condition $control =$

S1 $\wedge$ *sem* > 0 and action *sem* $\leftarrow$ *sem* $-$ 1; *control* $\leftarrow$ S2. Similarly, "S1: $V(sem)$; S2: ⟨statement⟩" can be modeled by an event with enabling condition *control* $=$ S1 and action *sem* $\leftarrow$ *sem* $+ 1$; *control* $\leftarrow$ S2.

As mentioned in the Introduction, properties such as partial correctness, invariants, termination, deadlock freedom, livelock freedom, etc., can be modeled by safety and progress assertions. Invariants and livelock freedom (or starvation freedom) were illustrated in the mutual exclusion example. Deadlock freedom for a concurrent program means that the program can never reach a state where all processes are blocked. It is modeled by *Invariant*(E), where E denotes the disjunction of the enabling conditions of all events. Note that deadlock freedom is a safety property. For a concurrent program that is supposed to terminate, partial correctness means that if the program terminates then some desired predicate P holds. Let $control_i$ denote the control variable for the ith process in the program. Let $start_i$ denote its initial value, and end_i denote its value at termination. Partial correctness is specified by

$$Invariant([\forall i\colon control_i = end_i] \Rightarrow P),$$

and termination of the program is specified by

$$[\forall i\colon control_i = start_i] \ \textit{leads-to}\ [\forall i\colon control_i = end_i].$$

3. SYSTEM MODEL AND ASSERTION LANGUAGE

In Section 2, we informally introduced a system model, namely, state transition systems and fairness requirements, an assertion language, namely, invariant and leads-to assertions, and some proof rules. In this section, we describe the system model and assertion language precisely, introduce a few extensions, and show how our formalism fits in with those of other authors. Proof rules are dealt with in Section 4.

Currently, our notion of fairness applies to an individual event. We will extend this to a set of events, because this is often more appropriate in many situations. We will allow assertions that consist of invariant assertions and leads-to assertions joined by logical connectives. We will allow auxiliary state variables, for recording history information of system execution. With auxiliary variables, any safety and progress property, including fairness requirements, can be expressed in terms of invariant and leads-to assertions.

3.1 System Model

A (concurrent) system A is defined by

- a state transition system defined by
  - —$Variables_A$, a set of state variables and their domains.
  - —$Initial_A$, an initial condition on $Variables_A$.
  - —$Events_A$, a set of events.
  - —For each event $e \in Events_A$:
    $enabled_A(e)$, an enabling condition (predicate in $Variables_A$); and
    $action_A(e)$, an action (sequential program that updates $Variables_A$);
- a finite set of fairness requirements
  - —each fairness requirement is a subset of $Events_A$ tagged with "weak fairness" or "strong fairness."

$Variables_A$ defines the set of system states; specifically, each value assignment to the variables denotes a system state. $Initial_A$ defines a subset of system states, referred to as the initial states. We assume that the set of initial states is nonempty. For each event e, the enabling condition and action define a set of **state transitions**, specifically, $\{(s, t)$: s, t are system states; s satisfies $enabled_A(e)$; and t is the result of executing $action_A(e)$ in $s\}$. We assume that $action_A(e)$ always terminates when executed in any state satisfying $enabled_A(e)$ and that its execution is atomic.

A **behavior** is a sequence of the form $\langle s_0, e_0, s_1, e_1, \ldots \rangle$ where the s_i's are states, the e_i's are events, s_0 is an initial state, and every (s_i, s_{i+1}) is a transition of e_i. A behavior can be *infinite* or *finite*. By definition, a finite behavior ends in a state. Note that for any behavior σ, every finite prefix of σ ending in a state is also a behavior. Let *Behaviors*(A) denote the set of behaviors of A. *Behaviors*(A) is sufficient for defining safety properties of A but not its progress properties (because it includes behaviors where A's fairness requirements are not satisfied).

We next define the behaviors of A that satisfy the fairness requirements. Let E be a subset of $Events_A$. The enabling condition of E, denoted $enabled_A(E)$, is defined by $[\exists e \in E: enabled_A(e)]$. Thus, E is enabled (disabled) in a state iff some (no) event of E is enabled in the state. Let $\sigma = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ be an infinite behavior. We say "E is enabled (disabled) infinitely often" in σ if E is enabled (disabled) at an infinite number of s_i's. We say "E occurs infinitely often" in σ if an infinite number of e_i's belong to E.

A behavior σ of A satisfies **weak fairness for** E iff (1) σ is finite and E is disabled in the last state of σ, or (2) σ is infinite and either E occurs infinitely often or is disabled infinitely often in σ [Lynch and Tuttle 1987]. Informally, this means that if E is enabled continuously, then it eventually occurs.

A behavior σ of A satisfies **strong fairness for** E iff (1) σ is finite and E is disabled in the last state of σ, or (2) σ is infinite and if E is enabled infinitely often in σ, then it occurs infinitely often in σ. Informally, this means that if E is enabled infinitely often, then it eventually occurs.

A behavior σ of A is **allowed** iff σ satisfies every fairness requirement of A. Let *AllowedBehaviors*(A) denote the set of allowed behaviors of A. *AllowedBehaviors*(A) is sufficient for defining the progress properties of A, as well as the safety properties of A.

Note that a prefix of an allowed behavior is not necessarily an allowed behavior. Intuitively, a finite behavior σ is allowed iff the fairness requirements of A do not require A to extend σ in the future. This does not mean that A must not extend σ. It just means that A is not *obliged* to extend σ. If A is not subject to fairness requirements, then every behavior of the system is an allowed behavior.

It may appear that fairness requirements are a complicated way of forcing an event to occur. However, a little thought shows that this is not so. For example, we cannot insist that an event occur as soon as it is enabled, because this means that two or more events cannot be enabled in the same state; i.e., it eliminates nondeterminism, which is fundamental to our modeling of concurrency. Another approach is to insist that events occur within some T seconds of being continuously enabled. Although this approach allows nondeterminism, its real-time constraint makes it harder to implement, and hence undesirable unless we are interested explicitly in real-time properties.

We allow events to have **parameters**. This is a convenient way of defining a collection of events. For example, consider an event $E(i)$ with enabling condition $x = 0$ and action $x \leftarrow i$, where x is an integer state variable and where the parameter i ranges over $\{1, 2, \ldots, 50\}$. Event $E(i)$ actually specifies a collection of 50 events, $E(1), E(2), \ldots, E(50)$.

3.2 Assertion Language

Let A be a system that we want to analyze. Henceforth, we use the term **state formula** to refer to a predicate in the state variables of A. A state formula evaluates to *true* or *false* for each state of the system.[8] We say that a state satisfies a state formula to mean that the state formula evaluates to *true* at that state.

In Section 2, we considered assertions of the form *Invariant*(P) and P *leads-to* Q, where P and Q are state formulas.

[8]The precise meaning of evaluating a state formula P at a state s is as follows: for each state variable v that appears free in P, replace v by the value of the state variable in s, and then evaluate the resulting P.

We now define precisely what it means for an assertion to hold (or be satisfied) for system A.

We first define what it means for an assertion to satisfy a behavior of A. Let $\sigma = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ be a (finite or infinite) sequence of alternating states and events of A. Sequence σ satisfies *Invariant*(P) iff every state s_i in σ satisfies P. Sequence σ satisfies P *leads-to* Q iff for every s_i in σ that satisfies P there is an s_j in σ, $j \geq i$, that satisfies Q.

System A satisfies *Invariant*(P) iff every behavior of A satisfies *Invariant*(P). System A satisfies P *leads-to* Q iff every *allowed* behavior of A satisfies P *leads-to* Q.

Actually, for system A to satisfy *Invariant*(P), it is sufficient if *Invariant*(P) holds for every finite behavior. And because every finite behavior is a prefix of an allowed behavior, it is sufficient if *Invariant*(P) holds for every allowed behavior. So, we can say that an (invariant or leads-to) assertion holds for A iff it holds for every allowed behavior of A.

We now extend the class of assertions in two ways. First, we allow assertions that are made up of invariant assertions or leads-to assertions joined by logical connectives (for example, (*Invariant*(T) $\wedge$ (P *leads-to* Q)) $\Rightarrow$ (R *leads-to* S)). Such an assertion L is satisfied by σ iff L evaluates to *true* after each invariant or leads-to assertion X in L is replaced by $X(\sigma)$, where $X(\sigma)$ is *true* if σ satisfies X and *false* if σ does not satisfy X.[9] As before, system A satisfies L iff every allowed behavior of A satisfies L.

Our second extension is to allow assertions to contain **parameters**, which are variables distinct from state variables (and thus not affected by events).[10] Parameters are convenient for defining classes of assertions. For example, $x = n$ *leads-to* $y = n + 1$, where n is an integer parameter, defines a collection of leads-to assertions. When evaluating an assertion, every parameter is universally quantified. Thus, $x = n$ *leads-to* $y = n + 1$ holds iff [$\forall n$: $x = n$ *leads-to* $y = n + 1$] holds.

Throughout this tutorial, unless otherwise indicated, we assume the following precedence of operator binding power: arithmetic and data structure operators, such as $+$, $=$, *prefix*, *subset*, bind stronger than logical operators; the logical operators $\neg$, $\wedge$, $\vee$, and $\Rightarrow$ are in decreasing order of binding power; followed by the *leads-to* operator. Here are some examples of invariant and progress assertions, together with an informal English interpretation:

- *Invariant*$(x > y)$: x is always greater than y;
- $x = 0$ *leads-to* $y = 1$: if x equals 0, then eventually y equals 1;
- $x = n$ *leads-to* $x = n + 1$: x keeps increasing;
- $x = n \wedge y \neq 0$ *leads-to* $x \geq n + 1 \vee y = 0$: x grows without bound unless y becomes 0;
- ($x \geq n$ *leads-to* $x \geq n + 1$) $\Rightarrow$ ($y \geq m$ *leads-to* $y \geq m + 1$): y grows without bound if x grows without bound.

Note that the third assertion is satisfied only by infinite behaviors, whereas each of the other assertions is satisfied by some finite behaviors and some infinite behaviors.

We use invariant assertions to express safety properties and use assertions built from leads-to and invariant assertions to express progress properties. We now make precise the terms *safety property* and *progress property*. Assertional reasoning is concerned with properties P such that for every sequence σ of alternating states and events, P is either true or false. P is a *safety property* if for any σ, if P holds for σ then it holds for any prefix of σ. This means that if a safety property does not hold for some σ, then there is a point in σ where it first stops holding. P is a *pure progress property* if

[9] Thus, σ satisfies (*Invariant*(T) $\wedge$ (P *leads-to* Q)) $\Rightarrow$ (R *leads-to* S) iff σ satisfies R *leads-to* S, or σ does not satisfy *Invariant*(T) or P *leads-to* Q.)

[10] Parameters are also referred to as *rigid variables* in the literature [Lamport 1991; Manna and Pnueli 1992]

any finite σ can be extended to a sequence that does satisfy P. Alpern and Schneider [1985] showed that every property (that evaluates to true or false for every sequence σ) can be expressed as the conjunction of a safety property and a pure progress property.

3.3 Auxiliary Variables

Our assertion language uses only two operators on sequences, namely, *Invariant* and *leads-to*. Because of this, we may not be able to express all safety and progress properties of interest for a given system A. One way to overcome this is to add more operators on sequences, as in Manna and Pnueli [1984; 1992] (discussed below).

Another way, which is the one taken in this tutorial, is to augment system A with auxiliary variables [Owicki and Gries 1976]. Auxiliary variables are state variables that record some history of system execution, but they do not affect the system execution and hence do not have to be implemented. Auxiliary variables are also known as history variables [Abadi and Lamport 1988].

Specifically, we can declare a subset *Vars* of the state variables of A to be **auxiliary variables** iff the variables in *Vars* (1) do not appear in event-enabling conditions, and (2) do not affect the update of any state variable not in *Vars*. Because we specify actions by programs, we can express condition (2) using the Owicki and Gries [1976] syntactic criteria: that is, variables from *Vars* can appear only in assignment statements; and if a variable from *Vars* appears in the right-hand side of an assignment statement, then the left-hand side must be also a variable from *Vars*.[11]

There is a difference between our use of auxiliary variables and that of Owicki and Gries [1976]. We use them in stating and proving desired properties, whereas Owicki and Gries used them only in proving; their desired properties were stated without recourse to auxiliary variables.

With auxiliary variables, we can express fairness requirements as progress assertions. For an event set E subject to fairness, let $count(E)$ be an auxiliary variable indicating the number of times E has occurred since the beginning of system execution; that is, include $count(E) \leftarrow count(E) + 1$ in the action of every event $e \in E$. The following assertions are equivalent to weak and strong fairness, respectively:

- $enabled(E) \wedge count(E) = k$ *leads-to* $\neg\, enabled(E) \vee count(E) = k + 1$ (weak fairness)
- $(\neg\, enabled(E)$ *leads-to* $enabled(E)) \Rightarrow (count(E) = k$ *leads-to* $count(E) = k + 1)$ (strong fairness)

3.4 Relationship to Other Formalisms

As mentioned earlier, most of the recent approaches to assertional reasoning of concurrent systems use a state transition model, for example, Abadi and Lamport [1988; 1990], Lamport [1989; 1991], Manna and Pnueli [1984; 1992], Chandy and Misra [1986; 1988], Back and Kurki-Suonio [1988], Lynch and Tuttle [1987], Lam and Shankar [1990]. All these approaches use a set of state variables to define the system state, but they differ in how they define the state transitions of an event, and in the kinds of fairness requirements.

In this tutorial, an event is defined by an enabling condition and an action. For example, a state transition system with three integer state variables, x, y, z can have an event e with enabling condition $x = 2$ and action $y \leftarrow y + 1;\ x \leftarrow 3$.

In UNITY [Chandry and Misra 1986; 1988], the abstraction from program is taken one level further. Each event, referred to as a UNITY statement, is considered to be always enabled, and the action has the form of a multiple-assignment statement with an optional guard. Thus, the above event e would become

[11] Implicit in this syntactic criteria is the assumption that the value of a variable can be changed only by assignment

the UNITY statement "$(x, y) \leftarrow (3, y + 1)$ if $x = 2$." Although the statement is always enabled, its occurrence changes the system state only if the guard is true.

In both this tutorial and in UNITY, the assignment statement is used to change the value of a state variable. Consequently, Hoare logic or weakest precondition logic (and in particular the assignment axiom) is needed to prove properties of individual events. For example, to prove that the above event e preserves a state formula P,[12] we would have to prove the Hoare-triple $\{P \wedge x = 2\}\ y \leftarrow y + 1;\ x \leftarrow 3\{P\}$. (These logics are discussed in Section 4.)

An alternative approach is to do away with assignments (e.g., Lamport [1983a; 1991], Shankar and Lam [1987], and Lam and Shankar [1990]). Define an event by a predicate in primed and unprimed versions of the state variables, where the primed version of a state variable refers to its new value. In this approach, the above event e is defined by the predicate $x = 2 \wedge x' = 3 \wedge y' = y + 1 \wedge z' = z$. Any state pair (s, t) that satisfies the predicate is a transition of the event. Because there is no assignment statement, reasoning about the transitions does not need Hoare logic; standard logic is sufficient. For example, to prove that event e preserves a state formula P, one has to prove $x = 2 \wedge x' = 3 \wedge y' = y + 1 \wedge z' = z \wedge P \Rightarrow P'$, where P' is P with every free occurrence of x, y, and z replaced by x', y', and z'. A variation of this method is to use pre- and post-conditions to define the transitions of an event [Lynch and Tuttle 1987]. Event e can be specified by precondition $x = 2 \wedge y = n$ and postcondition $x = 3 \wedge y = n + 1$. Note that n is not a state variable but a parameter used for referring to the value of y before the event occurrence (thus a precondition is not the same as our enabling condition).

The notions of weak fairness and strong fairness were introduced in Lehmann et al. [1981]. A detailed treatment of them (and other kinds of fairness) can be found in Francez [1986] and Manna and Pnueli [1992]. The definitions of behaviors and weak fairness used in this tutorial are taken from Lynch and Tuttle [1987].[13] The definition of strong fairness is taken from Lam and Shankar [1990]. UNITY uses the same notion of behaviors. The standard fairness notion in UNITY is that every UNITY statement is executed infinitely often. This corresponds to weak fairness for every event, if we treat a UNITY statement as an event with the UNITY guard becoming the event's enabling condition.

Manna and Pnueli [1992] and Lamport [1991] have a technically different definition of behaviors. They prefer to deal only with infinite behaviors. So they augment the system being studied with a hypothetical "idling event" that is always enabled and whose occurrence does not change state. Thus a finite behavior is extended to an infinite one by appending an infinite sequence of idling transitions. Also, a finite sequence of idling transitions can be inserted between any two transitions. Lamport refers to this as "stuttering." The fairness requirements in Lamport [1991] are, as in this tutorial, defined in terms of event sets subject to weak or strong fairness. In Manna and Pnueli [1992], individual events are subject to weak or strong fairness.[14]

Manna and Pnueli's [1984; 1992] temporal logic is a logic for reasoning about sequences of states. It has a variety of *temporal* operators, including $\Box$ (*henceforth*), $\Diamond$ (*eventually*), $\bigcirc$ (*next*), and U (*until*), for constructing assertions (temporal formulas) from state formulas. Let P and Q be state formulas, and let $\sigma =$

[12] That is, to prove that the execution of e in any state where P holds results in a state satisfying P

[13] Lynch and Tuttle [1989] use the terms "executions" for our behaviors and "actions" for our events. They require that different event sets subject to fairness must be mutually exclusive, and they do not consider strong fairness

[14] Lamport [1991] uses the term "action" for our event. Manna and Pnueli [1992] use the term "transition" for our event, "transition relation" for our set of transitions, "justice" for weak fairness, "compassion" for strong fairness, and "process-justice" for the fairness on event sets.

$\langle s_0, s_1, \dots \rangle$ be an infinite sequence of states. $\Box P$ is satisfied by σ iff every s_i satisfies P. $\Diamond P$ is satisfied by σ iff there is some s_i that satisfies P. $\bigcirc P$ is satisfied by σ iff s_1 satisfies P. $P\,U\,Q$ is satisfied by σ iff the following holds: if s_0 satisfies P, there is some s_i that satisfies Q and every s_j, $0 \le j < i$, satisfies P. In the above expressions, P can be replaced by a temporal formula, resulting in a rich assertion language. For example, $P \Rightarrow \Diamond Q$ is satisfied by σ iff the following holds: if s_0 satisfies P, then there is some s_i that satisfies Q. $\Box(P \Rightarrow \Diamond Q)$ is satisfied by σ iff for every s_i that satisfies P, there is some s_j, $j \ge i$ that satisfies Q. $\Box(P \Rightarrow \bigcirc Q)$ is satsified by σ iff for every s_i that satisfies P, s_{i+1} satisfies Q. The above temporal operators can be thought of as "future" operators, because they examine the states to the right of s_0. For each of these operators, Manna and Pnueli [1992] also define a "past" version that examines states to the left. A thorough treatment of the relationships between different temporal operators (e.g., $\neg \Box \neg P$ is equivalent to $\Diamond P$) and proof rules is given in Manna and Pnueli [1992].

In this tutorial, we make use of a fragment of Manna and Pnueli's temporal logic. Specifically, our *Invariant*(P) is $\Box P$, and our P *leads-to* Q is $\Box(P \Rightarrow \Diamond Q)$. Using only these two temporal operators may require the use of auxiliary variables to state certain properties. For example, the property "x never decreases" is specified in Manna and Pnueli's logic by $\Box(x = n \Rightarrow \bigcirc x \ge n)$, and in our logic by *Invariant*$(x_{old} \le x)$, where x_{old} is an auxiliary variable that is assigned the value of x at the start of every event action (which may affect x). Of course, if this is not convenient, we can always include additional temporal operators from Manna and Pnueli's logic (e.g., see Alaettinoglu and Shankar [1992]).

Lamport [1991] defines a Temporal Logic of Actions, referred to as TLA, in which he can express event specifications, fairness requirements, and a fragment of Manna and Pnueli's temporal logic (consisting of $\Box$ and $\Diamond$) under one unified logic.

4. PROVING SAFETY AND PROGRESS ASSERTIONS

Recall that our safety assertions are invariant assertions, and our progress assertions are built from lead-to assertions and invariant assertions. In Section 2, we informally described proof rules for proving invariant and leads-to assertions. In this section, we describe a more complete set of proof rules and give a rigorous justification for them. Because the system model and assertion language of this tutorial are very similar to those in Lamport [1991], Manna and Pnueli [1984, 1992], and Chandry and Misra [1986; 1988], our proof rules are also similar, and in many cases identical, to the rules found in those references.

Proof rules contain statements such as "e preserves P," where e is an event and P is a state formula. To reason rigorously about such statements, we resort to Hoare logic [Hoare 1969; Apt 1981], a well-known formalism for sequential programs. We also describe Dijkstra's weakest precondition logic [Dijkstra 1976], another well-known formalism for sequential programs. Although weakest precondition logic is not needed for stating or checking proofs, it is very helpful in inventing proofs.

In Section 4.1, we describe Hoare logic for reasoning about actions. In Section 4.2, we describe Dijkstra's weakest precondition logic for reasoning about actions. In Section 4.3, we extend this notation to reason about individual events. In Section 4.4, we describe rules for proving invariant assertions, and in Section 4.5 we describe a heuristic for applying the rules. In Section 4.6, we describe rules for proving leads-to assertions. Throughout this section, P, Q, and R denote state formulas.

4.1 Reasoning about Actions with Hoare-Triples

A **Hoare-triple** has the form $\{P\}S\{Q\}$, where P and Q are state formulas and where S is an action. $\{P\}S\{Q\}$ means that for every state s satisfying P, the execution of S starting from s terminates in a state that satisfies Q.

A Hoare-triple may be valid or invalid. Here are some valid Hoare-triples:

- $\{y = 3\}$ if $x \neq y$ then $x \leftarrow y + 1$ $\{x = 3 \vee x = 4\}$
- $\{true\}$ if $x \neq y$ then $x \leftarrow y + 1$ $\{x = y + 1 \vee x = y\}$
- $\{x = 0 \wedge y = n\}$ $x \leftarrow x + y$ $\{x = n\}$
- $\{x = n\}$ for $i = 0$ to 10: $x \leftarrow x + i$ $\{x = n + 55\}$
- $\{x = 0 \wedge y = 1\}$ while $x > 0$ do $x \leftarrow 2x$ $\{y = 1\}$

And here are some invalid Hoare-triples:

- $\{x = 3\}$ $x \leftarrow y + 1$ $\{x = 4\}$
- $\{x = 1 \wedge y = 1\}$ while $x > 0$ do $x \leftarrow 2x$ $\{y = 1\}$
- $\{P\}S\{false\}$, for any S and P.

We have given examples of Hoare-triples, but we have not given proof rules for them. The actions that we will encounter in this tutorial are simple. Consequently, we will be able to generate valid Hoare-triples by inspection, as in the above examples. At the same time, there are simple proof rules when the actions do not involve loops, and we now explain them. We can use them instead of, or to supplement, our "answer by inspection."

For any state formula P, let $P[x/t]$ denote the state formula obtained by replacing every free occurrence of x in P by t.[15] The proof rules for Hoare-triples are as follows (for rules of additional constructs, see Hoare [1969], Apt [1981], and Gries [1981]):

- $\{P\}x \leftarrow e\{Q\}$ holds if $P \Rightarrow Q[x/e]$ holds.
- $\{P\}$ if B then $S\{Q\}$ holds if $\{P \wedge B\}S\{Q\}$ and $P \wedge \neg B \Rightarrow Q$ hold.
- $\{P\}$ if B then S_1 else $S_2\{Q\}$ holds if $\{P \wedge B\}S_1\{Q\}$ and $\{P \wedge \neg B\}S_2\{Q\}$ hold.
- $\{P\}$ while B do $S\{Q\}$ holds if $\{P \wedge B\}S\{P\}$ and $P \wedge \neg B \Rightarrow Q$ hold.
- $\{P\}S_1;\ S_2\{Q\}$ holds if for some state formula R, $\{P\}S_1\{R\}$ and $\{R\}S_2\{Q\}$ hold.

4.2 Reasoning about Actions with Weakest Preconditions

Dijkstra's weakest precondition terminology offers another way to reason about actions. This terminology is more involved than Hoare-triples, and it is not needed for stating proofs. But it is very helpful for inventing proofs. Throughout we use "wrt" as an abbreviation for "with respect to."

P is a **sufficient precondition of** Q **wrt** S means that $\{P\}S\{Q\}$ holds. P is a **weakest precondition of** Q **wrt** S means that (a) P is a sufficient precondition of Q wrt S and (b) $R \Rightarrow P$ holds for any other sufficient precondition R of Q wrt S. That is, a weakest condition P specifies the *largest* set of states where the execution of S terminates with Q holding.

One way to obtain weakest preconditions is by inspection. Another way is to use Dijkstra's predicate transformer $wp(S, Q)$, which for a program S and state formula Q returns a weakest precondition of Q wrt S. For the constructs we consider here, $wp(S, Q)$ is as follows (for additional constructs, see Dijkstra [1976]):

- $wp(x \leftarrow e, Q) \equiv Q[x/e]$
- $wp(\text{if } B \text{ then } S, Q) \equiv (B \Rightarrow wp(S, Q)) \wedge (\neg B \Rightarrow Q)$
- $wp(\text{if } B \text{ then } S_1 \text{ else } S_2, Q) \equiv (B \Rightarrow wp(S_1, Q)) \wedge (\neg B \Rightarrow wp(S_2, Q))$
- $wp(S_1; S_2, Q) \equiv wp(S_1, wp(S_2, Q))$

[15] For example, if $P \equiv x = 2 \vee y = 3$, then $P[x/5] \equiv 5 = 2 \vee y = 3$ (which is equivalent to $y = 3$), $P[x/x+1] \equiv x + 1 = 2 \vee y = 3$ (which is equivalent to $x = 1 \vee y = 3$), $P[x/2] \equiv 2 = 2 \vee y = 3$ (which is equivalent to $true$), and $P[x/z, y/z] \equiv z = 2 \vee z = 3$

Every variable in P is either free or bound, i.e., within the scope of a quantifier. If the expression t contains an identifier that happens to be also used to identify a bound variable of P, then suitable renaming is needed to avoid name clashes. For example, consider $P \equiv x = 2 \wedge [\exists x\colon x = y]$. P has three variables the x inside the scope of the existential quantification, which is bound; the x outside the scope, which is free; and y, which is free To obtain $P[y/x]$, it is necessary to rename the bound x to something else, e.g., n, so that there is no name clash, thereby obtaining $x = 2 \wedge [\exists n\colon n = x]$.

Thus, $P \Leftrightarrow wp(S, Q)$ holds means that P is a weakest precondition of Q wrt S, and $P \Rightarrow wp(S, Q)$ holds means that P is a sufficient precondition of Q wrt S.

4.3 Reasoning about Events

We extend the Hoare-triple notation to an event e by defining $\{P\}e\{Q\}$ to mean $\{P \wedge enabled(e)\}action(e)\{Q\}$. That is, for every state s satisfying P, either e is not enabled at s, or the execution of $action(e)$ starting from s terminates in a state that satisfies Q. For example, given an event e with enabling condition $x \neq y$ and action $x \leftarrow y + 1$, $\{x \neq y\}e\{x = y + 1\}$, $\{x = y\}e\{x \neq y + 1\}$, and $\{x \neq y \Rightarrow y = n - 1\}e\{x = n\}$ hold, and $\{x \neq y\}e\{x \neq y + 1\}$ does not hold.

We extend the weakest precondition notation to events. P is a **sufficient precondition of** Q **wrt** e means that $\{P\}e\{Q\}$ holds. P is a **weakest precondition of** Q **wrt** e means that P is a sufficient precondition of Q wrt e, and $R \Rightarrow P$ holds for any other sufficient precondition R of Q wrt e. We define P to be a **necessary precondition of** Q **wrt** e if for every state s satisfying $\neg P$, e is enabled and its action results in a state satisfying $\neg Q$. If P is both a sufficient precondition of Q wrt e and a necessary precondition of Q wrt e, then P is a weakest precondition of Q wrt e.

The wp predicate transformer for event e is defined by $wp(e, Q) \equiv enabled(e) \Rightarrow wp(action(e), Q)$.

4.4 Proof Rules for Invariance

There are two kinds of rules for proving invariants. The first kind is for inferring invariant assertions from the system specification:

Invariance Rule. $Invariant(P)$ is satisfied by system A if the following hold:

(i) $Initial_A \Rightarrow P$
(ii) for every event e of A: $\{P\}e\{P\}$

Proof of Soundness. Let $\sigma = \langle s_0, e_0, s_1, e_1, \dots \rangle$ be a finite behavior of system A. We have to show that conditions (i) and (ii) imply that every s_i in σ satisfies P. We prove this by induction on the length of σ. Condition (i) implies that s_0 satisfies P. Assume that for some n, states $s_0, \dots, s_n$ in σ satisfy P. Assume that s_n is not the last state of σ (otherwise, we are done). Thus, e_n, s_{n+1} follows s_n in σ. It suffices to prove that s_{n+1} satisfies P. Because σ is a behavior, s_n satisfies $enabled_A(e_n)$. From the induction hypothesis, we have that s_n satisfies P. Therefore, condition (ii) implies that s_{n+1} satisfies P. □

Suppose we know that system A satisfies $Invariant(Q)$ for some state formula Q. How can we exploit this information in proving $Invariant(P)$? Basically, we can relax condition (ii) in rule 1 to $\{P \wedge Q\}e\{Q \Rightarrow P\}$, because we do not have to consider state transitions (s, t) where s or t does not satisfy Q. That is, the above invariance rule can be generalized to the following:

$Invariant(P)$ is satisfied by system A if the following hold for some state formula Q:

(i) $Initial_A \Rightarrow P$
(ii) for every event e of A: $\{P \wedge Q\}e\{Q \Rightarrow P\}$
(iii) $Invariant(Q)$ is satisfied by system A

Note that the general version reduces to the simple version if $Q \equiv true$. When we apply these rules in our examples, we shall say *$Invariant(P)$ holds by the invariance rule assuming $Invariant(Q)$* to mean that P and Q satisfy the conditions of the general invariance rule. We shall say *$Invariant(P)$ holds by the invariance rule* to mean that P satisfies the conditions of the simple invariance rule.

The second kind of proof rule is for inferring invariant assertions from other invariant assertions. Here are two examples:

- $Invariant(P)$ holds if for some state formula Q, $Q \Rightarrow P$ and $Invariant(Q)$ hold.

- *Invariant*(P) holds if for some state formulas Q and R, $Q \wedge R \Rightarrow P$, *Invariant*(Q), and *Invariant*(R) hold.

Actually, the first rule is a special case of the second rule with $R \equiv$ *true*. Because these rules are so obvious, we do not give them any special name. When we use the first (or second) rule, we simply say *Invariant*(P) holds because of *Invariant*(Q) (or *Invariant*(Q) and *Invariant*(R)).

Each rule above defines some sufficient conditions for invariance. Suppose we have a system and a state formula P, and we have to prove that the system satisfies *Invariant*(P). Where do we start? If we do not already know some invariant for the system, the first step is to see whether P satisfies the invariance rule. If we are lucky, it may. But in general, $\{P\}e\{P\}$ will not hold for some event e. This is natural because P is a desired property, and for any nontrivial desired property, the system has to maintain additional properties to achieve P. We need to unearth these additional properties, in order to find a state formula Q that implies P and satisfies the invariance rule.

Accumulator Example

Consider the following concurrent program written using the *cobegin/coend* construct:

```
x: integer Initially $x = 0$
cobegin
        $x \leftarrow x + 2^0 \| x \leftarrow x + 2^1 \| x \leftarrow x + 2^2 \|$
        $\cdots \| x \leftarrow x + 2^{N-1}$
coend
```

The above program has N subprograms that are executed concurrently. Let us assume that each $x \leftarrow x + 2^i$ statement is atomic. We can model the program by the following state transition system, where i is a parameter that ranges over $\{0, \ldots, N-1\}$ and where $b(i)$ denotes the control for the ith subprogram:

State variables and initial condition:
x: integer. Initially $x = 0$.
$b(i)$: $\{0, 1\}$. Initially $b(i) = 0$

Events:
$e(i)$
  enabled $\equiv b(i) = 0$
  action $\equiv x \leftarrow x + 2^i$; $b(i) \leftarrow 1$

Fairness requirements:
$\{e(0), \ldots, e(N-1)\}$ has weak fairness.

Suppose we want to prove that $x \geq 2^i$ if the ith subprogram has terminated. This property is stated by *Invariant*(B_0), where

$$B_0 \equiv b(i) = 1 \Rightarrow x \geq 2^i.$$

B_0 happens to satisfy the invariance rule: It holds initially. $\{B_0\}e(i)\{B_0\}$ holds because $e(i)$ makes the consequent true (by adding 2^i to x). $\{B_0\}e(k)\{B_0\}$ holds for $k \neq i$ because $e(k)$ does not make the antecedent true or the consequent false.

Suppose we want to prove that $x = 2^N - 1$ holds at termination of the program. The program is terminated iff all N subprograms have terminated. Thus the property can be stated as *Invariant*(A_0), where

$$A_0 \equiv [\forall i \colon b(i) = 1] \Rightarrow x = 2^N - 1.$$

Our first approach is to use the invariance rule. However, $\{A_0\}e(i)\{A_0\}$ does not hold. For example, consider a state s that satisfies $b(0) = 0$, $b(i) = 1$ for $i \neq 0$, and $x \neq 2^N - 2$. State s satisfies A_0 (vacuously). But $e(0)$ is enabled at s, and its occurrence would result in a state that does not satisfy A_0. Clearly, we need to keep more information about the relationship between the $b(i)$'s and x, which is that at any instant, x has accumulated the 2^i contributions of the i's where $b(i) = 1$. This leads us to the following state formula:

$$A_1 \equiv x = \sum_{i \in J} 2^i \text{ where } J = \{i \colon b(i) = 1\}.$$

A_1 satisfies the invariance rule (make sure of this). A_1 implies A_0. Therefore A_0 is invariant.

4.5 Generating Invariant Requirements

Suppose we have a system and a state formula A_0 that is to be proved invariant for the system. The difficulty is in coming up with a state formula that satisfies the invariance rule and implies A_0 (e.g., the state formula A_1 in the accumulator example above). This requires invention and insight into why the system works.

Although there is no algorithm to generate assertions, there is a heuristic based on weakest preconditions that is often successful [Shankar and Lam 1987]. The heuristic generates, starting from A_0, a succession of state formulas, $A_1, A_2, \ldots, A_{K-1}$, such that the conjunction $A_0 \wedge \cdots \wedge A_{K-1}$ satisfies the invariance rule.

At any point, the heuristic maintains the following:

- A set of state formulas, $A_0, A_1, \ldots, A_{K-1}$, where A_0 is the state formula to be proved invariant. Each A_i is referred to as an **invariant requirement**. K indicates the number of invariant requirements currently defined. For each A_i, *Initial* $\Rightarrow A_i$ holds. (We want *Invariant*(A_i) to hold.)
- A **marking**, defined as follows. Each (A_j, e) pair, where j ranges over $\{0, \ldots, K-1\}$ and where e ranges over the system events, is either **marked** or **unmarked**. Associated with each marked (A_j, e) pair is a subset J of $\{A_0, \ldots, A_{K-1}\}$ such that (the conjunction of the state formulas in) J is a sufficient precondition of A_j wrt e; that is,

$$\left\{ \bigwedge_{A_j \in J} A_j \right\} e \{A_j\}$$

holds. We refer to J as the **justification** for marking (A_j, e).

Note that the marking indicates the extent to which $A_0 \wedge \cdots \wedge A_{K-1}$ satisfies the invariance rule. If all (A_j, e) pairs are marked, then $A_0 \wedge \cdots \wedge A_{K-1}$ satisfies the invariance rule. The justifications allow us to "undo" parts of the proof if needed (see below). (The justifications are also useful for checking the proof.)

At the start of a heuristic application, assuming *Initial* $\Rightarrow A_0$ holds, we have a single invariant requirement, namely, A_0, and nothing is marked. (If *Initial* $\Rightarrow A_0$ does not hold, then A_0 is not invariant.)

At each step of the heuristic, we choose an unmarked (A_j, e) entry and do the following:

```
generate a weakest precondition $P$ of $A_j$
  wrt $e$.
if $Initial \Rightarrow P$ does not hold then STOP
  "heuristic terminates unsuccessfully"
else begin
  if $A_1 \wedge \cdots \wedge A_{K-1} \Rightarrow P$ does not
    hold then begin
      $A_K \equiv P$; $K \leftarrow K + 1$ end;
      "add a new invariant
      requirement"
      mark $(A_j, e)$ with justifica-
      tion $A_K$
    end
  else mark $(A_j, e)$ with justifica-
      tion $J$
      where $J \subseteq \{A_1, \ldots, A_{K-1}\}$
      such that $\wedge_{A_j \in J} A_j \Rightarrow P$
      holds
      end
```

The heuristic terminates successfully if all (A_j, e) pairs are marked; in this case, $A_0 \wedge \cdots \wedge A_{K-1}$ satisfies the invariance rule. The heuristic terminates unsuccessfully if a precondition P is generated that does not satisfy *Initial*; in this case, we can conclude that A_0 is not invariant [Shankar and Lam 1987].

Typically, a brute-force application of the heuristic will not terminate (except in special cases such as finite state systems). The following should be kept in mind when applying the heuristic.

First, it is crucial to simplify the expression for P as much as possible in each iteration. Otherwise the A_i's grow unmanageably. In addition to the usual algebraic and predicate calculus transformations, it is possible to make use of an existing invariant requirement, say A_i, to simplify the expression for P. For example, if $P \equiv y = 0 \Rightarrow x \in \{0, 1\}$ and $A_i \equiv x \in \{1, 2\}$, then we can simplify P to

$y = 0 \Rightarrow x = 1$; note that A_i now has to be included in the justification for marking (A_j, e).

Second, the choice of the unmarked (A_j, e) pair in each iteration is often very important. It is often very convenient to generate a precondition with respect to a sequence of events, rather than just one event; for example, $wp(e_j, A_i)$ may be complicated while $wp(e_j, wp(e_j, \cdots wp(e_j, A_i) \cdots))$ is simple.

Third, if the expression for a weakest precondition P becomes unmanageable (and this depends on our ingenuity and patience [Dijkstra 1976]), then we can obtain either a sufficient precondition or a necessary precondition. If we obtain a necessary precondition P, then we cannot mark (A_j, e). However, this step is still useful because it gives us another invariant requirement. If we obtain a sufficient precondition P, then we can mark (A_j, e). However, after this if the heuristic terminates unsuccessfully, i.e., without all (A_j, e) pairs marked, we *cannot* conclude that A_0 is not invariant. This is because the sufficient precondition P that was introduced as an invariant requirement may not be invariant; in which case, the heuristic failed because it attempted to prove that P is invariant. Thus, whenever we use sufficient preconditions, we must be prepared to roll back the heuristic in case of unsuccessful termination. (If we find that we have to remove an A_k from the set of invariant requirements, the justifications allow us to easily locate the marked (A_j, e) pairs that have to be unmarked.)

Fourth, after a few iterations of the heuristic, it is quite possible that our insight into the system improves, and we are able to guess a state formula Q that we believe is invariant and relevant to establishing *Invariant*(A_0) (i.e., it allows us to mark some unmarked (A_j, e) pairs.) We simply add Q to the set of invariant requirements, after making sure that *Initial* $\Rightarrow Q$ holds. In the same way, we can incorporate a property Q that is known *a priori* to be invariant, except that in this case we need not worry about marking Q wrt to the events.

4.6 Proof Rules for Leads-To

We have two kinds of leads-to proof rules. The first kind is for inferring leads-to assertions from the system specification:

Leads-to via Event Set Rule. Given a system A with weak fairness for event set $E \subseteq Events_A$, P *leads-to* Q is satisfied by A if the following hold:

(i) for every event $e \in E$: $\{P\}e\{Q\}$

(ii) for every event $e \in Events_A - E$: $\{P\}e\{P \vee Q\}$

(iii) *Invariant*$(P \Rightarrow enabled_A(E))$ is satisfied by A

Proof of Soundness. Let $\sigma = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ be an allowed behavior of system A. We have to show that the conditions of the rule imply that σ satisfies P *leads-to* Q. Let s_i satisfy P (if there is no such i, P *leads-to* Q holds vacuously). We have to show that there exists an s_j, $j \geq i$, in σ such that s_j satisfies Q. Let us assume the negation:

(iv) for all s_k in σ, $k \geq i$, s_k does not satisfy Q.

From (i) and (ii), we know that if s_k satisfies P and if s_k is not the last state in σ, then s_{k+1} satisfies $P \vee Q$. From (iv), we know that s_{k+1} does not satisfy Q. Thus, every s_k in σ, $k \geq i$ satisfies P, and hence the enabling condition of E (from condition (iii)). Also from condition (i), we know that $e_k \notin E$ for e_k in σ, $k \geq i$ (otherwise, Q would hold). If σ is finite, then E is enabled in the last state. If σ is infinite, then E is continuously enabled but never occurs. Either case is not possible because σ is an allowed behavior and E has weak fairness. □

If event set E has strong fairness instead of weak fairness, then in condition (iii) of the above rule we can replace *Invariant*$(P \Rightarrow enabled_A(E))$ by the weaker P *leads-to* $Q \vee enabled_A(E)$ (see Lam and Shankar [1990] for details).

The second kind of leads-to proof rule is for inferring leads-to assertions from other leads-to assertions. We refer to them as **closure rules**. Here are some obvious examples (for a complete treatment, see Chandy and Misra [1988] and Manna and Pnueli [1984; 1992]):

- P *leads-to* Q holds if $Invariant(P \Rightarrow Q)$ holds.
- P *leads-to* Q holds if for some R, P *leads-to* R and R *leads-to* Q hold.
- P *leads-to* Q holds if for some R, P *leads-to* $Q \vee R$ and R *leads-to* Q hold.
- P *leads-to* Q holds if $P = P_1 \vee P_2$, P_1 *leads-to* Q, and P_2 *leads-to* Q hold.
- P *leads-to* Q holds if $Invariant(R)$ and $P \wedge R$ *leads-to* $R \Rightarrow Q$ hold.

Sometimes we have to apply a closure rule N times, where N is a parameter of the problem being solved. For such cases, we need a generalization based on well-founded structures[16] [Manna and Pnueli 1984; Chandry and Misra 1988; Lamport 1991]:

Leads-To Well-Founded Closure Rule. Let $(Z, >)$ be a well-founded structure. Let $F(w)$ be a state formula with parameter $w \in Z$. P *leads-to* Q is satisfied if the following hold:

(i) P *leads-to* $Q \vee [\exists x: F(x)]$

(ii) $F(w)$ *leads-to* $Q \vee [\exists x < w: F(x)]$

Different instances of well-founded structures are appropriate in different situations. One special case of a well-founded structure is the natural integers; here the ordering is total. Another special case is the partial ordering induced by set inclusion on the subsets of a countable set.

When we use these rules in our examples, for brevity we shall say P *leads-to* Q *holds via* E to mean that P, Q, and E satisfy the conditions of the above leads-to via event set rule. We shall say P *leads-to* Q *holds by closure of* $L_1, L_2, \ldots$, where $L_1, L_2, \cdots$ are other assertions, to mean that P *leads-to* Q holds by applying the closure rules to $L_1, L_2, \ldots$. Finally, it is often the case when P *leads-to* Q holds via E, that only a subset F of the events in E are enabled when P holds. To explicitly indicate this, we say that P *leads-to* Q *holds via* $F \subseteq E$; formally, this means that P *leads-to* Q holds via E and $P \Rightarrow \neg\, enabled(E - F)$ holds.

[16]A well-founded structure $(Z, >)$ is a partial order $>$ on a nonempty set Z such that there is no infinite descending chain $z_1 > z_2 > \cdots$ where each $z_i \in Z$.

Accumulator Example (Continued)

Let us prove that the accumulator program (at the end of Section 4.4) satisfies

$$L_0 \equiv [\forall i: b(i) = 0] \text{ } leads\text{-}to \text{ } [\forall i: b(i) = 1].$$

Recall that event set $E = \{e(0), \ldots, e(N-1)\}$ has weak fairness. We expect L_0 to hold, because for each i, if $b(i) = 0$ then $e(i)$ eventually sets $b(i)$ to 1, and after that $b(i)$ is not affected.

Define $J = \{i: b(i) = 1\}$. We expect the following to hold:

$$L_1 \equiv |J| = n < N \text{ } leads\text{-}to \text{ } |J| = n + 1.$$

L_1 holds via event set E. The details are as follows: $|J| = n < N$ implies $enabled(E)$, because of the definition of J. For every event $e \in E$, $\{|J| = n < N\}e\{|J| = n + 1\}$, because e's occurrence increases $|J|$ by 1. There is no system event not in E.

The following can be derived from L_1 using N applications of the leads-to transitivity rule (or more precisely, using the well-founded closure rule with the naturals as the well-founded structure and $F(w) \equiv N - |J| = w$):

$$L_2 \equiv |J| = 0 \text{ } leads\text{-}to \text{ } |J| = N.$$

L_2 implies L_0 because of the definition of J (or more precisely, using the last closure rule). This completes the proof of L_0.

5. EXAMPLES OF CONCURRENT SYSTEMS ANALYSES

In this section, we consider some concurrent systems and analyze desirable properties. The weakest precondition heuristic for obtaining invariant requirements is illustrated.

Many of our examples involve sequences. If B is a set of values, then **sequence of** B denotes the set of finite sequences whose elements are in B. It includes the null sequence, denoted by $\langle\rangle$. For any sequence y, let $|y|$ denote the length of y, and let $y(i)$ denote the ith element in y, with the $y(0)$ being the leftmost element. Thus, $y = \langle y(0), \ldots, y(|y| - 1)\rangle$.

We say "*y prefix-of z*" to mean $|y| \leq |z|$ and $y = \langle z(0), \ldots, z(|y| - 1)\rangle$. For any nonnull sequence y, define $tail(y)$ to be $\langle y(1), \ldots, y(|y| - 1)\rangle$; i.e., y with the leftmost element removed. For any sequence y, define $head(y)$ to be $y(0)$ if y is nonnull and a special value *nil* (that is not in any variable's domain) if y is null. We use @ as the concatenation operator for sequences. Given two sequences y and z, $y@z$ is the sequence $\langle y(0), \ldots, y(|y| - 1), z(0), \ldots, z(|z| - 1)\rangle$.

5.1 A Bounded-Buffer Producer-Consumer Example

Consider a producer process and a consumer process that share a FIFO buffer of size $N > 0$. The producer can append data items to the buffer. The consumer can remove data items from the buffer. To avoid buffer overflow, the processes maintain a variable indicating the number of spaces in the buffer. We assume that if the buffer is not empty, the consumer process eventually consumes the item at the head of the buffer. However, we do not require the producer to repeatedly produce data items. This system can be modeled as follows, where *DATA* denotes the set of data items that can be produced, and parameter *data* has domain *DATA*:

State variables and initial condition:
buffer: sequence of *DATA*. Initially $\langle\rangle$. {The FIFO buffer shared between the processes}
numspaces: integer. Initially N. {The number of spaces currently available in the buffer}

Events:
Produce(*data*)
  enabled $\equiv numspaces \geq 1$
  action $\equiv buffer \leftarrow buffer@\langle data\rangle;$
    $numspaces \leftarrow numspaces - 1$
Consume(*data*)
  enabled $\equiv head(buffer) = data$
  action $\equiv buffer \leftarrow tail(buffer);$
    $numspaces \leftarrow numspaces + 1$

Fairness requirements:
{*Consume*(*data*): $data \in DATA$} has weak fairness.

Suppose we want to say that the buffer never overflows. This safety property can be specified by the following assertion:

(1) $Invariant(|buffer| \leq N)$.

Suppose we want to specify that data items are consumed in the order they were produced. This is a safety property. We cannot specify this property on the above model because it does not have any state variables that indicate the order of production or of consumption. So we introduce two auxiliary variables as follows:

- *produced*: sequence of *DATA*. Initially $\langle\rangle$. {Records data blocks in the order they are produced}
- *consumed*: sequence of *DATA*. Initially $\langle\rangle$. {Records data blocks in the order they are consumed}

The events are modified as follows (note that the auxiliary variable condition is satisfied).

- $produced \leftarrow produced@\langle data\rangle$ is added to the action of *Produce*(*data*).
- $consumed \leftarrow consumed@\langle data\rangle$ is added to the action of *Consume*(*data*).

The desired property can now be formalized by the following assertion:

(2) *Invariant*(*consumed prefix-of produced*).

Suppose we want to say "the buffer is empty just before data is produced." How can we specify this safety property? We can rephrase it as "the buffer is empty whenever data *can* be produced, i.e., whenever *Produce(data)* is enabled," and this is specified by the following assertion:

(3) $Invariant(numspaces > 0 \Rightarrow |buffer| = 0)$.

Suppose we want to say that whatever is produced is eventually consumed. This is a progress property. It can be specified by the following assertion:

(4) $produced = n$ *leads-to* $consumed = n$.

Given the safety assertion *Invariant(consumed prefix-of produced)*, it can also be specified by

(5) $|produced| = n$ *leads-to* $|consumed| = n$.

Suppose we want to say that the producer keeps producing data blocks. This progress property can be specified by the following assertion:

(6) $|produced| = n$ *leads-to* $|produced| = n + 1$.

Assertions (1), (2), (4), and (5) are satisfied by the system. Assertion (3) is not satisfied unless $N = 1$ or $N = 0$. Assertion (6) is not satisfied; it would be satisfied if, say, $\{Produce(data): data \in DATA\}$ is subject to weak fairness. We next give proofs of assertions (1), (2), and (5).

Proof of a Safety Property. Let us prove that $Invariant(A_0)$ holds for the system, where

$$A_0 \equiv |buffer| \leq N.$$

We use the heuristic for generating invariant requirements. The first step is to check whether $Initial \Rightarrow A_0$, which it does. Next, we set up the marking.

We represent the marking by a table that has a row for each invariant requirement and a column for each system event. For an invariant requirement A_i and an event e, if (A_i, e) is unmarked, its entry in the table is blank. If (A_i, e) is marked, its entry indicates the justification, i.e., a subset of the invariant requirements such that their conjunction is a sufficient precondition of A_i wrt e. Thus, the reader can easily check the validity of the marking. Also, an (A_i, e) entry in the table contains NA to indicate that e does *not affect* any of the state variables of A_i; thus $\{A_i\}e\{A_i\}$ holds trivially. Finally, just to remind ourselves that each invariant requirement must be checked first against *Initial*, we add a column for that purpose. It will contain OK to indicate that $Initial \Rightarrow A_j$ holds.

At the start, we have the following marking.

	Initial	*Produce(data)*	*Consume(data)*
A_0	OK		

Next we try to mark the unmarked (A_j, e) pairs. We can mark $(A_0, Consume(data))$ with the justification A_0 because $\{A_0\}Consume(data)\{A_0\}$ holds (since *Consume(data)* decreases $|buffer|$ by 1).

What about $(A_0, Produce(data))$? *Produce(data)*'s action increases $|buffer|$ by 1. So in order for A_0 to hold, it is necessary (and sufficient) that $|buffer| \leq N - 1$ hold whenever *Produce(data)* is enabled. That is, the following is a weakest precondition of A_0 wrt *Produce(data)*:

$$A_1 \equiv numspaces \geq 1 \Rightarrow |buffer| \leq N - 1.$$

A_1 is not implied by A_0, so we can add A_1 as an invariant requirement. But A_1 reminds us that we need to investigate the relationship between *numspaces* and *buffer*. In fact, the relationship is obvious; *numspaces* indicates the number of spaces in *buffer*:

$$A_2 \equiv |buffer| + numspaces = N.$$

Observe that A_2 implies A_1. A_2 holds initially. $\{A_2\}Produce(data)\{A_2\}$ holds because *Produce(data)* decrements *numspaces* by 1 and increments $|buffer|$ by 1. $\{A_2\}Consume(data)\{A_2\}$ holds in the same way, with *numspaces* and $|buffer|$ interchanged. We are done. Our proof is summarized in the following marking, where * is used to indicate an old entry:

	Initial	*Produce(data)*	*Consume(data)*
A_0	*	A_2	A_0
A_2	OK	A_2	A_2

In fact A_2 implies A_0, so we can remove the A_0 row in the above marking and just note that $A_2 \Rightarrow A_0$ holds. Is A_2 the weakest state formula that satisfies the invariance rule and implies A_0? See Note 1 in Appendix 1 for the answer. □

Proof of a Safety Property. Let us prove that $Invariant(B_0)$ holds for the system, where

$$B_0 \equiv consumed\ prefix\text{-}of\ produced.$$

B_0 holds initially. $\{B_0\}Produce(data)\{B_0\}$ holds because *Produce(data)* appends *data* to the right of *produced* and does not affect *consumed*. At this point, we have the following marking:

	Initial	*Produce(data)*	*Consume(data)*
B_0	OK	B_0	

Consider (B_0, *Consume(data)*). *Consume* preserves B_0 iff *head(buffer)* is the next element in sequence after the last element in *consumed*. That is, the following is a weakest precondition of B_0 wrt *Consume(data)*:

$$B_1 \equiv |buffer| > 0 \Rightarrow consumed@\langle buffer(0)\rangle\ prefix\text{-}of\ produced.$$

B_1 reminds us that we need to investigate the relationship between *buffer*, *produced*, and *consumed*. In fact, *buffer* stores whatever has been produced and not yet consumed. This gives us the following:

$$B_2 \equiv produced = consumed@buffer.$$

Note that B_2 implies B_0 and B_1. B_2 holds initially. $\{B_2\}Produce(data)\{B_2\}$ holds because *Produce(data)* appends *data* to the right of *produced* and to the right of *buffer*, and it does not affect *consumed*. $\{B_2\}Consumed(data)\{B_2\}$ holds because *Consume(data)* transfers the leftmost element in *buffer* and appends it to the right of *consumed*, provided *buffer* was nonnull before the occurrence. We are done, as summarized in the following marking, where we also indicate that B_2 implies B_0:

	Initial	*Produce(data)*	*Consume(data)*
B_2	OK	B_2	B_2

$B_2 \Rightarrow B_0$

Is B_2 the weakest state formula that satisfies the invariance rule and implies B_0? See Note 2 in Appendix 1 for the answer. □

Note that the proofs of $Invariant(A_2)$ and $Invariant(B_2)$ are independent of each other.

Proof of a Progress Property. Let us prove that L_0 holds for the system, where

$$L_0 \equiv |produced| \geq n\ leads\text{-}to\ |consumed| \geq n.$$

We expect this to hold because as long as $|produced| \geq n$ and $|consumed| < n$ hold, the buffer is not empty, and eventually *Consume* will extend *consumed*. Consider the following assertion:

$$L_1 \equiv |produced| \geq n \wedge |consumed| = m < n\ leads\text{-}to\ |produced| \geq n \wedge |consumed| = m + 1$$

Using closure rules, we can derive L_0 from L_1 (make sure of this). Thus, it suffices to establish L_1.

We now show that L_1 holds via event set $E = \{Consume(data)\colon data \in DATA\}$. The details are as follows: Because *consumed@buffer = produced* (from B_2 or B_4), $|produced| \geq n \wedge |consumed| = m < n$ implies $buffer \neq \langle\rangle$, which implies *enabled(E)*. The occurrence of any event in E (i.e., *Consume(data)* for any *data*) establishes $|produced| \geq n \wedge |consumed| \geq$

$m + 1$. The occurrence of any event not in E (i.e., *Produce*(*data*) for any *data*) preserves $|produced| \geq n \wedge |consumed| = m < n$. □

5.2 An Interacting-Loops Example

Consider the following program written in a traditional concurrent programming language.

```
x: integer. Initially x = 4.
cobegin
        repeat x ← 1 until x = 3
    ||
        repeat if x = 1 then x ← 2
               else x ← 4 until x = 3
    ||
        repeat if x = 2 then x ← 3
               else x ← 4 until x = 3
coend
```

Assuming that each *if* statement and each $x = 3$ test is atomically executed, we can model the above program by the following system, where state variables a, b, and c represent the control variables of the three processes.

State variables and initial condition:
x: integer. Initially $x = 4$.
a, b, c: integer. Initially $a = b = c = 0$.

Events:

```
e_1
  enabled ≡ a = 0
  action ≡ x ← 1; a ← 1
e_2
  enabled ≡ a = 1
  action ≡ if x = 3 then a ← 2
                else a ← 0
f_1
  enabled ≡ b = 0
  action ≡ if x = 1 then x ← 2
                else x ← 4;
           b ← 1
f_2
  enabled ≡ b = 1
  action ≡ if x = 3 then b ← 2
                else b ← 0
g_1
  enabled ≡ c = 0
  action ≡ if x = 2 then x ← 3
                else x ← 4;
           c ← 1
g_2
  enabled ≡ c = 1
  action ≡ if x = 3 then c ← 2
                else c ← 0
```

Fairness requirements:
$\{e_1, e_2\}, \{f_1, f_2\}, \{g_1, g_2\}$ have weak fairness.

Proof of a Safety Property. It is obvious that x equals 3 if the program terminates; that is, *Invariant*(A_0) holds, where $A_0 \equiv a = b = c = 2 \Rightarrow x = 3$, because A_0 satisfies the invariance rule. □

Proof of Negation of a Safety Property. Does x equal 3 if only some of the loops have terminated? That is, does *Invariant*(B_0) hold, where

$$B_0 \equiv a = 2 \vee b = 2 \vee c = 2 \Rightarrow x = 3.$$

We shall prove that *Invariant*(B_0) does not hold, by providing a finite behavior that ends in a state that does not satisfy B_0. Let us represent the system state by the value of the 4-tuple (x, a, b, c). Consider the following sequence of alternating states and events:

$$\sigma = \langle (4,0,0,0), e_1, (1,1,0,0), e_2, (1,0,0,0), f_1, (2,0,1,0), g_1, (3,0,1,1), f_2, (3,0,2,1), e_1, (1,1,2,1) \rangle$$

It is easy to check that σ is a behavior of the system and that its last state $(1,1,2,1)$ does not satisfy B_0. (Is there a shorter counterexample?) □

Proof of Negation of a Progress Property. Does the system eventually terminate? That is, does L_0 hold, where

$$L_0 \equiv a = b = c = 0 \textit{ leads-to } a = b = c = 2.$$

It is obvious that the system has terminating behaviors, for example, the following behavior (for brevity, we show only the sequence of events in a behavior):

$$\sigma_0 = \langle e_1, f_1, g_1, e_2, f_2, g_2 \rangle.$$

It is also obvious that the system has nonterminating behaviors, for example, the infinite behavior $\langle e_1, e_2, e_1, e_2, \cdots \rangle$. However, we cannot conclude from this behavior that the system does not satisfy

L_0 because it is not an allowed behavior (e.g., f_1 is continuously enabled but never occurs).

We now disprove L_0 by demonstrating allowed behaviors that are nonterminating. In fact, we show something stronger, that the system can reach a state from where termination is impossible. Consider the following behaviors:

$$\sigma_1 = \langle e_1, e_2, f_1, g_1, f_2, g_2, e_1, e_2, e_1, \cdots \rangle$$
$$\sigma_2 = \langle e_1, e_2, f_1, g_1, g_2, e_1, f_2, e_2, \cdots \rangle$$

After a certain point in behavior σ_1, only e_1 and e_2 are active. After a certain point in behavior σ_2, only e_1, e_2, f_1, and f_2 are active. In both cases, g_1 and g_2 are permanently disabled. Because g_1 is the only event that can set x to 3, the system has no possibility of terminating after that point. □

6. DISTRIBUTED SYSTEM MODEL

In this section, we specialize our system model for message-passing distributed systems. We consider a distributed system to be an arbitrary directed graph whose nodes are processes and whose edges are one-way communication channels. A channel can be either **perfect** or **imperfect**. A perfect channel is a FIFO buffer. An imperfect channel can lose, reorder, and/or duplicate messages in transit.

The state transition system modeling the distributed system has a state variable for each channel indicating the sequence of messages traveling in the channel.[17] For each process, there is a set of nonauxiliary state variables. Additionally, the system can have other state variables that are auxiliary.

The state transition system has events for each process and for each imperfect channel; a perfect channel has no events (enqueuing and dequeuing of messages is done by send and receive primitives in process events). The events of an imperfect channel model the imperfections of the channel; they can access (read or write) only the channel state variable and auxiliary state variables. The events of a process can access only auxiliary state variables, the state variables of that process, and state variables of channels connected to the process.[18] Furthermore, a process event can access an outgoing (incoming) channel state variable only by sending (receiving) messages.

To formalize these constraints, let $transit_{ij}$ denote the state variable for a channel (i, j) from process i to process j. Define the following operations:

$$Send(transit_{ij}, m) \equiv transit_{ij} \leftarrow transit_{ij} @ \langle m \rangle$$

$$Remove\ (transit_{ij}) \equiv transit_{ij} \leftarrow tail(transit_{ij})$$

6.1 Blocking Channels

We first consider distributed systems with **blocking channels**, i.e., where a channel can block a process from sending a message into it. Specifically, let state formula $ready(transit_{ij})$ be $true$ iff channel (i, j) is ready to accept a message from the process, for example, $ready(transit_{ij}) \equiv |transit_{ij}| < N$. Each event of process i is one of the following three types:

- An internal event e (i.e., does not send or receive messages) has the form:

$$enabled \equiv P$$
$$action \equiv S$$

[17] The practice of modeling a channel by the sequence of messages in transit has been used by Chandy and Misra [1988] and in networking literature (e.g., Knuth [1981] and Shankar and Lam [1983]). Another way to model a channel is by the sequence of messages sent into it and the sequence of messages received from it [Hailpern and Owicki 1983].

[18] In fact, these conditions can be relaxed further: a process or channel event can read nonauxiliary state variables of other processes and channels provided their values are used only to update auxiliary state variables

- A send event $e(m)$ that sends message m into an outgoing channel (i,j) has the form:

$$enabled \equiv P \wedge ready(transit_{ij})$$
$$action \equiv S; Send(transit_{ij}, m)$$

- A receive event $e(m)$ that receives message m from an incoming channel (j,i) has the form:

$$enabled \equiv P \wedge head(transit_{ji}) = m$$
$$action \equiv Remove(transit_{ji}); S$$

where P and S do not access nonauxiliary state variables of other processes or channels.[19]

6.2 Nonblocking Channels

We next consider distributed systems with **nonblocking channels**, i.e., a channel never blocks a process from sending a message. Such a distributed system can be modeled as above, with $ready(transit_{ij}) \equiv true$. However, in this case a simpler model can be used [Lamport 90]. We can classify the events of process i into those that receive a message and those that do not receive a message, as follows:

- A receive event $e(m)$ that receives message m from an incoming channel (j,i) has the form:

$$enabled \equiv P \wedge head(transit_{ji}) = m$$
$$action \equiv Remove(transit_{ji}); S$$

- An event e that does not receive a message has the form:

$$enabled \equiv P$$
$$action \equiv S$$

where P and S are as in the blocking case above, except that S can now include send operations on outgoing channels.

[19]Of course, they can access auxiliary variables provided the auxiliary variable condition is satisfied.

6.3 Channel Fairness Requirements

In order to prove useful progress properties for a distributed system, it is generally necessary to assume some fairness requirement for every channel (i,j) of the system. For each message m that can be sent into channel (i,j), let $E(m)$ denote the set of process events that can receive message m from channel (i,j). For any set of messages N, let $E(N) = \bigcup_{m \in N} E(m)$. We say the *receive events for channel* (i,j) are always ready iff $head(transit_{ij}) = m \Rightarrow enabled(E(m))$ is invariant for the distributed system; that is, there is always an event ready to receive whatever message is at the head of channel (i,j).

For every perfect channel (i,j), we assume the following fairness requirement: For any message set N, $\{E(N)\}$ has weak fairness. Thus, if the receive events are always ready, any message in the channel is eventually received.

For every imperfect channel (i,j), we assume the following fairness requirement: If the receive events for channel (i,j) are always ready, then for every allowed behavior σ of the distributed system and for every message set N, if messages from N are sent infinitely often in σ, then messages from N are received infinitely often in σ. Thus, any message that is repeatedly sent is eventually received [Hailpern and Owicki 1983]. (This is similar to strong fairness, and it is not implied by weak fairness for $\{E(N)\}$ for any message set N.)

The above fairness requirements justify the following proof rule, where $count(N)$ is a (auxiliary) variable indicating the number of times messages from N have been sent since the beginning of system execution.

Leads-to Via Message Set Rule. Given a channel whose receive events are always ready and a set of messages N that can be sent into the channel, P *leads-to* Q is satisfied if the following are satisfied:

(i) for every event $e \in E(N)$: $\{P\}e\{Q\}$
(ii) for every event $e \notin E(N)$: $\{P\}e\{P \vee Q\}$

(iii) $P \wedge count(N) \geq k$ *leads-to*
$Q \vee count(N) \geq k + 1$

The above rule is valid whether channel (i, j) is perfect or imperfect. However, it is typically used only when channel (i, j) is imperfect, because if channel (i, j) is perfect then the stronger leads-to via event set $E(N)$ rule can be used.

7. EXAMPLES OF DISTRIBUTED SYSTEMS ANALYSES

In this section, we present some distributed systems and analyze desirable properties. Throughout, we assume that channels are nonblocking.

7.1 A Data Transfer Protocol with Flow Control

A data transfer protocol is a distributed solution to the producer-consumer problem. Consider two processes, 1 and 2, connected by two one-way channels, (1, 2) and (2, 1). Process 1 produces data items and sends them to process 2 which consumes them. We want data to be consumed in the same order as it was produced. We have an additional requirement that the number of data items in transit must be bounded. (This prevents congestion if, as is often the case, the channel is implemented over a store-and-forward network.) We consider a simple scheme that achieves this, assuming that the channels are error-free. Process 2 acknowledges each data item it receives. Process 1 allows at most N data items to be *outstanding*, i.e., sent and not acknowledged.

The protocol specification follows, where *DATA* denotes the set of data items that can be produced:

Process 1 state variables, initial condition, and events:

```
produced: sequence of DATA.
  Initially ⟨⟩.
  Auxiliary variable recording the sequence of data items produced.
s: integer. Initially 0.
  Indicates number of data items sent.
a: integer. Initially 0.
  Indicates number of data items acknowledged.
Produce(data)
  enabled ≡ $s - a < N$
  action ≡ $produced \leftarrow produced@\langle data\rangle$;
           $s \leftarrow s + 1$;
           $Send(transit_{1,2}, data)$
RecACK
  enabled ≡ $head(transit_{2,1}) = ACK$
  action ≡ $Remove(transit_{2,1})$;
  $a \leftarrow a + 1$
```

Process 2 state variables, initial condition, and events:

```
consumed: sequence of DATA
  Initially ⟨⟩.
  Auxiliary variable recording the sequence of data items consumed.
RecDATA(data)
  enabled ≡ $head(transit_{1,2}) = data$
  action ≡ $Remove(transit_{1,2})$;
           consumed
           $\leftarrow consumed@\langle data\rangle$;
           $Send(transit_{2,1}, ACK)$
```

Channels (1, 2) and (2, 1), state variable, and initial condition:

$transit_{1,2}$, $transit_{2,1}$: sequence of messages. Initially $\langle\rangle$.
(There are no channel events because the channels are error-free.)

Fairness requirements:

Fairness requirements for the channels.

Proof of a Safety Property. Let us prove that the number of data items in transit does not exceed N; that is, prove $Invariant(A_0)$, where

$$A_0 \equiv |transit_{1,2}| \leq N.$$

Produce(*data*) is the only event that can falsify A_0. It is enabled only if $s - a < N$ holds. Note that for every data item outstanding at process 1, either the data item is in transit in channel (1, 2), or an acknowledgment for it is in transit in channel (2, 1). That is, we expect the following to be invariant:

$$A_1 \equiv s - a = |transit_{1,2}| + |transit_{2,1}|.$$

Because process 1 allows at most N items to be outstanding, we expect the following to be invariant:

$$A_2 \equiv s - a \leq N.$$

This completes the proof because $A_1 \wedge A_2$ implies A_0, and each of A_1 and A_2 satisfies the invariance rule. Is $A_1 \wedge A_2$ the weakest state formula that satisfies the invariance rule and implies A_0? See Note 3 in Appendix 1. □

Proof of a Safety Property. Let us prove $Invariant(B_0 \wedge B_1)$, where

$$B_0 \equiv consumed\ prefix\text{-}of\ produced$$

$$B_1 \equiv |produced| - |consumed| \leq s - a.$$

Because channel (1, 2) is a perfect FIFO buffer, we expect the following to be invariant:

$$B_2 \equiv consumed@transit_{1,2} = produced.$$

B_2 implies B_0 and $|produced| - |consumed| = |transit_{1,2}|$. We know that $|transit_{1,2}| \leq s - a$ (from A_1). The proof is complete, as summarized in the following marking:

	Initial	*Produce (data)*	*RecACK*	*RecDATA (data)*
B_2	OK	B_2	NA	B_2

$B_2 \Rightarrow B_0$
$B_2 \wedge A_1 \Rightarrow B_1$ □

Proof of a Progress Property. Let us prove that L_0 holds, where

$$L_0 \equiv |produced| \geq n\ leads\text{-}to\ |consumed| \geq n.$$

We can establish L_0, proceeding as in the bounded-buffer producer-consumer example. Define the following assertion:

$$L_1 \equiv |produced| \geq n \wedge |consumed| = m < n\ leads\text{-}to\ |produced| \geq n \wedge |consumed| = m + 1.$$

L_0 can be derived from L_1 using closure rules. L_1 holds via event set $\{RecDATA(data): data \in DATA\}$ because of B_2 (make sure of this). □

7.2 A Shortest-Distance Algorithm

Consider a distributed system with an arbitrary but finite topology. Let the set of processes be $\{1, \ldots, N\}$, and let the set of channels be $E \subseteq \{1, \ldots, N\} \times \{1, \ldots, N\} - \{(i, i): i = 1, \ldots, N\}$. Associated with each channel (i, j) is a nonnegative length $D(i, j)$. We say process i is *reachable* iff $i = 1$ or if there is a path in E from process 1 to process i. For each reachable process i, let $D(i)$ indicate the length of a shortest path from process 1 to process i.

We consider an algorithm that informs each reachable node of its shortest distance from process 1. Specifically, each process i maintains an estimate of the shortest distance from process 1 in a variable $dist(i)$. Process 1 starts the algorithm by sending on every outgoing channel $(1, j)$ a message containing $D(1, j)$. When a process i receives a message d, it sends $d + D(i, j)$ on every outgoing channel (i, j) iff d is less than $dist(i)$.

The algorithm specification follows, where $i, j \in \{1, 2, \ldots, N\}$, and ∞ is considered to be greater than any number. For convenience, we assume an initial state where process 1 has already started the algorithm.

Process i state variable, initial condition, and events:

```
$dist(i)$: real. Initially $dist(1) = 0 \wedge$
                        $[\forall i \neq 1: dist(i) = \infty]$
    $Rec_i(j, d)$
  $enabled \equiv head(transit_{ji}) = d$
  $action \equiv$  $Remove(transit_{ji})$;
             if $d < dist(i)$ then begin
                 $dist(i) \leftarrow d$;
                 for all $(i, k) \in E_i$
                 $Send(transit_{ik},$
                 $d + D(i, k))$
end
```

Channel $(i, j), (i, j) \in E$, state variable, and initial condition:

```
$transit_{ij}$: sequence of messages.
           Initially = $\langle D(1, j) \rangle \wedge$
$transit_{1,j}$
$transit_{ij} = [\forall i \neq 1: \langle\ \rangle]$
```

Fairness requirements:
Fairness requirements for the channels.

Proof of a Safety Property. Let us prove that if $dist(i) \neq \infty$, then i is reachable and $dist(i)$ is the length of some path from 1 to i, that is, *Invariant*(A_0), where:

$$A_0 \equiv dist(i) = d \neq \infty \Rightarrow$$

there is a path from 1 to i of length d.

A_0 holds initially. The following is a sufficient precondition of A_0 wrt an occurrence of $Rec_i(j, d)$ that changes $dist(i)$:

$$A_1 \equiv d \in transit_{ji} \Rightarrow$$

there is a path from 1 to i of length d.

A_1 holds initially and is preserved by every event. This completes the proof, as summarized in the following marking:

	Initial	Rec_i
A_0	OK	A_1, A_0
A_1	OK	A_1

□

Proof of a Progress Property. Let us prove that every reachable process eventually learns its shortest distance, i.e., L_0 holds, where

$$L_0 \equiv i \text{ reachable } \textit{leads-to } dist(i) = D(i).$$

One way to prove this is to consider a shortest path from process 1 to process i and establish that each process on this path eventually informs its successor on the path of the successor's shortest distance from process 1. In the rest of the proof, let i be a reachable process; let $\langle j_0, \ldots, j_n \rangle$ be a shortest path from 1 to i where $j_0 = 1$ and $j_n = i$, and let k range over $\{0, \ldots, n-1\}$. Define the following progress assertion:

$$L_1 \equiv dist(j_k) = D(j_k) \textit{ leads-to}$$
$$dist(j_{k+1}) = D(j_{k+1}).$$

L_0 follows by closure of L_1 (specifically, using the chain rule with $F(n-k) \equiv dist(j_k) = D(j_k)$). Thus, it suffices to prove L_1.

Define state formula $H(l)$ to be true iff the lth message in $transit_{j_k, j_{k+1}}$ is $D(j_{k+1})$, that is, $H(l) \equiv transit_{j_k, j_{k+1}}(l) = D(j_{k+1})$. Define the following assertions, where for brevity we use X to denote $dist(j_k) = D(j_k) \wedge dist(j_{k+1}) > D(j_{k+1})$

$$A_2 \equiv X \Rightarrow [\exists l\colon H(l)]$$

$$L_2 \equiv X \wedge H(l) \wedge l > 0 \textit{ leads-to}$$
$$dist(j_{k+1}) = D(j_{k+1})$$
$$\vee (X \wedge H(l-1))$$

$$L_3 \equiv X \wedge H(0) \textit{ leads-to}$$
$$dist(j_{k+1}) = D(j_{k+1})$$

A_2 satisfies invariance rule 1. Each of L_2 and L_3 hold via event set $\{Rec_{j_k}(j_{k+1}, \cdot)\}$. L_1 follows by closure of *Invariant*(A_2), L_2, and L_3. □

Proof of a Progress Property. Let us prove that the algorithm eventually terminates, i.e., L_4 holds, where

$$L_4 \equiv \textit{Initial leads-to } transit_{i,j} = \langle\rangle.$$

Because of L_0, it suffices to prove the following:

$$L_5 \equiv [\forall \text{ reachable } i\colon dist(i) = D(i)]$$
$$\textit{leads-to } transit_{i,j} = \langle\rangle.$$

Let G denote the number of messages in transit in all channels. Define the following:

$$L_6 \equiv [\forall \text{ reachable } i\colon dist(i) = D(i)]$$
$$\wedge\, G = n > 0 \textit{ leads-to}$$
$$[\forall \text{ reachable } i\colon dist(i)$$
$$= D(i)] \wedge G < n$$

L_6 holds via event set $\{Rec_i\}$ using A_1. L_5 follows from closure of L_6. □

7.3 A Termination Detection Algorithm for Diffusing Computations

We consider an algorithm presented in Dijkstra and Scholten [1980] for detecting the termination of diffusion computations. Consider a distributed system with a set of processes $\{1, \ldots, N\}$ and a set of channels $E \subseteq \{1, \ldots, N\} \times \{1, \ldots, N\} - \{(i, i)\colon i = 1, \ldots, N\}$ such that if $(i, j) \in E$ then $(j, i) \in E$.

A diffusing computation is a distributed computation with the following features. Each process can be active or inactive. An active process can do local computations, send and receive messages, and spontaneously become inactive. An inactive process does nothing. An inactive process becomes active iff it receives a message. Initially, all processes are inactive except for a distinguished process, say process 1, and all channels are empty. (The shortest-distance algorithm in the previous section is a diffusion computation, with an initial state where process 1 has already sent out messages.)

The following specification models an arbitrary diffusing computation:

Process $i, i = \{1, \ldots, N\}$, **state variables, initial condition, and events**:

```
    $active_i$: boolean. Initially $true$ iff $i = 1$.
        Indicates whether or not process $i$
        is active.
    $vars_i$: integer.
        Variables of diffusing computation.

    $Local_i$
        $enabled \equiv active_i \wedge P$
        $action \equiv S$
    $Send_i(j, m)$
        $enabled \equiv active_i \wedge P$
        $action \equiv Send(transit_{ij}, m); S$
    $Rec_i(j, m)$
        $enabled \equiv head(transit_{ji}) = m$
        $action \equiv Remove(transit_{ji}); S;$
                $active_i \leftarrow true$
    $Deactivate_i$
        $enabled \equiv active_i \wedge P$
        $action \equiv S; active_i \leftarrow false$
```

where P denotes a state formula in $vars_i$, and S denotes an action in $vars_i$ and $active_i$; S can set $active_i$ to false. (The P's and S's of different events need not be the same.)

Channel (i, j), $(i, j) \in E$, **state variable, and initial condition**:

```
    $transit_{ij}$: sequence of messages.
            Initially $\langle\rangle$.
```

Fairness requirements:

(Arbitrary fairness requirements)

A diffusing computation is said to have *terminated* iff all processes are inactive and no messages are in transit. Note that once the computation has terminated, it can never leave the terminated state.

We now "superimpose" on the above diffusing computation system a termination detection algorithm that will allow process 1 to detect termination of the diffusing computation. The termination detection algorithm uses messages referred to as *signals* that are distinct from the messages of the diffusing computation. Henceforth, we use *message* to mean a diffusion computation message. For each message that is sent from process i to process j, a signal is sent at some time from process j to process i.

Each process is either *disengaged* or *engaged* to some other process. Initially, all processes are disengaged. Process i becomes engaged to process j iff i receives a message from j and was disengaged before the reception. While engaged, if process i receives a message from any process k, it responds by immediately sending a signal to k. Process i becomes disengaged by sending a signal to the process j to which it is engaged; process i can do this only if it is not active and it has received signals for all messages it sent.

Note that every active process is engaged and that a process can become engaged and disengaged several times during the course of the diffusion computation. We say that there is an engagement edge from i to j iff i is engaged to j. We shall see that the engaged processes and the engagement edges form an in-tree rooted at process 1; that is, for each engaged process i, there is a path from i to process 1. Thus, whenever process 1 has received a signal for every message it sent, it can deduce that no other process is engaged.

The specification of the termination detection algorithm, superimposed on the above model for an arbitrary diffusion computation, follows (for notational convenience, we let process 1 be always engaged to itself):

Process $i, i = \{1, \ldots, N\}$, **state variables, initial condition, and events**:

```
    $active_i$: boolean. Initially $true$ iff $i = 1$.
    $vars_i$: integer.
```

```
$deficit_i$: integer. Initially 0.
  The number of messages sent by process $i$ for which signals have not
  been received.
$engager_i$: $\{1, \ldots, N\} \cup \{nil\}$. Initially
  $engager_1 = 1 \wedge [\forall i \neq 1: engager_i = nil]$.
  $nil$ iff process $i$ is disengaged.
$Local_i$
  enabled $\equiv active_i \wedge P$
  action $\equiv S$
$Send_i(j, m)$
  enabled $\equiv active_i \wedge P$
  action $\equiv Send(transit_{ij}, m)$; $S$;
          $deficit_i \leftarrow deficit_i + 1$
$Rec_i(j, m)$
  enabled $\equiv head(transit_{ji}) = m$
  action $\equiv Remove(transit_{ji})$; $S$;
          $active_i \leftarrow true$;
          if $engager_i = nil$ then
            $engager_i \leftarrow j$
          else
            $Send(transit_{ij}, signal)$
$DeActivate_i$
  enabled $\equiv active_i \wedge P$
  action $\equiv S$; $active_i \leftarrow false$
$RecSignal_i(j)$
  enabled $\equiv head(transit_{ji}) = signal$
  action $\equiv Remove(transit_{ji})$;
          $deficit_i \leftarrow deficit_i - 1$
$DisEngage_i$ for $i \neq 1$
  enabled $= \neg active_i \wedge engager_i \neq nil \wedge deficit_i = 0$
  action $\equiv Send(transit_{ij}, signal)$
          where
  $j = engager_i$;
          $engager_i \leftarrow nil$
```

Channel $(i, j), (i, j) \in E$, **state variable, and initial condition**:

$transit_{ij}$: sequence of messages. Initially $\langle\,\rangle$.

Fairness requirements:

$\{Disengage_i\}$ has weak fairness.
Fairness requirements for channels.

Proof of a Safety Property. Let us prove that if $deficit_1 = 0$ and process 1 is not active, then the diffusion computation has terminated; i.e., no process is active, and no (diffusion computation) messages are in transit. Formally, let

$$Termination \equiv [\forall i: \neg active_i] \wedge \left[\forall (i, j) \in E: \text{ no messages in } transit_{ij}\right],$$

and let us prove $Invariant(A_0)$, where

$$A_0 \equiv deficit_1 = 0 \wedge \neg active_1 \Rightarrow Termination.$$

We next define some functions on the system state:

- F_{ij}, for every $(i, j) \in E$, is 1 if $engager_j = i$ and is 0 if $engager_j \neq i$.
- The set of engaged processes, $Engaged = \{i: engager_i \neq nil\}$.
- The set of engagement edges, $Engagements = \{(j, i): engager_j = i \wedge i \neq nil\} - \{(1, 1)\}$.

Note that 1 is an engaged process and $(1, 1)$ is not an engagement edge.

Define the following:

$A_1 \equiv deficit_i = \sum_{(i,j) \in E}$ number of messages in $transit_{ij}$ + number of signals in $transit_{ji}$ + F_{ij}.

$A_2 \equiv i \in Engaged - \{1\} \Rightarrow$ there is a path of engagement edges from i to 1.

$A_3 \equiv active_k \Rightarrow k \in Engaged.$

$A_4 \equiv deficit_k > 0 \Rightarrow k \in Engaged.$

A_1 implies that if $deficit_i = 0$ then process i has no incoming engagement edge. A_2 implies that the engaged nodes and the engagement edges form an in-tree rooted at process 1; recall that each engaged process other than 1 has exactly one outgoing edge and that process 1 has no outgoing edge. A_3 and A_4 imply that a disengaged process is not active and has zero deficit. It can be checked that A_1 satisfies invariance rule 1 and that $A_2 \wedge A_3 \wedge A_4$ satisfies invariance rule 1 given that A_1 is invariant. Also $A_1 \wedge A_2 \wedge A_3 \wedge A_4$ implies A_0, as follows: Assume $deficit_1 = 0$; because of A_1 and A_2, this implies that $i \notin Engaged$ for $i \neq 1$, which implies $\neg active_i$ (from A_3) and $deficit_i = 0$ (from A_4), which implies that no messages are in transit (from A_1).

The proof is summarized in the following marking, where we have abbreviated the event names:

	Initial	*Local*	*Send*	*Rec*
A_1	OK	NA	A_1	A_1
A_2	OK	NA	NA	A_4, A_2, A_1
A_3	OK	NA	NA	*true*
A_4	OK	NA	A_4	*true*

	DeAct	*RecSig*	*DisEng*
A_1	NA	A_1	A_1
A_2	NA	NA	A_1
A_3	*true*	NA	*true*
A_4	NA	A_4	*true*

$A_1 \wedge A_2 \wedge A_3 \wedge A_4 \Rightarrow A_0$ □

Proof of a Progress Property. Let us prove that if the diffusion computation has terminated, then eventually $deficit_1$ becomes 0; that is, L_0 holds, where

$$L_0 \equiv \textit{Termination leads-to } deficit_1 = 0.$$

We shall prove that once *Termination* holds, the leaf nodes of the engagement tree keep leaving the engagement tree until it consists of only process 1. Define the following functions on the system state:

- The set of leaf nodes, $Leaves = \{i: i \in Engaged \wedge [\forall j: (j, i) \notin Engagements]\}$.
- State formula $H(n, m) = true$ iff $|Engaged| = n$ and $(\sum_{i \in Leaves} deficit_i) = m$.

Define the following progress assertions:

$$L_1 \equiv Termination \wedge H(n, m) \wedge n > 1 \wedge m > 0 \textit{ leads-to } Termination \wedge (H(n, m-1) \vee [\exists l: H(n-1, l)])$$

$$L_2 \equiv Termination \wedge H(1, m) \wedge m > 0 \textit{ leads-to } Termination \wedge H(1, m-1)$$

$$L_3 \equiv Termination \wedge H(n, 0) \wedge n > 1 \textit{ leads-to } Termination \wedge [\exists l: H(n-1, l)]$$

Each of L_1 and L_2 holds via event set $\{RecSignal_i: i \in Leaves\}$. L_3 holds via event set $\{DisEngage_i: i \in Leaves\}$. In each case, the event set is not empty because *Leaves* is never empty.

Consider a lexicographic ordering of integer 2-tuples; i.e., $(j, k) < (n, m)$ iff $j < n$ or $j = n \wedge k < m$. Then, we get the following from the closure of L_1, L_2, and L_3:

$$L_4 \equiv Termination \wedge H(n, m) \wedge (n, m) > (1, 0) \textit{ leads-to } Termination \wedge [\exists (j, k) < (n, m): H(j, k)]$$

Using the chain rule on this, we get

$$L_5 \equiv Termination \wedge H(n, m) \textit{ leads-to } H(1, 0)$$

Because process 1 is always engaged, $[\exists (n, m) \geq (1, 0): H(n, m)]$ is invariant. From the definition of $H(n, m)$, we have that $H(1, 0)$ implies $deficit_1 = 0$. Combining these with L_5, we obtain L_0. □

8. DISCUSSION

In this tutorial, we focused on the following: Given a concurrent system S and desired properties P, express P in terms of safety and progress assertions and prove that S satisfies P using a set of proof rules. We modeled a system by a set of state variables, a set of events each with an enabling condition and an action, and a set of fairness requirements on the events. We used only invariant and leads-to assertions (but other kinds of assertions can be easily added). We introduced auxiliary state variables whenever needed to express a correctness property.

As discussed in Section 3.4, our system model, assertion language, and proof rules are similar to those of other authors, for example, Lamport [1989; 1990], Lynch and Tuttle [1987], Manna and Pnueli [1984; 1992], Chandy and Misra [1986; 1988], Back and Kurki-Suonio [1988], and Abadi and Lamport [1988; 1990]. The formalism in this tutorial comes to a large extent from Lam and Shankar [1990]. These references also contain many examples of assertional analyses. Other examples may be found in Dijkstra [1965; 1976; 1977], Dijkstra and Schloten [1980], Dijkstra et al. [1978; 1983], Drost and Leeuwen [1988], Drost and Schoone [1988], Hailpern and Owicki, [1983], Knuth [1981], Lamport [1982; 1987], Murphy and Shankar [1991], Andrews [1989], Schneider and

Andrews [1986], Schoone [1987], Shankar [1989], Shankar and Lam [1983; 1987], Tel [1987], Tel et al. [1988], and Alaettinoglu and Shankar [1992] (this list is only a sampling).

Assertional reasoning allows the proof of a system property to be presented at a convenient level of detail, by omitting obvious details of proof rule applications. (Of course, what is obvious to one person may not be so to another person.)

The difficulty in proving a system property P is in coming up with additional properties Q such that Q satisfies the proof rules and implies P. The most successful approach to obtaining the additional properties of Q seems to be to develop them while developing the system, as demonstrated by Dijkstra [1976; 1977] in his numerous program derivations; see also his paper "Two Starvation-Free Solutions of a General Exclusion Problem," EWD 625, Plataanstraat 5,5671, Al Nunen, The Netherlands, date unknown. In this approach, one starts with a skeleton system and a set of desired properties and successively adds (and modifies) states variables, events, and desired properties. The process ends when we have a system and a set of properties that satisfy the proof rules. This approach has been formalized into stepwise refinement techniques by several authors. See, for example, Abadi and Lamport [1988], Back and Kurki-Suonio [1988], Back and Sere [1990], Chandy and Misra [1986; 1988], Lamport [1983; 1989], Lynch and Tuttle [1987], and Shankar and Lam [1992].

Typically, a concurrent system S consists of smaller concurrent systems $S_1, \ldots, S_n$ that interact via message-passing primitives (including procedure calls) or shared variables. (The structure can be hierarchical in that S_i can itself consist of concurrent systems.) For example, a data transfer protocol consists of a producer system, a consumer system, and two channel systems; an operating system may consist of a process management system, a memory management system, and a file system. We can analyze such a composite system S by ignoring its subsystem structure (and that is what we did with distributed systems in this tutorial). Although this is efficient for small systems, it does not generally scale up to larger systems. A **compositional** approach is required, where we can prove that S satisfies a property P in two stages: first prove that each S_i satisfies some property P_i, and then prove that the P_i's together imply P.

For such an approach to work, we need a composition theorem of the kind: if each S_i satisfies P_i, then the composition of the S_i's satisfies the conjunction of the P_i's. It turns out that such composition theorems are not a straightforward matter. Typically, each P_i consists of an assumption on the environment of S_i and a requirement on S_i, and the difficulty is in avoiding circular reasoning of progress properties. There are several compositional approaches based on temporal logic in the literature. See, for example, Abadi and Lamport [1990], Chandy and Misra [1988], Lam and Shankar [1992], Lynch and Tuttle [1987], and Pnueli [1984]. Each places certain restrictions on the types of properties and compositions allowed.

So far, we have thought of P_i as a desired property of S_i, that is, a property of interest to the environment of S_i. Let us go one step further and think of an embellished P_i, say E_i, that includes *all* (and only those) properties of interest to the environment of S_i. In addition to safety and progress properties, E_i must include information needed for composing S_i with systems in its environment, such as which events and variables of S_i are visible to the environment, which transitions are controlled by the environment, and which by S_i, etc. Given E_i, we can think of S_i as an implementation of E_i. More importantly, we can replace S_i by another implementation T_i that satisfies E_i, without affecting the properties of S. Such a compositional approach is very useful for building, maintaining, and updating large concurrent systems. Perhaps the most difficult step in applying this approach is in identifying which properties of a system S_i are important to its environment. Such compositional theories are presented. See, for example, Abadi and Lamport [1990], Chandy and Misra [1988], Lam and Shankar [1982], and Lynch and Tuttle [1987].

APPENDIX 1

Note 1

Let us go back to the point just after we obtained A_1 and see whether a brute-force application of the heuristic would yield A_2. At this point, (A_1, *Produce*(*data*)) is unmarked.

$$\begin{aligned}
&wp(Produce(data), A_1)\\
&\quad \equiv enabled(Produce(data)) \Rightarrow\\
&\qquad wp(action(Produce(data), A_1)\\
&\quad \Leftrightarrow numspaces \geq 1 \Rightarrow (numspaces - 1\\
&\qquad \geq 1 \Rightarrow |buffer| + 1 \leq N - 1)\\
&\quad \Leftrightarrow numspaces \geq 1 \wedge numspaces \geq 2\\
&\qquad \Rightarrow |buffer| \leq N - 2\\
&\quad \Leftrightarrow numspaces \geq 2 \Rightarrow |buffer| \leq N - 2
\end{aligned}$$

Thus, we have a new invariant requirement:

$$A_3 \equiv numspaces \geq 2 \Rightarrow |buffer| \leq N - 2.$$

Note that A_3 is a weakest precondition of A_0 with respect to *two* successive occurrences of *Produce*, i.e., $wp(Produce, wp(Produce, A_0))$. If we consider a weakest precondition of A_0 with respect to k occurrences of *Produce*, where $1 \leq k \leq numspaces$, we get the following invariant requirement:

$$A_4 \equiv numspaces \geq k \Rightarrow |buffer| \leq N - k.$$

Observe that $A_4 \Rightarrow A_4[k/k - 1]$. So there is no need to consider A_4, for every k. It suffices to consider $A_4[k/numspaces]$, which is

$$numspaces \geq numspaces \Rightarrow$$
$$|buffer| \leq N - numspaces.$$

Because the antecedent is always *true*, the above is equivalent to its consequent, which yields:

$$A_5 \equiv |buffer| + numspaces \leq N$$

It is easy to see that $Initial \Rightarrow A_5$, $\{A_5\}Produce(data)\{A_5\}$, and $\{A_5\}Consume(data)\{A_5\}$ hold. Also $A_5 \Rightarrow A_0$ holds. Thus, A_5 is a weaker substitute for A_2. It is probably fair to say that when we obtained A_2, we felt that there is nothing weaker that satisfies A_0 and invariance rule 1. Now we see that this is not true. This is typical of how weakest preconditions throw additional light on even the simplest algorithms. Another way to think of it is that once we understand an algorithm, we usually know more about it than we need to.

Note 2

Let us go back to the point just after we obtained B_1 and see if a brute-force application of the heuristic would yield B_2. We derive the following weakest precondition of B_0 wrt k occurrences of *Consume*, where $k = |buffer|$ (similar to the derivation of A_5 above):

$$B_3 \equiv consumed@buffer$$
$$prefix\text{-}of\ produced.$$

We next obtain a weakest precondition of B_3 wrt *Produce*(*data*):

$$\begin{aligned}
&wp(Produce(data), B_3)\\
&\Leftrightarrow \quad consumed@buffer@\langle data\rangle\\
&\qquad prefix\text{-}of\ produced@\langle data\rangle\\
&\Leftrightarrow \quad consumed@buffer\ prefix\text{-}of\\
&\qquad produced\ \wedge\\
&\text{(i)} \quad ((|consumed| + |buffer| < |produced|\\
&\qquad \wedge data = produced(|consumed| +\\
&\qquad |buffer| + 1)) \vee\\
&\text{(ii)} \quad (|consumed| + |buffer| = |produced|))
\end{aligned}$$

Above, disjunct (i) requires the data item produced to equal a previously produced data item. Because this cannot be enforced in the current system, the only way to achieve the weakest precondition is to enforce disjunct (ii), which yields

$$B_4 \equiv consumed@buffer = produced.$$

So in this case, brute force has yielded the same invariant requirement as obtained by using intuition, i.e., B_2.

Note 3

Let us start from the beginning, just after A_0 is specified. *Produce(data)* is the only event that can falsify A_0. Let us obtain the weakest precondition of A_0 wrt the maximum possible number of occurrences of *Produce*. *Produce(data)* is enabled as long as $s - a < N$ holds. Therefore, starting from any state g where $s - a = k$, $N - k$ occurrences of *Produce* are possible. The resulting state satisfies A_0 iff g also satisfies $|transit_{1,2}| + N - k \leq N$, or equivalently $|transit_{1,2}| - (s - a) \leq 0$. In other words, we have the following weakest precondition of A_0 wrt $N - (s - a)$ occurrences of *Produce*:

$$A_3 \equiv |transit_{1,2}| \leq s - a.$$

RecACK is the only event that can falsify A_3. Starting from any state g where $|transit_{2,1}| = k$ holds, k occurrences of *RecACK* are possible. The resulting state satisfies A_3 iff g satisfies $|transit_{1,2}| \leq s - (a + k)$, or equivalently $|transit_{1,2}| \leq s - (a + |transit_{2,1}|)$. That is, we have the following weakest precondition of A_3 wrt $|transit_{2,1}|$ occurrences of *RecACK*:

$$A_4 \equiv |transit_{1,2}| + |transit_{2,1}| \leq s - a.$$

Note that A_4 implies A_3. No more assertions are needed to establish *Invariant*(A_0), as the following marking demonstrates:

	Initial	Produce (*data*)	*RecACK*	RecDATA (*data*)
A_0	OK	A_4	NA	A_0
A_4	OK	A_4	A_4	A_4

Finally, we observe that $A_0 \wedge A_4$ is weaker than $A_1 \wedge A_2$.

ACKNOWLEDGMENTS

The comments of the anonymous reviewers greatly improved the presentation and accessibility of this tutorial; in particular, Section 2 was suggested by reviewer 1. Richard Muntz suggested further improvements. Cengiz Alaettinoglu, Jean Bolot, and Carol Whitney carefully read various parts of the tutorial. Pravin Bhagwat and Debanjan Saha caught a mistake Jim Anderson and Ken Salem, both of whom taught from a previous version, provided invaluable feedback.

REFERENCES

ABADI, M., AND LAMPORT, L. 1990. Composing specifications. In *Stepwise Refinement of Distributed Systems*. Lecture Notes in Computer Science, vol 430, Springer-Verlag, New York. Also in *ACM Trans. Program. Lang. Syst. 15*, 1 (Jan 1993), 73–132.

ABADI, M , AND LAMPORT, L. 1988. The existence of refinement mappings Tech. Rep. Digital Systems Research Center, Palo Alto, Calif Also in *Theor Comput Sci 82*, 2 (May 1991), 253–284.

ALAETTINOGLU, C., AND SHANKAR, A. U 1992 Stepwise assertional design of distance-vector routing algorithms In *IFIP Proceedings of the 12 International Symposium on Protocol Specification, Testing, and Verification* (Orlando, Fla, June). IFIP, Arlington, Va.

ALPERN, B., AND SCHNEIDER, F 1985. Defining liveness *Inf. Process. Lett. 21*, 4 (Oct.), 181–185

ANDREWS, G. R 1989. A method for solving synchronization problems. *Sci Comput. Program 13*, 4 (Dec.), 1–21.

APT, K. R. 1981. Ten years of Hoare's logic A survey—Part I. *ACM Trans Program Lang. Syst 3*, 4 (Oct) 431–483

APT, K. R., FRANCEZ, N., AND KATZ, S. 1988. Appraising fairness in languages for distributed programming. *Distrib. Comput. 2*, 4, 226–241.

BACK, R J R , AND KURKI-SUONIO, R. 1988 Distributed cooperation with action systems. *ACM Trans Program. Lang. Syst 10*, 4 (Oct.), 513–554.

BACK, R J. R., KURKI-SUONIO, R. 1983. Decentralization of process nets with a centralized control. In the *2nd ACM SIGACT-SIGCOPS Symposium on the Principles of Distributed Computing* (Montreal, Aug.) ACM, New York, 131–142.

BACK, R. J. R., AND SERE, K. 1990. Stepwise refinement of parallel algorithms *Sci. Comput. Program. 13*, 2–3, 133–180.

CHANDY, K. M , AND MISRA, J 1988. *A Foundation of Parallel Program Design* Addison-Wesley, Reading, Mass

CHANDY, K. M., AND MISRA, J. 1986. An example of stepwise refinement of distributed programs: Quiescence detection *ACM Trans. Program. Lang. Syst. 8*, 3 (July), 326–343.

DIJKSTRA, E W 1977 A correctness proof for communicating processes—A small exercise. EWD-607, Burroughs, Nuenen, The Netherlands

DIJKSTRA, E. W. 1976 *A Discipline of Programming* Prentice-Hall, Englewood Cliffs, N.J.

DIJKSTRA, E. W. 1965 Solution of a problem in concurrent programming control. *Commun. ACM 8*, 9 (Sept)

DIJKSTRA, E. W., AND SCHOLTEN, C. S. 1980. Termination detection for diffusing computations. *Inf. Process. Lett 11* (Aug.), 1–4.

DIJKSTRA, E W., FEIJN, W. H. J., AND VAN GASTEREN, A. J. M. 1983. Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett. 16*, 217–219.

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., AND SCHOLTEN, C. S. 1978. On-the-fly garbage collection: An exercise in cooperation *Commun. ACM 21*, 11 (Nov.), 966–975.

DROST, N. J , AND SCHOONE, A. A. 1988. Assertional verification of a reset algorithm. Rijksuniversiteit Utrecht, RUU-CS-88-5.

DROST, N. J., AND VAN LEEUWEN, J. 1988. Assertional verification of a majority consensus algorithm for concurrency control in multiple copy databases. Rijksuniversiteit Utrecht, RUU-CS-88-13.

FLOYD, R. W. 1967. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*. Vol. 19. American Mathematical Society, 19–32.

FRANCEZ, N. 1986. *Fairness*. Springer-Verlag, New York.

GRIES, D 1981. *The Science of Programming*. Springer-Verlag, New York

HAILPERN, B. T., AND OWICKI, S. S. 1983. Modular verification of computer communication protocols. *IEEE Trans. Commun. COM-31*, 1 (Jan.), 56–68.

HOARE, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J.

HOARE, C A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct.), 576–583.

KNUTH, D. E. 1981. Verification of link-level protocols *BIT 21*, 31–36.

LAM, S. S., AND SHANKAR, A. U. 1992 Specifying modules to satisfy interfaces: A state transition system approach. *Distrib. Comput. 6*, 1, 39–63.

LAM, S. S., AND SHANKAR, A. U. 1990 A relational notation for state transition systems. *IEEE Trans. Softw. Eng. 16* (July), 755–775.

LAMPORT, L. 1991. The temporal logic of actions. *DEC SRC Rep. 57*, Palo Alto, Calif Revised 1991

LAMPORT, L. 1990. A theorem on atomicity in distributed algorithms. *Distrib. Comput. 4*, 2, 59–68.

LAMPORT, L. 1989. A simple approach to specifying concurrent systems *Commun. ACM 32*, 1 (Jan.), 32–45.

LAMPORT, L. 1987. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst. 5*, 1 (Feb), 1–11

LAMPORT, L. 1983a. What good is temporal logic. in *Proceedings of Information Processing 83*. IFIP, Arlington, Va.

LAMPORT, L. 1983b. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr.), 190–222.

LAMPORT, L. 1982. An assertional correctness proof of a distributed algorithm. *Sci Comput. Program. 2*, 3, 175–206.

LAMPORT, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. SE-3*, 2 (Feb.), 125–143.

LEHMAN, D., PNUELI, A., STAVI, J. 1981. Impartiality, justice, and fairness: The ethics of concurrent termination. In *Proceedings of the 8th ICALP* (Acre, Israel, July). Lecture Notes in Computer Science, vol. 115 Springer-Verlag, New York.

LYNCH, N. A., AND TUTTLE, M. R. 1987. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Vancouver, B C.). ACM, New York.

MANNA, Z., AND PNUELI, A. 1984. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4, 257–289.

MANNA, Z., AND PNUELI, A. 1992. *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, 1992.

MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N.J

MURPHY, S. L., AND SHANKAR, A. U. 1991 Connection management for the transport layer: Service specification and protocol construction. *IEEE Trans. Commun. 39* (Dec.), 1762–1775.

OWICKI, S., AND GRIES, D. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica 6*, 4, 319–340.

OWICKI, S., AND LAMPORT, L. 1982. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst. 4* (July), 455–495.

PETERSON, G. L. 1981. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12*, 3 (June) 1133–1145

PNUELI, A. 1984. In transition from global to modular temporal reasoning about programs. In *NATO ASI Series, Logics and Models of Concurrent Systems. vol. F13*. Springer-Verlag, Berlin, 123–144.

PNUELI, A. 1979. The temporal semantics of concurrent programs. In *Semantics of Concurrent Computation* Lecture Notes in Computer Science, vol. 70. Springer-Verlag, New York, 1–20.

PNEULI, A. 1977 The temporal logic of programs. In *Proceedings of the 18th ACM Symposium on the Foundations of Computer Science*. ACM, New York, 46–57.

SCHNEIDER, F. B., AND ANDREWS, G. R. 1986. Concepts for concurrent programming. In *Current Trends in Concurrency*. Lecture Notes in

Computer Science, vol. 224. Springer-Verlag, New York, 669–716.

SCHOONE, A A. 1987. Verification of connection-management protocols. Rijksuniversiteit Utrecht, RUU-CS-87-14

SHANKAR, A. U. 1989. Verified data transfer protocols with variable flow control. *ACM Trans Comput. Syst. 7*, 3 (Aug.), 281–316.

SHANKAR, A. U., AND LAM, S S. 1992 A stepwise refinement heuristic for protocol construction. *ACM Trans. Program Lang Syst 14*, 3 (July), 417–461

SHANKAR, A. U., AND LAM, S. S. 1987 Time-dependent distributed systems Proving safety, liveness and real-time properties *Distrib. Comput. 2*, 2, 61–79.

SHANKAR, A. U , AND LAM, S. S. 1983 An HDLC protocol specification and its verification using image protocols. *ACM Trans. Comput. Syst 1*, 4 (Nov.), 331–368.

SISTLA, A. P. 1984 Distributed algorithms for ensuring fair interprocess communications In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C , Aug.). ACM, New York.

TEL, G 1987 Assertional verification of a timer-based protocol. Rijksuniversiteit Utrecht, RUU-CS-87-15.

TEL, G., TAN, R B , AND VAN LEEUWEN, J. 1988 The derivation of graph marking algorithms from distributed termination detection protocols. *Sci Comput. Program. 10*, 2, 107–137.

Received January 1991, final revision accepted May 1993