A Logic for Concurrent Programming

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

April 12, 1994

Abstract

The UNITY-logic, a fragment of linear temporal logic, was introduced in [5]. In this paper, we describe several recent modifications to this logic. In particular, the operator **co** replaces *unless*, for expressing safety properties and, *transient* predicates form the basis for the progress properties. Our experience suggests that these modifications simplify both the theory and practice of this logic.

Overview

The UNITY-logic, a fragment of linear temporal logic, was introduced in [5]. Experience gained in applying this logic in practice has led us to modify some of its operators. In particular, we had adopted *unless* as the primary operator for expressing safety properties, for many years. We suggest a new operator, **co**, to take its place. Our experience, see, Staskauskas([42],[41],[43]), suggests that the simplicity of formal manipulations is at least as important as the expressive power of an operator. Theoretically, *unless* and **co** are equally expressive, while the latter has more pleasing derived rules that allow simpler manipulations. Similarly, we suggest *transient* predicate as the basis for a theory of progress, replacing *ensures* as the basic operator.

This paper is in two parts. We study *safety properties* in part I of this paper. Lamport gives the following informal meaning for safety: "bad things do not happen". Roughly, a safety property constrains the permitted actions–and, therefore, the permitted state changes–of a program. For instance, requiring that an integer x be nondecreasing in a program prevents any action of the program from decreasing x. Clearly, an action that causes no state change–a *skip*–implements any safety property. We will be particularly interested in several special kinds of safety properties, such as *invariant*, that a predicate remain *true* at all times during a program's execution; *stable predicate*, that a predicate remain *true* once it becomes *true*, and *constant*, that an expression never change value.

A safety property allows us to state that the program does no harm. Such properties impose an "upper bound" on the set of allowable execution sequences of the program. A trivial program such as *skip*, for instance, satisfies all the safety properties, because it includes a single execution sequence. Several formal aspects of program design and refinement are seriously affected by the absence of a "lower bound," that the program is required to have certain execution sequences. Thus, safety properties alone are insufficient as a basis of program design.

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658–219 and by the National Science Foundation Award CCR–9111912.

In part II, we study a class of properties known as *progress*. Progress properties state that "the program does some good." For instance, such a property may state that the value of x increases eventually. A nonprogramming example of a progress property is, "I press the switch and then the light is on." A safety property for this system might be "the light never comes on unless the switch is pressed." This safety property is conveniently implemented by smashing the light bulb. Conversely, the given progress property might be implemented by having a light that is permanently on. It is the interplay between the safety and progress properties that determines a nontrivial design.

The definitions of the logical operators and their derived rules constitute a small part of the paper. The bulk of the paper is devoted to treating examples from diverse areas in order to demonstrate the usefulness of the proposed operators. We also include a number of theoretical results about these operators that have been developed in the recent years.

Space does not permit a treatment of program composition. A logic for UNITY-program composition appers in [5, chapter 7]. The operators proposed in this paper can be treated similarly. In particular, the *union theorem* applies equally well to the new operators; see [34, chapters 5,6] for details.

We do not propose a new programming language. The proposed logic is applicable to any discrete action system, with a finite or infinite number of actions.

Part I
Safety

1 Introduction

The systems we consider are (discrete) *action systems*. Such a system consists of a number of actions each of which may, possibly, change the state of the system. A sequential program, expressed in a conventional language, is such a system; the state of the system is given by the values of the program variables and the program counter; each statement corresponds to an action that is effective only if the program counter has the appropriate value. At most one action may change any program state, because at most one action is "effective" for any given value of the program counter. A program with concurrently executing processes can also be regarded as an action system in which more than one action may be effective in a given program state. A particularly simple view of action systems is captured in UNITY[5] in which a program counter is shown explicitly (as a part of the system state). The results of this paper apply to *any* action system with an arbitrary—finite or infinite—number of actions. The only restriction that we impose is that there be a *skip* action in the system that may be applied in any program state; this action does not change the program state.

The primary operator for expressing safety properties is **co** (for *co*nstrains). This operator is designed to facilitate reasoning by induction on the number of computation steps. Most safety properties arising in practice are succinctly expressed using **co**. Additionally, **co** has simple manipulation rules which permit easy deduction of new properties from the given ones.

Outline of Part I

The operator **co** for a program is defined in terms of the actions of the program. Several special cases of **co**–such as invariant, stable, constant–arise frequently in practice; they are described in Section 2. The derived rules for **co**–the main ones being universal conjunctivity and universal disjunctivity–are given in Section 4. This section also describes the Substitution Axiom and the Elimination Theorem, which are essential devices for constructions of succinct proofs.

The major part of the paper, Section 5, is devoted to applying the theory. Safety properties for a number of small systems are formalized and manipulated. The examples are chosen from diverse application areas, in order to demonstrate the usefulness of the proposed operator. Section 6 contains a few theoretical results. A synopsis of this part appears in Section 7.

2 The meaning of co

The safety properties of a system are stated using the *constrains* (**co**) operator. We write p **co** q to denote that whenever p holds, q holds following the execution of the next action.

Given p **co** q it follows that $p \Rightarrow q$, because the action *skip* can be applied in any state satisfying p. Also, it follows that once p holds q continues to hold upto (and including) the point where p ceases to hold (if ever p ceases to hold). Also, if beyond a certain point p remains *true* forever, so does q (because if p holds then so does q). In any case, once p holds it continues to hold until $\neg p \wedge q$ holds.

We define

$$p \ \mathbf{co} \ q \ \triangleq \ \langle \forall\, t \ :: \ \{p\} \ t \ \{q\} \rangle$$

where t is an action of the sytem (and the quantification is over all actions). Here, $\{p\}\ t\ \{q\}$ stands for, if action t is started in any system state satisfying p and execution of t completes then q holds upon completion.

An equivalent formulation of **co** using Dijkstra's *wp*-calculus[10] is

$$p \ \mathbf{co} \ q \ \triangleq \ \langle \forall\, t \ :: \ p \Rightarrow wlp.t.q \rangle$$

This formulation allows us to establish the derived rules for **co**, in Section 3, by exploiting the properties of *wlp*.

Note on the binding powers of operators The operator **co** has lower binding power than all arithmetic and logical operators. thus,

$$p \wedge q \ \mathbf{co} \ r \wedge s$$

is to be interpreted as

$$(p \wedge q) \ \mathbf{co} \ (r \wedge s)$$ □

Mathematical modeling often consists of converting imprecise or ambiguous informal descriptions to formal descriptions. Such is the case with safety properties which are informally stated using, "... may remain *true as long as* ... ," "... can be changed *only if* ... ," or "... it is *never the case* that ... ." The formal descriptions using **co** are usually easy to construct. Experience helps; therefore, this paper includes a large number of examples of varying difficulties. Below, we show some typical informal descriptions and their formal counterparts.

Examples In the following examples, x, y are of type integer.

1. Once x is zero it remains zero until it becomes positive. Other ways of stating this are:

   x can become nonzero only by becoming positive or,
   x can not be decreased if it is zero.

   We observe that for any action if $x = 0$ is a precondition then either x remains zero or x becomes positive following the action, i.e.,

   $$x = 0 \ \ \mathbf{co} \ \ x \geq 0$$

2. x never decreases. One way to formalize this is to start with the equivalent: if x has a certain value, m, it continues to have that value until it exceeds m. This is identical to Example (1), except that 0 is now replaced by m. That is, for all m

   $$x = m \ \mathbf{co} \ x \geq m$$

   Here, m is a free variable; the above property is universally quantified over all integers m. Therefore, this property actually represents a set of properties, one property corresponding to each value of m. Another way to express that x never decreases is by,

   $$x \geq m \ \mathbf{co} \ x \geq m$$

   for all m. The formal correspondence between the two properties shown here can be established using the properties of **co**. The second property is in a particularly important form that will be studied in the next section.

3. x may only be incremented (i.e., increased by 1). This means that in any step, either x retains its old value, say m, or acquires the new value, $m + 1$. With free variable m,

   $$x = m \ \mathbf{co} \ x = m \ \vee \ x = m + 1$$

4. A philosopher may transit from thinking to hungry state. Using predicates t and h to represent that a (particular) philosopher is in thinking or hungry state, respectively, the above says that whenever t is falsified h is established. However, writing

   $$t \ \mathbf{co} \ h$$

   is incorrect, because t does not imply h. In such cases, we put the predicate in the lhs as a disjunct in the rhs to form

   $$t \ \mathbf{co} \ t \vee h$$

   The above says that whenever t is falsified $(t \vee h)$ holds; therefore, $\neg t \wedge (t \vee h)$ holds in such a state, which implies that h holds whenever t is falsified.

5. A message is received only if it has been sent. Such a statement is best treated by considering its contrapositive: Any message that is unsent remains unreceived. (A rule of thumb is to apply contrapositive whenever "only," "only if" or "provided that" appear in a description.) There are two possible interpretations—using *sent*, *received* to denote that a specific message has been sent, received, respectively—

   $$\neg sent \wedge \neg received \ \mathbf{co} \ \neg received$$
   $$\neg sent \wedge \neg received \ \mathbf{co} \ received \Rightarrow sent$$

   The second property allows for simultaneous truthification of *received* and *sent*—modeling a rendezvous-style communication—whereas the first property prohibits simultaneous transmission and delivery.

6. x, y never change simultaneously. This is an important property that holds in every asynchronous system where x, y are variables that can only be changed by different processes. Using free variables m, n

   $$x, y = m, n \ \mathbf{co} \ x = m \ \vee \ y = n$$

   That is, at least one of the variables is unchanged by every action. □

Relationship of co to other kinds of safety properties One aspect of a safety property is that it can be "implemented" by doing nothing. Therefore, every safety property holds in a system in which *skip* (i.e., do nothing) is the only action. For such a system, p **co** q holds whenever $p \Rightarrow q$, because

$$\begin{array}{l} \quad p \\ \Rightarrow \{p \ \Rightarrow\ q\ ,\ q = wlp.skip.q\} \\ \quad wlp.skip.q \end{array}$$

Several standard types of safety properties are simply expressed using **co**. We show some below; others appear in examples in this paper.

In [5], we had introduced p *unless* q, for arbitrary predicates p, q, to mean that once p holds it continues to hold as long as q does not; if p and q hold simultaneously nothing can be asserted about the next state. Formally, p *unless* q is, for all actions t,

$$\{p \ \wedge\ \neg q\} \ \ t \ \ \{p \ \vee\ q\}$$

Since $p \ \wedge\ \neg q \ \Rightarrow\ p \ \vee\ q$, we have

$$p \ \mathit{unless}\ q \ \equiv\ p \ \wedge\ \neg q \ \mathbf{co}\ p \ \vee\ q.$$

Conversely, $p\ \mathbf{co}\ q \ \equiv\ p \ \mathit{unless}\ \neg p \ \wedge\ q$.

To say that p can be falsified only if q is a precondition, we have, for all t, $\{p \ \wedge\ \neg q\}\ \ t\ \ \{p\}$,i.e,

$$p \ \wedge\ \neg q \ \mathbf{co}\ p$$

To express that once p holds it continues to hold until q holds, we have, for all t, $\{p\}\ \ t\ \ \{p \ \vee\ q\}$,i.e.,

$$p\ \mathbf{co}\ p \ \vee\ q$$

3 Special cases of co

3.1 Stable, Invariant, Constant

Several special cases of **co** appear frequently in practice; so, we have special names for these cases.

$$\begin{array}{ll} p \text{ stable} & \equiv p\ \mathbf{co}\ p \\ p \text{ invariant} & \equiv \textbf{initially}\ p \ \text{ and } p \ \text{ stable} \\ e \text{ constant} & \equiv \langle \forall\ k \ \ ::\ \ e = k \text{ stable} \rangle \\ & \quad \text{, where } e \text{ is an expression and } k \text{ is a free variable of the same type as } e. \end{array}$$

From above, p stable means that once p is *true* it remains *true* forever, because p **co** p is

$$\langle \forall\ s \ \ ::\ \ \{p\}\ \ s\ \ \{p\} \rangle$$

which means that no action falsifies p. Predicates *false* and *true* are stable in any program. Stable predicates are ubiquitous, for example: The system is deadlocked, a path exists between a sender and a receiver and the number of messages sent along a channel exceeds a certain value. We will see many instances of stable predicates in Section 5.

A predicate is *invariant* for a system if it holds initially and it remains *true* thereafter. Therefore, an invariant predicate is always *true* during a program execution. Predicate *true* is an invariant except in the pathological case where *false* holds initially. Invariance is a basic notion in program design. Note that we associate an invariant with a program, not with any point in the program text.

There is a subtle difference between a predicate being invariant and a predicate being "always-true." The difference is analogous to the distinction between "provability" and "truth" in logic. The following example is instructive.

Program *distinction*
  initially $x, y = 0, 0$
  assign $x, y := 0, x$
end

Here, $x = 0 \wedge y = 0$ is an invariant because it holds initially and

$$\{x = 0 \wedge y = 0\} \;\; x, y := 0, x \;\; \{x = 0 \wedge y = 0\}.$$

Since $x = 0 \wedge y = 0 \Rightarrow y = 0$, we can assert that $y = 0$ is always *true* during the program execution. However, $y = 0$ cannot be shown to be an invariant, because $y = 0$ cannot be shown stable according to our definition:

$$\{y = 0\} \;\; x, y := 0, x \;\; \{y = 0\}$$

does not hold. In Section 4.3, we show that a predicate that is "always-true" is also an invariant, using an extended notion of invariant that employs the "substitution axiom."

An expression, e, is constant if its value never changes during a program execution. Our definition says that once e has a value k, it will continue to have that same value. Similar to the above discussion for invariants, there are expressions whose values never change, but that cannot be proven constant according to our definition; for example, y cannot be shown constant in the above program though its value never changes from zero. We can use the substitution axiom, described in Section 4.3, to prove that such an expression is constant.

3.2 Fixed Point

For a given program, the fixed point predicate, FP, holds upon "termination"; that is, for a state in which FP holds, further execution of the program will not change state, and in a state in which FP does not hold, there is an execution of the program that causes it to change state.

The notion of program termination can be couched in terms of FP: a program is guaranteed to terminate iff it eventually reaches a state that satisfies FP; this is a progress property (discussed in Part II). The well-known phrase, "program is deadlock-free," means that $\neg FP$ is always *true*; then, it is always possible to change the program state.

We give the following formal characterization of FP in Section 6.3: It is the weakest predicate p such that $p \wedge b$ is stable for any b. We will show that there is a unique weakest predicate of this form. The following result is a direct consequence of this definition.

(Stability at Fixed Point) For any predicate b, $FP \wedge b$ stable. □

It is possible to compute FP syntactically from a UNITY program text: it is obtained by converting each assignment statement to an equality and conjoining all the equalities so obtained; see [5, chapter 3] for details.

4 Derived Rules

4.1 Basic Rules

The following rules for **co** are obtained easily from the facts about the predicate transformer, *wlp*. Here, p, q, p', q', r are arbitrary predicates.

- $$\mathit{false} \;\textbf{co}\; p$$

- $$p \ \mathbf{co} \ \mathit{true}$$

- (conjunction; disjunction) $$\frac{p \ \mathbf{co} \ q \ , \ p' \ \mathbf{co} \ q'}{\begin{array}{c} p \ \wedge \ p' \ \mathbf{co} \ q \ \wedge \ q' \\ p \ \vee \ p' \ \mathbf{co} \ q \ \vee \ q' \end{array}}$$

The conjunction and disjunction rules follow from the conjunctivity and monotonicity properties of wlp[10] and of logical implication. These rules generalize in the obvious manner to any set—finite or infinite—of **co**-properties, because wlp and logical implication are universally conjunctive and universally disjunctive. As corollaries of conjunction and disjunction—conjoining r **co** true and disjoining false **co** r, respectively, to p **co** q—we obtain

- (lhs strengthening) $$\frac{p \ \mathbf{co} \ q}{p \ \wedge \ r \ \mathbf{co} \ q}$$

- (rhs weakening) $$\frac{p \ \mathbf{co} \ q}{p \ \mathbf{co} \ q \ \vee \ r}$$

Also, **co** is transitive $$\frac{p \ \mathbf{co} \ q \ , \ q \ \mathbf{co} \ r}{p \ \mathbf{co} \ r}$$

This is because $q \ \Rightarrow \ r$ from q **co** r; then rhs of p **co** q can be weakened to p **co** r. Transitivity, however, seems to be of little value because any application of transitivity could be replaced by lhs strengthening—strengthen q to p—or rhs weakening—weaken q to r.

The operator **co** is a form of temporal implication. It shares many of the properties of logical implication, such as the ones shown above. However, it is not reflexive (i.e., p **co** p does not always hold) nor are we allowed to deduce a contrapositive ($\neg q$ **co** $\neg p$ cannot be deduced from p **co** q).

Proof Format for co Since the lhs of a **co**-property can be strengthened and its rhs can be weakened, we can use **co** as an operator in a proof presented as a weakening chain, as shown below to establish p **co** t.

$$\begin{array}{ll} & p \\ \Rightarrow & \{\text{justification}\} \\ & q \\ \Rightarrow & \vdots \\ & r \\ \mathbf{co} & \{\text{justification}\} \\ & s \\ \Rightarrow & \vdots \\ \Rightarrow & \{\text{justification}\} \\ & t \end{array}$$

This is not the only proof-format that we employ. In many proofs we deal with several **co**-properties over which conjunctions and disjunctions are employed; in such cases, we write one property per line and associate justifications with each line.

4.2 Rules for the special cases

The following rules follow from the conjunction and disjunction rules given earlier.

(stable conjunction, stable disjunction)

$$\frac{p \;\textbf{co}\; q \;,\; r \;\text{ stable}}{\begin{array}{l} p \wedge r \;\textbf{co}\; q \wedge r \\ p \vee r \;\textbf{co}\; q \vee r \end{array}}$$

A special case of the above is,

$$\frac{p \;\text{ stable} \;,\; q \;\text{ stable}}{\begin{array}{l} p \wedge q \;\text{ stable} \\ p \vee q \;\text{ stable} \end{array}}$$

Similarly,

$$\frac{p \;\text{ invariant} \;,\; q \;\text{ invariant}}{p \wedge q \;\text{ invariant}}$$

(constant formation)

Any expression built out of constants and free variables is a constant.

4.3 Substitution Axiom

Substitution axiom allows us to deduce properties that we cannot deduce directly from the definition. It states: *An invariant may be replaced by true, and vice versa, in any property of a program.* For instance, given that p **co** q and that J invariant, we can conclude

$p \wedge J \;\textbf{co}\; q$, $p \;\textbf{co}\; q \wedge J$, $p \wedge J \;\textbf{co}\; q \wedge J$, $p \vee \neg J \;\textbf{co}\; q \wedge J$, etc.

In particular, given that p is invariant and $p \Rightarrow q$, we can show q invariant, as follows.

p invariant	, given
$p \wedge q$ invariant	, $p \equiv p \wedge q$, since $p \Rightarrow q$
q invariant	, replace p by *true* using the substitution axiom

In Program *distinction* of Section 3.1, we showed a predicate that is "always-true" though we could not show it to be invariant. In particular, we could show $x = 0 \wedge y = 0$ invariant, though $y = 0$ could not be proven invariant. Using the argument given above (since $x = 0 \wedge y = 0 \Rightarrow y = 0$), we can now claim, $y = 0$ invariant. Thus, the substitution axiom allows us to remove the distinction between "always-true" and invariant.

Another consequence of the substitution axiom is that a theorem and an invariant have the same status; an invariant can be treated as a theorem, and a theorem, of course, is an invariant. Therefore, we often write simply J, rather than J invariant.

Note: For compositions of progams, a generalization of the substitution axiom applies[34, Chapter 6]. $\square$

Rationale for the substitution axiom The following rationale for the substitution axiom is due to Knapp[21]. In the definition of p **co** q, we interpreted $\{p\}\ t\ \{q\}$ to mean that if action t is started in any state satisfying p then q holds in the state upon completion of t, assuming t completes. What is "any state"? We restrict our state space to the *reachable* states: A *reachable state* is a state (an assignment of values to variables) that may arise during a computation; i.e., either it satisfies the initial condition, or it is a state that may result by applying an action to a reachable state. It follows that every reachable state satisfies every invariant. In fact, the set of reachable states is the set of states that satisfy the strongest invariant (we show the existence of the strongest invariant in Section 6.2).

We interpret $\{p\}\ t\ \{q\}$, for a given program that includes action t, to mean that if action t is started in any reachable state satisfying p then the resulting (reachable) state satisfies q. We may establish $\{p\}\ t\ \{q\}$ by proving, for any invariant J,

$$p \ \wedge\ J \ \Rightarrow\ wlp.t.q$$

This is because the set of states satisfying $p \ \wedge\ J$ *includes* all reachable states satisfying p (since a reachable state satisfies any invariant). In particular, we can use the types of variables, or even *true*, as the invariant, J. Even though the notion of invariant is defined using **co**, and the definition of **co** seems to require the notion of invariant, there is no circularity if we start with a known invariant, such as *true* or the variable-types. We could then deduce other invariants which could be applied in deducing other **co**-properties and invariants.

4.4 Elimination Theorem

Free variables are essential to our theory. Free variables can be introduced in the lhs by strengthening and in the rhs by weakening, e.g., from p **co** q we can deduce, for program variable x and free variable m,

$$p \ \wedge\ x = m \ \mathbf{co}\ q \quad \text{and,}$$
$$p\ \mathbf{co}\ q \ \vee\ x \neq m$$

Free variables can be eliminated by taking conjunctions or disjunctions. We give a useful theorem below for eliminations of free variables by employing disjunction.

Let p be any predicate, x be a program variable and m be a free variable (of the same type as x). Denote by $p[x\ :=\ m]$ the predicate obtained by replacing all occurrences of x by m in p. Now, if p names no program variable other than x then $p[x\ :=\ m]$ has no program variable, and, hence, it is a constant. In particular, $p[x\ :=\ m]$ is stable. Observe that,

$$p = \langle \exists\ m\ :\ p[x\ :=\ m]\ :\ x = m \rangle$$

Theorem(Elimination Theorem)

$$\frac{\begin{array}{c} x = m\ \mathbf{co}\ q\ ,\ \text{where } m \text{ is free} \\ p \text{ does not name } m \text{ nor any program variable other than } x \end{array}}{p\ \mathbf{co}\ \langle \exists\ m\ ::\ p[x\ :=\ m]\ \wedge\ q \rangle}$$

Proof:

$x = m\ \mathbf{co}\ q$	, premise
$p[x\ :=\ m]\ \wedge\ x = m\ \mathbf{co}\ p[x\ :=\ m]\ \wedge\ q$	, stable conjunction with $p[x\ :=\ m]$
$\langle \exists\ m\ ::\ p[x\ :=\ m]\ \wedge\ x = m \rangle\ \mathbf{co}\ \langle \exists\ m\ ::\ p[x\ :=\ m]\ \wedge\ q \rangle$	, disjunction over all m
$p\ \mathbf{co}\ \langle \exists\ m\ ::\ p[x\ :=\ m]\ \wedge\ q \rangle$	, simplifying the lhs $\square$

The elimination theorem can be applied where x is a list of variables (and m is a list of free variables). In the following example, we apply the theorem with $x = (u, v)$.

Example: Suppose

$$u, v = m, n \ \mathbf{co}\ u, v = m, n \ \vee\ (m > n \ \wedge\ u, v = m-1, n)$$

We will show that $u \geq v$ stable. Using $p \equiv u \geq v$ in the elimination theorem we have

$$\begin{array}{ll} & u \geq v \\ \mathbf{co} & \{\text{elimination theorem}\} \\ & \langle \exists\, m, n \ :: \ (m \geq n \ \wedge\ u, v = m, n) \ \vee\ (m \geq n \ \wedge\ m > n \ \wedge\ u, v = m-1, n) \rangle \\ \Rightarrow & \{\text{weakening each disjunct}\} \\ & \langle \exists\, m, n \ :: \ u \geq v \ \vee\ u \geq v \rangle \\ \Rightarrow & \{\text{simplifying}\} \\ & u \geq v \end{array}$$

□

5 Applications

We apply our theory to several small problems. In each case, we convert an informal description to a set of **co**-properties, apply some of the manipulation rules given in the preceding section and interpret the derived results. These exercises suggest that the proposed theory is preferable to intuitive reasoning, not merely for avoiding errors, but also for its simplicity and conciseness.

5.1 Common Meeting Time

This problem has been discussed in Chandy and Misra[5, Section 1.4]. The purpose of this example is to explore a family of design alternatives by considering the safety properties common to all members of the family.

It is required to find the earliest meeting time acceptable to every member in a group. Time is nonnegative and real-valued. To simplify notation, assume that there are only two persons, F and G, in the group. Associated with F, G are functions, f, g, respectively, where each function maps nonnegative reals to nonnegative reals (i.e., times to times). For any real t, $f(t)$ is the earliest time at or after t when F can meet; similarly $g(t)$ is defined. Time t is acceptable to F iff $f(t) = t$. Time t is a *common meeting time* iff it is acceptable to both F and G, i.e., $f(t) = t \ \wedge\ g(t) = t$. The goal is to design an algorithm that computes the earliest (i.e., smallest nonnegative) common meeting time, provided one exists.

Several algorithms and their implementations on various architectures have been described in [5]. Here, we define the essential safety properties common to *all* these algorithms.

First, we have to make certain assumptions about f, g so that the earliest meeting time can be computed effectively. Our verbal description suggests that $t \leq f(t)$, but we won't require this property (see, however, the discussion at the end of this section). We postulate that f, g be monotonic, i.e., for all nonnegative real m, n

$$\begin{array}{l} m \leq n \ \Rightarrow\ f(m) \leq f(n) \\ m \leq n \ \Rightarrow\ g(m) \leq g(n) \end{array} \qquad \text{(CMT1)}$$

We adopt the following strategy in computing the earliest meeting time: we have a variable t, nonnegative and real, whose value never exceeds the earliest common meeting time, and t is increased, eventually, if it is not a common meeting time. The latter property is discussed in Section 13.1. Here, we consider the safety aspect of the problem, given by the first requirement on t. A strategy for implementing this requirement is to set t to 0, initially. The rule for modifying t is: if t's value is m before an action then it does not exceed both $f(m)$ and $g(m)$ after the action. It is not obvious that this strategy prevents t from exceeding the earliest common meeting time; we prove this fact below.

The formal description of the strategy is as follows.

initially $t = 0$ (CMT2)
$t = m$ **co** $t \le \max(f(m), g(m))$ (CMT3)

Let $com(n)$ denote that n is a common meeting time, i.e., $f(n) = n \ \wedge \ g(n) = n$. We prove, from CMT1-CMT3, that t exceeds no common meeting time, i.e., for any n,

$$com(n) \ \Rightarrow \ t \le n \qquad \text{(CMT4)}$$

To prove that (CMT4) is an invariant, note that it holds initially for any nonnegative real n, using (CMT2). The remaining proof obligation is, rewriting (CMT4),

$$(\neg com(n) \ \vee \ t \le n) \text{ stable} \qquad \text{(CMT5)}$$

which we proceed to prove.

$$\begin{array}{ll}
 & \neg com(n) \ \vee \ t \le n \\
\textbf{co} & \quad \{ \text{ elimination theorem on (CMT3) where } p \text{ is } \neg com(n) \ \vee \ t \le n \ \} \\
 & \langle \exists\, m \ :: (\neg\, com(n) \ \vee \ m \le n) \ \wedge \ t \le \max(f(m), g(m)) \rangle \\
\equiv & \quad \{ \text{ write the disjunction as } \neg\, com(n) \ \vee \ (com(n) \wedge m \le n) \text{ and expand } \} \\
 & \langle \exists\, m \ :: [\neg\, com(n) \ \wedge \ t \le \max(f(m), g(m))] \ \vee [com(n) \ \wedge \ m \le n \wedge \ t \le \max(f(m), g(m))] \rangle \\
\Rightarrow & \quad \{ \text{ the first disjunct implies } \neg com(n) \text{, the second disjunct implies (see below) } t \le n \ \} \\
 & \langle \exists\, m \ :: (\neg\, com(n) \ \vee \ t \le n) \rangle \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
 & \neg\, com(n) \ \vee \ t \le n
\end{array}$$

This establishes (CMT5). Now we prove the result claimed in the above proof:

$$com(n) \ \wedge \ m \le n \ \wedge \ t \le \ \max(f(m), g(m)) \ \Rightarrow \ t \le n$$

$$\begin{array}{ll}
 & t \\
\le & \quad \{ \text{ from the antecedent } \} \\
 & \max(f(m), g(m)) \\
\le & \quad \{ \ m \le n \Rightarrow [f(m) \le f(n) \wedge g(m) \le g(n)] \text{, from (CMT1) } \} \\
 & \max(f(n), g(n)) \\
= & \quad \{ \text{ from definition, } com(n) \equiv [f(n) = n \ \wedge \ g(n) = n] \ \} \\
 & n
\end{array}$$

The strategy given by (CMT3) is quite general. It subsumes the following strategies.

$t = m$ **co** $t = m$
$t = m$ **co** $t = m \ \vee \ t = \max(f(m), g(m))$
$t = m$ **co** $t = m \ \vee \ t = f(m) \ \vee \ t = g(m)$

In each of the above properties, the rhs is stronger than that of (CMT3); hence, (CMT3)—and, consequently, (CMT5)—can be derived from each of the above properties. A weak safety property like (CMT3) is preferable for initial design explorations because it constrains the allowable actions only minimally.

Each of the following programs, P0–P3 (in which the initial condition, $t = 0$, is not shown) implements (CMT3). In a program its actions are separated by $[\!]$. The program executes its actions repeatedly, choosing any action, nondeterministically, for execution in each step.

$$\begin{array}{lll}
\text{P0} & :: & t := t \qquad \{\text{does nothing useful}\} \\
\text{P1} & :: & t := f(t) \ [\!] \ t := g(t) \\
\text{P2} & :: & t := \max(f(t), g(t)) \\
\text{P3} & :: & t := t + 1 \quad \text{if} \quad t + 1 \le \max(f(t), g(t))
\end{array}$$

A useful strengthening of CMT3 is to require that t be nondecreasing, i.e.,

$$t = m \textbf{ co } m \le t \le \max(f(m), g(m)) \qquad \text{(CMT3}')$$

This property is easily implemented (by programs P1,P2, for instance) if we know that the functions f, g are ascending:

$$n \le f(n) \ \wedge \ n \le g(n) \qquad \text{(CMT0)}$$

The reader can prove (by applying elimination theorem to CMT3$'$) that

$$t = n \wedge \ com(n) \ \ \text{stable}$$

i.e., t does not change once its value equals a common meeting time. Interestingly, neither (CMT0) nor (CMT3$'$) is required in deriving (CMT4). Another fact about f (and g) suggested by the verbal description is that $f(f(n)) = f(n)$. This fact is not required in any of our derivations.

5.2 A Small Concurrent Program: Token Ring

The method advocated in the last section—describing an algorithm by its properties—is most profitably applied to concurrent algorithms. Here, we consider a simple mutual exclusion problem.

A set of processes are connected in a ring where the message transmissions take place over the edges of the ring; thus a process may communicate only with its left or right neighbor. It is given that at most one process may transmit at any time. A technique to meet this requirement is to have a single token circulate in the ring and allow only the token-holder to transmit. The following is an abstraction of this solution.

A process is in one of three states: waiting to transmit (also called *hungry*), transmitting (also called *eating*) and neither of the above (also called *thinking*). The terms hungry, eating, thinking are taken from the dining philosophers problem which is a standard abstraction in resource allocation. An eating process can transit only to thinking (see TR1, below), a thinking process can transit only to hungry (see TR2) and a hungry process can transit only to eating (TR3); a hungry process remains hungry as long as it does not hold the token (TR4); the token is relinquished only by a noneating process (TR5). We would like to prove that there is at most one eating process—i.e., at most one transmitting process—in the system. An intuitive proof is obvious: There is at most one token in the system and an eating process holds the token. Unfortunately, the proof is far from obvious; for instance, the rule for relinquishing the token, as stated above, is so vague that it allows a hungry process to release the token when it transits to eating. Furthermore, initially the eating process, if any, has to hold the token (TR0). Such details, which are often ignored in informal descriptions, cause endless problems.

Notation: We write $h_i/e_i/t_i$ to denote that process i is hungry / eating / thinking. These predicates are mutually exclusive and $h_i \ \vee \ t_i \ \vee \ e_i \equiv true$. The position of the token is in the variable p, i.e., $p = i$ (as in TR5) denotes that process i holds the token. □

In the following, i ranges over all processes.

$$\textbf{initially } e_i \ \Rightarrow \ p = i \qquad \text{(TR0)}$$
$$e_i \textbf{ co } e_i \ \vee \ t_i \qquad \text{(TR1)}$$
$$t_i \textbf{ co } t_i \ \vee \ h_i \qquad \text{(TR2)}$$
$$h_i \textbf{ co } h_i \ \vee \ e_i \qquad \text{(TR3)}$$
$$h_i \ \wedge \ p \ne i \textbf{ co } h_i \qquad \text{(TR4)}$$
$$p = i \textbf{ co } p = i \ \vee \ \neg e_i \qquad \text{(TR5)}$$

The properties, TR0–TR5, specify only the safety aspects of the system. In particular, the protocol for transmitting the token (which is sent to the neighbor in a specific direction in the ring) is completely ignored in this specification. The complete protocol is specified in Section 13.2. Here, we show that the partial specification is sufficient for establishing mutual exclusion, i.e., that there is at most one eating process.

The assertion that there is at most one eating process, can be expressed by the invariant

$$e_i = e_j \;\Rightarrow\; i = j$$

for all i, j. This result is shown by proving that an eating process holds the token, i.e., for any i, $e_i \;\Rightarrow\; p = i$. Then $(e_i \;\wedge\; e_j) \;\Rightarrow\; (p = i \;\wedge\; p = j) \;\Rightarrow\; (i = j)$. We show the invariant $e_i \;\Rightarrow\; p = i$, for any i. Initially, the result follows from TR0. The remaining proof obligation is to show that $(e_i \;\Rightarrow\; p = i)$ is stable, for any i.

$t_i \;\vee\; h_i \;\vee\; p = i$ **co** $t_i \;\vee\; h_i \;\vee\; p = i \;\vee\; \neg e_i$
, disjunction of (TR2, TR4, TR5)
$e_i \;\Rightarrow\; p = i$ stable , replacing $t_i \;\vee\; h_i$ by $\neg e_i$ and rewriting

Observe that (TR1, TR3) are unnecessary for the proof of mutual exclusion. This proof is concise, partly because it does not mimic the usual commonsense reasoning. The proof demonstrates the effectiveness of the disjunction rule.

5.3 From Program Texts to Properties

Sometimes portions of a system are specified by program fragments whereas other portions are specified by properties. It may be useful, therefore, to convert a program text to properties. This conversion can be carried out automatically, especially for programs in which each statement corresponds to an action. We sketch, below, how this is done.

Let x denote a group of program variables. Consider all the statements that can change one or more variables in x. Let these statements be of the form

$$x \;:=\; e_1 \quad [\!]\; \ldots \; x \;:=\; e_i \; \ldots \; [\!] \; \ldots$$

Let $e[x \;:=\; m]$ be the expression obtained by replacing x by m in e. Then the current value of x, say m, can be changed to $e_1[x \;:=\; m]$ by executing the statement $x \;:=\; e_1$, or, in general, to $e_i[x \;:=\; m]$ by executing $x \;:=\; e_i$. Since x can be changed only by the above statements we get

$$x = m \;\; \textbf{co} \;\; x = m \;\vee\; x = e_1[x \;:=\; m] \;\vee\; \ldots \; x = e_i[x \;:=\; m] \;\vee\; \ldots$$

Note that e_i may name variables outside x.

A common syntactic variation is a guarded command of the form

$g_i \;\longrightarrow\; x \;:=\; e_i$
or, $x \;:=\; e_i$ if g_i (in UNITY).

Given a set of statements of this form that can modify x, we obtain

$$\begin{array}{l} x = m \;\textbf{co}\; x = m \vee (g_1[x \;:=\; m] \;\wedge\; x = e_1[x \;:=\; m]) \;\vee\; \ldots \\ \qquad\qquad\qquad (g_i[x \;:=\; m] \;\wedge\; x = e_i[x \;:=\; m]) \;\vee\; \ldots \end{array}$$

In systems consisting of multiple processes in which a variable is changed by at most one process, the variables are naturally partitioned among the processes, each variable *belonging* to the process that may change it (the variables that are never changed are constants). For each process, we obtain a **co**-property in the above manner over the variables that it may change. For example, given below is a system that represents two processes, one of which may modify x and the other y (though the latter can read x):

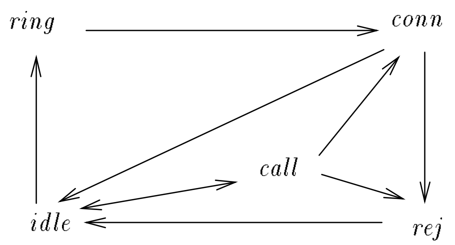


Figure 1: A State Transition Diagram for a Telephone

$x \; := \; x+1 \; [\!] \; y \; := \; y+1 \quad \text{if} \quad x > y$

we obtain for the two parts, respectively,

$x = m$ **co** $x = m \;\vee\; x = m+1$ and,
$y = n$ **co** $y = n \;\vee\; (x > n \;\wedge\; y = n+1)$

In order to prove a safety property for this action system—say, $x \geq y$ stable—we could either start with the program text (and show that each action preserves the property) or we could start with the safety property of the system obtained as above and then deduce the result. Whenever we have a mixture of programming fragments and properties, the latter strategy is preferable since we can then work using a single notation.

5.4 Finite State Systems

Finite state descriptions are often used for specifications of communication protocols and discrete control systems. Here, we show how the state transitions can be described in our theory. As an example of a finite state system we consider an extremely simplified version of a telephone system; see Staskauskas[42] for a more realistic version.

We focus our attention on a single telephone. We postulate it to have the following states:

idle: the handset is on-hook and not ringing

ring: the handset is on-hook and ringing

call: the handset is off-hook and a number is being dialed

conn: the telephone is engaged in a call

rej: the telephone is receiving a busy signal or the other party has disconnected

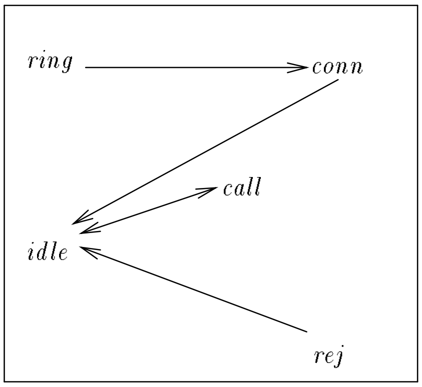
Our model is so simple that we can't even distinguish in the *conn* state if this telephone initiated a call or received a call.

A possible state transition diagram is shown in Figure 1. Most of the transitions are self-explanatory; for instance, the transition from *conn* to *idle* is taken when the user hangs up and the transition from *call* to *rej* is effected by the telephone switch giving a busy signal to the caller.

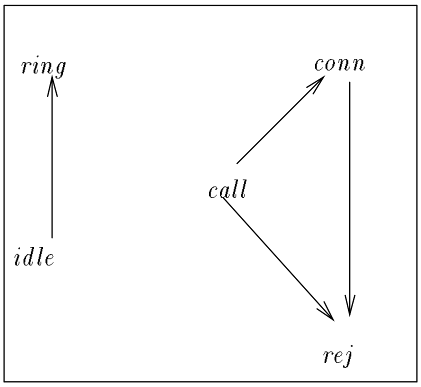
The safety properties of a finite state system can be obtained automatically as follows. There is one safety property for each state. For a state x that has possible transitions to states y and z, the corresponding property is

$$\text{state} = x \;\; \textbf{co} \;\; \text{state} = x \;\vee\; \text{state} = y \;\vee\; \text{state} = z$$

which expresses the fact that from the current state, x, a transition can effect a state change to y or z only. The safety properties obtained from the diagram in Figure 1 are as follows (in these properties we write, for instance, *idle* to denote state=*idle*).



2.2a: Transitions Made by the User



2.2b: Transitions Made by the Switch

Figure 2: Partitioning the Transitions of Figure 1 between the User and the Switch

$$\begin{array}{lll} idle & \mathbf{co} & idle \vee ring \vee call \\ call & \mathbf{co} & call \vee idle \vee conn \vee rej \\ conn & \mathbf{co} & conn \vee idle \vee rej \\ rej & \mathbf{co} & rej \vee idle \\ ring & \mathbf{co} & ring \vee conn \end{array}$$

The transitions in Figure 1 are of two types—those made by the user and those made by the telephone switch. For instance, the transition from *idle* to *ring* is made by the telephone switch (the user can never make a telephone ring) whereas all transitions to *idle* are made by the user. Figure 2 shows two transition diagrams, one for the user and one for the switch. Clearly, properties can be written down for each of these figures as shown above.

In these diagrams, we have not shown the conditions under which a transition takes place. For instance, the switch causes a transition from *call* to *conn* only if a connection has been made to the dialed number, and the transition from *conn* to *idle* by the user is made as a result of the user hanging up. Sometimes, the condition cannot be expressed as a finite state property. To see this, let *num* be a variable in which the number dialed by the user is stored (in this, extremely trivial, description we ignore the fact that digits are stored one by one into *num*; instead, we assume that the entire number is stored into *num* in a single atomic action). The variable *num* can also take on a special value, $\perp$, denoting that no number has been stored in *num*.

A precondition for transition of a user u from *call* to *conn* is that

its *num* differs from $\perp$, and
its called number, v, is in *ring* state

A postcondition of this transition is that u, v are both in *conn* states or that u hangs up. This fact can be expressed by (where we prefix states by the user identity to avoid confusion), for all users u, v $(u \neq \perp, v \neq \perp)$

$$u.call \ \wedge \ u.num = v \ \wedge \ v.ring \ \mathbf{co}$$
$$(u.call \ \wedge \ u.num = v \ \wedge \ v.ring) \ \vee u.idle \ \vee (u.conn \ \wedge \ v.conn \ \wedge \ u.num = v)$$

This is, however, a very coarse property; we cannot even state that a user is connected to exactly one party in the *conn* state. By suitably introducing other variables (for instance, *u.party* to denote the party to which u is connected in the *conn* state; then, $u.conn \ \wedge \ u.num \neq \perp \ \Rightarrow \ u.party = u.num$) we can obtain a more refined specification.

It becomes clear as we add more and more details to a specification that the finite state diagram provides a gross description of the flow of control. It is sometimes appealing to work with a diagram because a visual check may suffice to deduce a property. However, many of these systems have aspects that cannot be captured by a finite number of states.

5.5 Auxiliary Variables

In stating and verifying properties of programs, it is sometimes necessary to introduce *auxiliary* variables, variables whose values at any point in the computation depend only on the history of the other variable values. For instance, for u, an integer valued variable, define an auxiliary variable v whose value is the number of times that u's value has changed during the course of a computation. Now, v can be introduced by defining it to be initially zero, and modifying the program text to increment v whenever u is changed. The problem with this approach is that the relationship between u, v as stated above is lost; it has to be gleaned from the program text. Since we now have a logical operator to relate values of various variables over the course of a computation, we can define v directly, as follows; here, m, n are integer-valued free variables.

$$\mathbf{initially}\ v = 0$$
$$u, v = m, n \ \ \mathbf{co} \ u, v = m, n \ \vee \ (u \neq m \ \wedge \ v = n + 1)$$

This property asserts that every change in u is accompanied by an increment in the value of v and as long as u does not change, v does not change either. It is now a simple matter to prove various facts about v, such as that v is nondecreasing.

A useful auxiliary variable is the sequence of distinct values written into a given variable; for a variable x let $\widehat{x}$ be this sequence. Then, for m, n of the appropriate type

$$\mathbf{initially}\ \widehat{x} = \langle\langle x \rangle\rangle \quad \{\langle\langle x \rangle\rangle \text{ is the sequence consisting of the value of } x\}$$
$$x, \widehat{x} = m, n \ \ \mathbf{co} \ \ x, \widehat{x} = m, n \ \vee \ (x \neq m \ \wedge \ \widehat{x} = n; x)$$

where ";" is the sequence concatenation operator.

Finally, we show that for a semaphore s, $s - nv + np$ is constant where np, nv are the numbers of successful "P" and "V" operations, respectively. We define np, nv as follows; in the following m, n, r are natural valued free variables.

$$\mathbf{initially}\ np, nv = 0, 0$$
$$s, nv = m, n \ \mathbf{co} \ n \leq nv \leq n + 1 \ \wedge \ (s = m + 1 \ \equiv \ nv = n + 1)$$
$$s, np = m, r \ \mathbf{co} \ r \leq np \leq r + 1 \ \wedge \ (s = m - 1 \ \equiv \ np = r + 1)$$

A semaphore value may change (increase or decrease) by 1 in any step:

$$s = m \ \mathbf{co} \ m - 1 \le s \le m + 1$$

Conjoining the above three **co**-properties

$$\begin{array}{l} s, nv, np = m, n, r \ \mathbf{co} \\ \qquad m - 1 \le s \le m + 1 \ \wedge \ n \le nv \le n + 1 \ \wedge \\ \qquad r \le np \le r + 1 \ \wedge \ (s = m + 1 \ \equiv \ nv = n + 1) \wedge \ (s = m - 1 \ \equiv \ np = r + 1) \end{array}$$

Weaken the rhs to $s - nv + np = m - n + r$ to get

$$s, nv, np = m, n, r \ \mathbf{co} \ s - nv + np = m - n + r.$$

Applying the elimination theorem,

$$s - nv + np \quad \text{constant}$$

5.6 Deadlock

A trivial form of deadlock—captured by the "after you, after you" paradigm —arises in a cycle of processes when each process waits for its left neighbor. It is intuitively obvious that this system state will persist forever. However, a formal proof or even a rigorous argument based on this verbal description is messy. Most such proofs are by contradiction: If this system state does not persist forever then there is a process that stops waiting; let x be the first process to stop waiting; x can stop waiting only if the process for which it has been waiting is not waiting; therefore, there is a process that stopped waiting earlier than x, contradicting our choice of x as the first process to stop waiting. Such informal arguments, though appealing, are neither precise about the assumptions they make—for instance, can x, y, where x is waiting for y, stop waiting simultaneously—nor are they concise.

Process x waits for y means that whenever both x, y are waiting both will continue waiting until y stops waiting. This informal description is quite ambiguous; it admits at least two interpretations as shown below. Denoting by xw that process x is waiting (and similarly, for yw) x waits for y means

$$xw \ \wedge \ yw \ \ \mathbf{co} \ \ xw \qquad \text{(strong wait) or,} \qquad \text{(D1)}$$
$$xw \ \wedge \ yw \ \ \mathbf{co} \ \ yw \ \Rightarrow \ xw \qquad \text{(weak wait)} \qquad \text{(D2)}$$

In strong wait, both processes cannot stop waiting simultaneously; in weak wait they could. By suitably weakening the rhs of strong wait we can obtain weak wait. Therefore, strong wait is indeed stronger than weak wait.

Now, it is trivial to show that two processes, x, y, waiting strongly for each other are deadlocked. Because, we have then,

$$\begin{array}{l} xw \ \wedge \ yw \ \ \mathbf{co} \ \ xw \\ yw \ \wedge \ xw \ \ \mathbf{co} \ \ yw \end{array}$$

Applying conjunction we get

$$xw \ \wedge \ yw \ \ \mathbf{co} \ \ xw \ \wedge \ yw$$

We conclude from the above that $xw \ \wedge \ yw$ is stable; hence $xw \ \wedge \ yw$, once $true$, persists forever. Our proof avoids explicit arguments about time, or arguments based on contradiction. The essence of the proof is an induction on the number of computation steps, which is captured in the **co**-properties. Now, we prove the result for any finite cycle of processes, but we will do more. We show that if one process

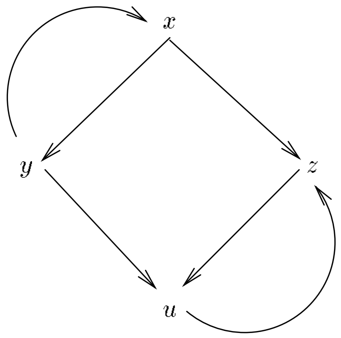


Figure 3: A Knot of Waiting Processes

in a cycle strong-waits and the remaining processes weak-wait then a deadlock arises when all processes are waiting.

Let the processes in the cycle be indexed 0 through N, $0 < N$. Let w_i denote that process i is waiting. Suppose process 0 strong-waits for process N; from (D1) this is expressed by

$$w_0 \ \wedge \ w_N \ \ \mathbf{co} \ \ w_0 \tag{D3}$$

In the remaining processes, process $(i+1)$ weak-waits for process i, $0 \leq i < N$. This is expressed, as in (D2), by

$$\langle \forall \ i \ : \ 0 \leq i < N \ : \ w_{i+1} \ \wedge \ w_i \ \ \mathbf{co} \ \ w_i \ \Rightarrow \ w_{i+1} \rangle \tag{D4}$$

This completes the mathematical modeling of the problem. Conjunction of (D4), over all i, $0 \leq i < N$, yields

$$\langle \forall \ i \ : \ 0 \leq i < N \ : \ w_{i+1} \ \wedge \ w_i \rangle \ \ \mathbf{co} \ \ \langle \forall \ i \ : \ 0 \leq i < N \ : \ w_i \ \Rightarrow \ w_{i+1} \rangle$$

Conjunction of the above and (D3) yields, after simplifying the lhs,

$$\langle \forall \ i \ : \ 0 \leq i \leq N \ : \ w_i \rangle \ \ \mathbf{co} \ \ w_0 \ \wedge \ \langle \forall \ i \ : \ 0 \leq i < N \ : \ w_i \ \Rightarrow \ w_{i+1} \rangle.$$

The rhs, using the induction principle, implies $\langle \forall \ i \ : \ 0 \leq i \leq N \ : \ w_i \rangle$. Therefore,

$$\langle \forall \ i \ : \ 0 \leq i \leq N \ : \ w_i \rangle \ \ \text{stable}$$

Contrast this argument with a typical informal argument of the following kind. Consider the state of the system where all processes are waiting. If any process $(i+1)$, $i \geq 0$, stops waiting subsequently then process i must have stopped waiting then or before $(i+1)$. Applying the same argument to i, $i-1$, etc., we conclude that process 0 stops waiting at or before any other process stops waiting. However, process N has to stop waiting strictly prior to process 0. Applying the same kind of argument from process N down to $(i+1)$, we conclude that $(i+1)$ stops waiting before the time we postulated for it to stop waiting. Our experience suggests that a temporal argument, particularly if it employs proof by contradiction, can be replaced by a succinct formal argument using the **co**-properties.

A more involved example of mutual waiting arises when a process can wait for any one of a set of processes. This is the typical situation where processes hold resources and a waiting process can proceed only after receiving *any one* of the resources for which it is waiting. Figure 3 shows an example in which four processes, x, y, z, u, are deadlocked. The outgoing arrows from x (to y and z) denote that x is waiting for one of y and z. The waiting condition for x—using xw, yw, zw to denote that x, y, z are waiting, respectively—is that $xw \ \wedge \ yw \ \wedge \ zw$ **co** xw.

Unlike the previous case, the existence of a cycle in the graph does not guarantee deadlock. A nonempty set of nodes, S, in a directed graph forms a *knot* if every outgoing edge from a node in S is to a node in S. Figure 3 has knots $\{x, y, z, u\}$ and $\{z, u\}$. We show that when all processes in a knot S are waiting in the above sense, they are deadlocked. As before, for a process i in S, w_i denotes that process i is waiting. Let process i have outgoing edges to nodes in S_i, $S_i \subseteq S$. Then, the waiting condition for process i is given by

$$w_i \wedge \langle \forall j : j \in S_i : w_j \rangle \ \mathbf{co} \ w_i$$

Taking conjunction over all i in S

$$\langle \forall i : i \in S : w_i \wedge \langle \forall j : j \in S_i : w_j \rangle \ \rangle \ \mathbf{co} \ \langle \forall i : i \in S : w_i \rangle$$

Simplifying the lhs, using $S_i \subseteq S$ (since S is a knot),

$$\langle \forall i : i \in S : w_i \rangle \text{ stable}$$

Observe that the type of waiting is captured by the **co**-property for each i and the structure of the knot is exploited in simplifying the conjunction. An informal proof, we suspect, will be much longer.

5.7 Axiomatization of a Communication Network

In this example, we develop axioms to describe a message communicating network. These axioms are based on Misra[30, Section 6.1]. See Chandy and Misra[6] for a more general treatment.

To keep matters simple, suppose that we have two processes, A, B, that communicate via two one way channels. The number of messages sent by a process is at least as large as the number received by the other, and both quantities are nonnegative. Furthermore, the number of messages sent and received along each channel is nondecreasing. A channel is *empty* if the number of messages sent equals the number of messages received along that channel. The state of a process is either *idle* or *active*. An idle process remains idle until it receives a message. Only active processes may send messages.

Our main interest is in specifying the properties of the system, formally. We will also show that when both processes are idle and both channels are empty then the system is terminated, in the sense that no further messages will be sent nor received nor will any of the processes become active.

In describing a system, we first have to decide on the variables whose values determine the state of the system. The choice is not always clear. In this case, for instance, we will not explicitly introduce a channel state. Instead, we introduce sA, sB to denote the number of messages sent by A, B and rA, rB for the number of messages received by A, B, respectively. The channel from A to B, for example, is *empty* if $sA = rB$. That the number of messages sent by a process is at least as large as the number received by the other, and both quantities are nonnegative, is given by

$$sA \geq rB \geq 0 \text{ invariant}$$
$$sB \geq rA \geq 0 \text{ invariant} \qquad \text{(CN1)}$$

The number of messages sent and received are nondecreasing. Using free variables m, n (of type natural)

$$sA \geq m \text{ stable}, \quad sB \geq m \text{ stable},$$
$$rA \geq n \text{ stable}, \quad rB \geq n \text{ stable} \qquad \text{(CN2)}$$

To express the facts dealing with the states of a process we introduce the boolean variable qA denoting that A is idle; similarly, qB.

The following properties say that each process remains idle as long as it receives no message.

$$qA \wedge rA = m \ \mathbf{co} \ rA = m \Rightarrow qA$$
$$qB \wedge rB = m \ \mathbf{co} \ rB = m \Rightarrow qB \qquad \text{(CN3)}$$

Note that it is not specified if an idle process becomes active upon receiving a message; our specification allows it to be in either state. That only active processes may send messages is best understood as, an idle process does not send messages, i.e.,

$$qA \ \wedge \ sA = n \ \textbf{co} \ sA = n$$
$$qB \ \wedge \ sB = n \ \textbf{co} \ sB = n \qquad \text{(CN4)}$$

A Digression: There is a subtle point in our formulation of CN4. We prohibit an idle process from receiving a message, becoming active and then sending a message, all in one step. If we wished to allow this possibility we would have written CN4 analogously to CN3:

$$qA \ \wedge \ sA = n \ \textbf{co} \ sA = n \ \Rightarrow \ qA$$
$$qB \ \wedge \ sB = n \ \textbf{co} \ sB = n \ \Rightarrow \ qB$$

Observe that with this specification we can no longer prove termination: The specification does not say that an idle process has to receive a message *strictly* prior to becoming nonidle and sending a message. Therefore, in a state where both processes are idle and both channels are empty, a next state is possible where both processes are idle, both channels are empty *and* one additional message has been sent and received along every channel. □

This completes our axiomatization. Our axioms capture only some aspects of the system behavior. In particular, we have no guarantee that all messages are eventually delivered—a progress property—or that messages are delivered in the same order in which they are sent (a safety property).

The state in which both processes are *idle* and both channels are *empty* is given by

$$(qA \ \wedge \ qB) \ \wedge \ sA, sB = rB, rA$$

We can show that the above predicate is stable. But that is not enough; it leaves open the possibility, for instance, that the values of sA, rB could change while preserving this predicate. Therefore, we will prove that once the above predicate holds none of the variables will change value. Since $sA, sB = rB, rA$ from the predicate, it is sufficient to show that rA, rB do not change value, i.e., for free m, n we can show that

$$(qA \ \wedge \ qB) \ \wedge \ sA, sB = rB, rA \ \wedge \ rA, rB = m, n \text{ stable} \qquad \text{(CN5)}$$

Proof of CN5 is left to the reader.

5.8 Coordinated Attack

The following version of a vicious synchronization problem has received considerable attention in the literature. We show how it can be dealt with, relatively simply, by using our theory. The following description of the problem is from Gray[14].

> "Two divisions of an army are camped on two hilltops overlooking a common valley. In the valley awaits the enemy. It is clear that if both divisions attack the enemy simultaneously they will win the battle, whereas if only one division attacks it will be defeated. The divisions do not initially have plans for launching an attack on the enemy, and the commanding general of the first division wishes to coordinate a simultaneous attack (at some time the next day). Neither general will decide to attack unless he is sure that the other will attack with him. The generals can only communicate by means of a messenger. Normally, it takes the messenger one hour to get from one encampment to the other. However, it is possible that he will get lost in the dark or, worse yet, be captured by the enemy. Fortunately, on this particular night, everything goes smoothly. How long will it take them to coordinate an attack?"

The original arguments to show that there is no protocol to achieve coordinated attack were long and cumbersome. Halpern and Moses[15] gave the first convincing proof of this fact based on the notions of *knowledge* and *common knowledge*. We give a very brief outline of that theory as it pertains to this problem.

For a process x and predicate p, we denote "x knows p" by $x\ k\ p$. The value of the predicate $x\ k\ p$ is a function of the state of x; hence $x\ k\ p$ does not change as long as x performs no action (an action is either a local computation, sending a message or receiving a message). For a predicate p, predicate cp denotes that p is *common knowledge* (for some group of processes). We have, for any process x in the group

$$cp \ \equiv \ x\ k\ (cp)$$

That is, p is common knowledge iff all processes know it to be common knowledge.

The coordinated attack problem requires us to establish common knowledge of the attack time. Halpern and Moses[15] showed that if each atomic action affects the state of only one process, as is customarily the case in asynchronous message communicating systems, then common knowledge cannot be gained. Later, Chandy and Misra[4] showed that under the same assumptions, common knowledge can neither be gained nor lost; that is, a previously unplanned attack cannot be coordinated nor can a planned coordinated attack be called off. This result holds under the much weaker assumption that no action can affect the states of *all* processes (though any proper subset of processes could be affected). We prove this more general result below.

Theorem: In a group having more than one process where no atomic action can affect the states of *all* processes, common knowledge can neither be gained nor lost; that is, for any predicate p, cp constant.

Proof: An atomic action does not affect the states—hence the knowledge predicates—of *all* processes simultaneously. Therefore, in a group having more than one process, knowledge (or, ignorance) of at least one process about a predicate, q, is unchanged by any action. We write this assertion—where x_i denotes the i^{th} process, i is quantified over all processes in the group and b_is are free boolean variables—

$$\langle \forall\ i \ :: \ (x_i\ k\ q) = b_i \rangle \ \mathbf{co} \ \langle \exists\ i \ :: \ (x_i\ k\ q) = b_i \rangle$$

Instantiate q by cp. Then,

$x_i\ k\ q = x_i\ k\ (cp) = cp$,from the definition of cp
$\langle \forall\ i \ :: \ cp = b_i \rangle \ \mathbf{co} \ \langle \exists\ i \ :: \ cp = b_i \rangle$,replacing $x_i\ k\ q$ by cp
cp **co** cp ,setting all b_is to $true$. Note i is quantified over a nonempty group
$\neg cp$ **co** $\neg cp$,similarly, setting all b_is to $false$
cp constant ,from the above two using the definition of constant

5.9 Dynamic Graphs

This example, based on Chandy and Misra[5, Chapter 12] and Misra[29], illustrates how we express and deduce facts about dynamic data structures. In this example, the data structure is a finite directed graph which can be changed by the following operation: All edges incident on a node may be directed towards that node in one (atomic) step.

A node that has no outgoing edge is called *bottom*. We are required to show that no path is ever created to a nonbottom node, that is if there is no path from node u to node v initially then there is no path from u to v at any point in the computation unless v is a bottom node at that point.

Let us first give a typical proof that draws upon the well known theorems and terminology of graph theory. Suppose that there is no path from u to v before a step. Call all the edges of the graph "old" at this time. Following the step that redirects all incident edges towards a node, w, call the newly redirected edges to w "new." Suppose there is a path from u to v following the step and v is nonbottom. Not all edges on the path are "old" because then there would have been a path before the step. Since some new edge is on this path, node w is on this path. We show that node w is not on the path, thus leading to a contradiction: Node w is different from v because v is assumed to be a nonbottom node, and w is a bottom node after the step; node w is not an intermediate node on the path nor is $w = u$, because w has no outgoing edge.

Our formal proof avoids the temporal argument and the proof by contradiction. However, the proof has to make explicit the notion of a path. In the following proof, u, v, w range over nodes of the graph. We use the following predicates:

$u.\perp$: u is a bottom node
$u\ R^k\ v,\ k > 0$: there is a path of length k from u to v
$u\ R\ v$: there is a path from u to v

Now, $u\ R^1\ v$ denotes that there is an edge from u to v; define $u\ R^{k+1}\ v$, $k \geq 1$, inductively as follows.

$$u\ R^{k+1}\ v \ \equiv\ \langle \exists\ w\ ::\ u\ R^1\ w\ \wedge\ w\ R^k\ v\rangle \qquad \text{(GR1)}$$

Then,

$$u\ R\ v \ \equiv\ \langle \exists\ k\ :\ k \geq 1\ :\ u\ R^k\ v\rangle \text{ and} \qquad \text{(GR2)}$$
$$u.\perp \ \equiv\ \langle \forall\ v\ :\ \neg u\ R\ v\rangle \qquad \text{(GR3)}$$

The operation of changing the graph is given by the following **co**-property. It states that no edge from u to v is created as long as v remains nonbottom.

$$\neg u\ R^1\ v\ \ \mathbf{co}\ \ \neg u\ R^1\ v\ \vee\ v.\perp \qquad \text{(GR4)}$$

We prove that no path from u to v is created as long as v remains a nonbottom node.

Theorem: $\neg u\ R\ v\ \ \mathbf{co}\ \ \neg u\ R\ v\ \vee\ v.\perp$

Proof: We show that for all k, $k \geq 1$, and all u, v

$$\neg u\ R^k\ v\ \ \mathbf{co}\ \ \neg u\ R^k\ v\ \vee\ v.\perp \qquad \text{(GR5)}$$

Taking conjunction of (GR5) over all k, $k \geq 1$, concludes the proof of the theorem.

Proof of (GR5): by induction on k.

$k = 1$: the result follows from (GR4)
$k + 1$: show $\neg u\ R^{k+1}\ v\ \ \mathbf{co}\ \ \neg u\ R^{k+1}\ v\ \vee\ v.\perp$

From (GR4), using w for v, we have

$$\neg u\ R^1\ w\ \ \mathbf{co}\ \ \neg u\ R^1\ w\ \vee\ w.\perp$$

Using induction hypothesis (GR5) with w in place of u we have

$$\neg w\ R^k\ v\ \ \mathbf{co}\ \ \neg w\ R^k\ v\ \vee\ v.\perp$$

Disjunction of the above two gives us

$$\neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \ \mathbf{co}\ \ \neg u\ R^1\ w\ \vee\ \neg w\ R^k\ v\ \vee\ w.\perp\ \vee v.\perp$$

From (GR3) $w.\bot \Rightarrow \neg w \; R \; v$ and from (GR2) $\neg w \; R \; v \;\Rightarrow\; \neg w \; R^k \; v$. Hence, $\neg w \; R^k \; v \;\vee\; w.\bot$ in the rhs of the above can be replaced by $\neg w \; R^k \; v$.

$$\neg u \; R^1 \; w \;\vee\; \neg w \; R^k \; v \;\; \mathbf{co} \;\; \neg u \; R^1 \; w \;\vee\; \neg w \; R^k \; v \;\vee\; v.\bot$$

Taking conjunction of the above over all w,

$$\langle \forall \; w \;\; :: \;\; \neg u \; R^1 \; w \;\vee\; \neg w \; R^k \; v \rangle \;\; \mathbf{co} \;\; \langle \forall \; w \;\; :: \;\; \neg u \; R^1 \; w \;\vee\; \neg w \; R^k \; v \rangle \;\vee\; v.\bot$$

Using (GR1), replace $\langle \forall \; w \;\; :: \;\; \neg u \; R^1 \; w \;\vee\; \neg w \; R^k \; v \rangle$ by $\neg u \; R^{k+1} \; v$

$$\neg u \; R^{k+1} \; v \; \mathbf{co} \; \neg u \; R^{k+1} \; v \;\vee\; v.\bot$$ □

We note, as a corollary, that once a node u is outside any cycle, it remains outside cycles. In particular, if the graph is initially acyclic, it stays acyclic. Node u is outside any cycle if $\neg u \; R \; u$ holds.

Corollary: $\neg u \; R \; u$ stable.

Proof:

$\neg u \; R \; v \; \mathbf{co} \; \neg u \; R \; v \;\vee\; v.\bot$	, theorem
$\neg u \; R \; u \; \mathbf{co} \; \neg u \; R \; u \;\vee\; u.\bot$	, replacing v by u in the above
$\neg u \; R \; u \; \mathbf{co} \; \neg u \; R \; u \;\vee\; \neg u \; R \; u$	, weakening rhs using (GR3) for $u.\bot \Rightarrow \neg u \; R \; u$
$\neg u \; R \; u$ stable	, predicate calculus on the rhs

6 Theoretical Results

There are several results about **co** that are used in formulating other concepts, such as the strongest invariant or FP; however, these results are rarely used in dealing with specific applications.

6.1 Strongest rhs; weakest lhs

In a given program, for any predicate p there is a strongest q such that p **co** q; similarly, for any q there is a weakest p such that p **co** q. We only prove the first result; the proof of the second one is similar.

Theorem (strongest rhs): For any p there exists a q such that

- $p \; \mathbf{co} \; q$ and,
- $(\forall \; r \; : \; p \; \mathbf{co} \; r \; : \; q \;\Rightarrow\; r)$

Proof: We give an explicit description of q; it is the conjunction of the rhs of all **co**-properties in which p appears in the lhs:

$$q \;\equiv\; \langle \wedge \; b \; : \; p \; \mathbf{co} \; b \; : \; b \rangle$$

Since **co** is universally conjunctive, we have p **co** q. For any r such that p **co** r, we get $q \;\Rightarrow\; r$ from the definition. □

6.2 Strongest Invariant

Every program has a (unique) strongest invariant. This is proved by defining the strongest invariant to be, simply, the conjunction of all invariants. We give a longer alternative proof which provides a "constructive" procedure for obtaining the strongest invariant. The essence of the procedure is to start with the initial condition—call it p_0, and obtain a sequence of p_is where p_i **co** p_{i+1} and p_{i+1} is the strongest rhs for p_i. The disjunction of all the p_is is the strongest invariant.

Lemma: Let $p_0, \ldots, p_i, \ldots$ be an infinite sequence where, for all i, $i \geq 0$,

$$p_i \ \textbf{co} \ p_{i+1}$$

Then, $\langle \exists\, i \ :: \ p_i \rangle$ stable.

Proof: Taking the disjunction over all the **co**-properties, p_i **co** p_{i+1}, we get

$$\langle \exists\, i \ : \ i \geq 0 \ : \ p_i \rangle \ \textbf{co} \ \langle \exists\, i \ : \ i > 0 \ : \ p_i \rangle$$

Weakening the rhs by p_0, we obtain the result. □

Theorem (strongest invariant): Let p_0 be the initial condition. Let p_i **co** p_{i+1}, for all i, $i \geq 0$, where p_{i+1} is the strongest rhs for p_i. Then, $\langle \exists\, i \ :: \ p_i \rangle$ is the strongest invariant.

Proof: Let $P = \langle \exists\, i \ :: \ p_i \rangle$. From the previous lemma, P is stable. Furthermore, since $p_0 \ \Rightarrow \ P$, initially P holds. Therefore, P is an invariant.

To show that P is the strongest invariant, we show that for any invariant J, $P \ \Rightarrow \ J$, i.e., $\langle \forall\, i \ :: \ p_i \ \Rightarrow \ J \rangle$. The proof is by induction on i.

$p_0 \ \Rightarrow \ J$: initially J holds. Since p_0 is the initial condition, $p_0 \ \Rightarrow \ J$.

Assume $p_i \ \Rightarrow \ J$ and show $p_{i+1} \ \Rightarrow \ J$:

p_i **co** p_{i+1} , given
$p_i \ \wedge \ J$ **co** $p_{i+1} \ \wedge \ J$, stable conjunction with J
p_i **co** $p_{i+1} \ \wedge \ J$, $p_i \ \wedge \ J \ \equiv \ p_i$ since $p_i \ \Rightarrow \ J$
$p_{i+1} \ \Rightarrow \ p_{i+1} \ \wedge \ J$, p_{i+1} is the strongest rhs in any **co**-property whose lhs is p_i
$p_{i+1} \ \Rightarrow \ J$, from the above using predicate calculus □

The strongest invariant includes the initial condition, p_0, as a disjunct. Therefore, if p_0 is not identically *false* then neither is the strongest invariant. Conversely, if p_0 is *false* then so is p_1—because, *false* is stable—and hence, the strongest invariant is *false*. We have often used the term "reachable states" informally in this paper. Formally, a state is *reachable* iff it satisfies the strongest invariant.

6.3 Fixed Point

The predicate FP characterizes the set of states that do not change as a result of program execution. We can define FP using **co**. Observe that any subset of states satisfying FP is stable, i.e., for any predicate b, $FP \ \wedge \ b$ stable. This, however, does not identify a unique FP. In particular, *false* $\wedge \ b$ is stable, for any b. We define FP to be the *weakest* predicate p such that $p \ \wedge \ b$ is stable, for all b. It can be shown that FP has the following closed form.

$$FP \ \equiv \ \langle \exists\, p \ : \ \langle \forall\, b \ :: \ p \ \wedge \ b \ \ \text{stable} \rangle \ : \ p \rangle$$

As is the case for the strongest invariant, it is difficult to compute FP using the above formulation; it is preferable to obtain it by syntactic manipulations of the program text.

A program is "deadlock-free" iff FP is always-*false*, i.e., $\neg FP$ is always-*true*. Using the substitution axiom, then $\neg FP$ is invariant.

7 Synopsis

This section contains an assessment of the strengths and weaknesses of **co**. Earlier, we had gained considerable experience in writing and manipulating *unless* properties[5, 42], a predecessor of **co**. There is ample reason to believe that **co** would be at least as powerful for expressing the usual kinds of safety properties for reactive systems. The manipulation rules for **co** are simple and effective. The examples in Section 5—particularly, common meeting time (5.1), deadlock (5.6) and coordinated attack (5.8)—were handled by extremely concise proofs; it is difficult to see how intuitive reasoning could be any cheaper! Though our examples often used UNITY-style programs, the theory is applicable to any action system.

Temporal logic is an elegant extension of classical logic that includes the temporal operators, $\Box$ and $\Diamond$ (read as *always* and *eventually*). The primary operator for expressing the safety properties is $\Box$; for example, the temporal formula $p \Rightarrow \Box p$ denotes that once p is true it is always true, i.e., p is stable. Temporal logic has a well developed theory and it has been applied in a variety of problems in computer science, see Manna and Pnueli[27]. There are many properties that are easily expressible in linear temporal logic but are difficult to express using **co**. Consider: Once x exceeds 5 it remains positive. In temporal logic, this is simply

$$x > 5 \;\Rightarrow\; \Box\,(x > 0).$$

Such a property cannot be written directly using **co** because examining a system state, say, $x = 2$, yields little clue about the value of x in the next state. Whether $x > 0$ in some state may depend on the history of the computation; thus, for any state we have to know if x has exceeded 5 in the past. Introducing an auxiliary boolean variable b that is true iff x has ever exceeded 5, we can express this assertion:

initially $b \equiv x > 5$,
$x > 5 \;\Rightarrow\; b$,
$b \;\wedge\; x > 0$ stable

Note that the proof of, $x > 5 \;\Rightarrow\; \Box\,(x > 0)$ in temporal logic would require the introduction of a predicate analogous to b.

Event predicates[22] and TLA[25] have proved useful for describing safety properties. An example of an event predicate is $x' \geq x$ which says that the value of x after any action—this value is denoted by x'—is at least the value of x before the action—the value before the action is, simply, denoted by x. This formalism is attractive because the inference rules are simply those of predicate calculus. Lamport combines event predicates, temporal operators and quantification (over variables and actions) to obtain a powerful logic, called TLA. He advocates using the same logic for representing both a system and its properties, thus simplifying the proofs of implementations.

A disadvantage of our theory (compared to event predicates and TLA) is that we often have to introduce free variables in stating the properties. For instance, "x is nondecreasing" is written as, $x \geq m$ stable with a free variable m, whereas $x' \geq x$ is an event predicate that expresses the same fact. In the other parts of our theory, free variables are essential (to say, for instance, that x increases eventually). Therefore, we do not feel especially guilty in introducing free variables for stating safety properties.

8 Bibliography

The notions of pre- and post-conditions are from Floyd[12] and Hoare[16]. The *wp*-calculus is due to Dijkstra[10]; see Dijkstra and Scholten[11] for an elaborate treatment of predicate transformers and their applications in program semantics. Lamport[23] was the first to coin the terms *safety* and *liveness*; formal definitions of these terms are in Alpern and Schneider[1]. Our earlier theory of safety[5], using the *unless* operator, was inspired by temporal logic. The present treatment tries to overcome some of the pragmatic difficulties of using *unless*. There are a number of papers on the substitution axiom, in particular by Sanders[39], Knapp[21] and Misra[33]; the interpretation given in this paper is from Knapp[21]. A clear example of the distinction between "invariant" and "always-true" is in van Gasteren and Tel[45]. The notion of the strongest invariant has been around for a long time; see Lamport[24] and Sanders[39], in particular. Section 5.5 on auxiliary variables follows the treatment in Misra[31]. For completeness of UNITY logic see Jutla, Knapp and Rao[17] and Cohen[7].

There are many other formalisms that are effective for expressing safety properties. Notable among these are temporal logic[27], event predicates[22] and TLA[25], which have already been discussed. Rao[38] has proposed logical operators for expressing *probabilistic* safety properties. Carruth[2] has generalized the **co** operator in order to state and prove real time properties. His generalization introduces a parameter for time into **co** . He shows that the derived rules for **co** given in this paper also apply for his generalizations.

Part II
Progress

9 Introduction

Safety properties, discussed in Part I, allow us to state that "the program does no harm." A progress property may be regarded as a performance guarantee. Such guarantees typically include time bounds: The light comes on within 10 ms of pressing the switch or a car travelling at 60 mph stops within 130 feet after the brakes are jammed. Absolute performance guarantees, though desirable, are hard to implement, because such guarantees depend on the speed of the underlying machine or the network, the scheduling strategy or even the load on the system, factors that are outside our control during program design. A useful abstraction employed in complexity theory is to specify the rate of growth of the computation time as a function of the input size. This abstraction, effectively, ignores speed-ups by constant factors. An even coarser abstraction is to classify the rate of growth as being polynomial or non-polynomial. Unfortunately, we don't yet have a theory to provide such performance guarantees for the asynchronous systems that we consider here; we don't even have the appropriate parameters by which to measure a problem size. So, we abstract further by eliminating the notion of absolute time. We will state and prove properties of the form: Once predicate p holds, eventually predicate q will hold in the system. For the lighting problem, p might be "the switch has been pressed" and q might be "the light is on." The time duration between the occurrences of p and q is left unspecified. We will develop the logic to state and deduce such properties. Realize that this is a far cry from the absolute performance guarantees that we seek. However, it is a useful first step in establishing the performance of the system. Once we have such a guarantee we may attempt to deduce the performance empirically or by using analytic modelling.

Outline of Part II

In Section 10, we describe several notions of fairness that are essential for studying progress properties in asynchronous systems. In Section 11, we introduce *transient* predicates; a transient predicate is guaranteed to be falsified eventually, under the given fairness assumption. Our primary progress operator, *leads-to*, is introduced in Section 12. Most progress specifications and deductions are done with *leads-to*. So, we give a variety of manipulation rules and show several examples of *leads-to* in Section 13. Certain theoretical issues are taken up in Section 14. A synopsis appears in Section 15.

10 Fairness

The need for fairness and the various flavors of it can be explained by considering the "guarded-command" program shown below. The guards for the statements α, β, γ are $true, true, x \neq y$, respectively. In this program, x, y, z are integers.

Program Fairness

$$\begin{array}{lllll} & \alpha & :: & x := x+1 & \\ [\!] & \beta & :: & y := y+1 & \\ [\!] & \gamma & :: & z := z+1 & \text{if} \quad x \neq y \end{array}$$

end {**Fairness**}

A fairness condition defines the order in which the statements, α, β, γ are executed. We study the following kinds of fairness: minimal progress, weak fairness and strong fairness.

10.1 Minimal Progress

Under this notion of fairness, a non*skip* action whose guard is *true* in the current state is chosen (arbitrarily) and executed; this step is repeated until all guards are *false*.

For the above example, minimal progress forbids executing γ forever starting in a state where $x = y$, thus preserving the same state. We can assert for this program that $x + y + z$ will eventually increase (and hence, it will increase without bound). This is because all guards are never *false*—α, β have *true* as their guards—and any change in the system state increases $x + y + z$. However, neither x, y nor z is guaranteed to increase because, for instance, β (or α or γ) might be executed indefinitely. Similarly, no guarantee can be made that $x + y$ will increase (γ might be executed forever once $x \neq y$); there is no guarantee about eventual increase of $x + z$ or $y + z$ either.

10.2 Weak Fairness

Under this notion of fairness, in an infinite execution each statement is executed infinitely often. Executing a statement in a state where its guard is *false* causes no state change.

This fairness condition guarantees that different processes in a multiprocess program will be individually allowed to proceed. We can imagine that the statements representing the various processes constitute the program under consideration. For example, α might belong to one process and β, γ to another. If α is chosen forever for execution (as in minimal progress), we have, effectively, permanently blocked the second process; weak fairness prevents such executions.

For the example program we can assert that x (and y) will increase without bound because each execution of α (or β) will cause x (or y) to increase. We cannot assert that z will increase. For instance, consider the following execution starting in state $x, y = 0, 0$: Execute α, β, γ in this order and repeat the sequence forever. Whenever γ is executed $x = y$ and hence, z is never increased.

Weak fairness is sometimes expressed as: If the guard of a statement remains continuously *true* then the statement is eventually executed (in a state where the guard is *true*). This formulation is identical to our earlier formulation.

10.3 Strong Fairness

Execution of a statement is strongly fair if in every execution sequence, whenever the guard of the statement is infinitely often *true*, the statement is executed (in a state where the guard is *true*) infinitely often.

For the above example, if all three statements are executed in a strongly fair manner then x, y, z will all increase indefinitely. It is easy to see this result for x, y; for z, note that $x \neq y$ is infinitely often *true* since x, y are incremented asynchronously; therefore, z will be incremented infinitely often.

10.4 Which is the Fairest One?

We may mix different kinds of fairness for the same program; we may group the statements and require different forms of fairness for each group. For instance, if we require minimal progress for $\{\alpha, \beta\}$ and strong fairness for $\{\gamma\}$ in the program **Fairness**, then we can assert that $x + y$ and z will increase indefinitely, but neither x nor y can be asserted to increase. If $\{\alpha, \beta\}$ have the weak fairness restriction and $\{\gamma\}$ the strong fairness then all three—x, y, z—will grow indefinitely.

Grouping statements introduces the possibility of hierarchy; subgroups within a group may have different fairness conditions attached to them. Though these possibilities are interesting theoretically, we do not pursue them here. In fact, we develop the theory only for minimal progress and weak fairness and, partially, for strong fairness.

A traditional sequential program permits no choice in statement executions. Therefore, the only pertinent notion of fairness in this case is minimal progress. The significant progress property is termination; it can be stated as "starting in any state that satisfies the initial condition eventually a state is reached that satisfies FP," (FP is the fixedpoint predicate; a state satisfying FP is a terminal state). Termination can be proved by displaying a function whose value decreases eventually as a result of program execution; if the function assumes values from a well-founded set its value cannot decrease forever, and hence, termination is guaranteed. Minimal progress is also useful in concurrent programs for proving the "absence of deadlock"; a typical example is if there is a hungry philosopher (in a dining philosophers problem) then some philosopher will eat.

Minimal progress is not sufficient to guarantee "absence of individual starvation." Even though some philosopher may eat, and eating is performed infinitely often, a particular philosopher may stay hungry forever (and starve). In the program **Fairness** studied in this section, the system as a whole makes progress by increasing $x + y + z$; no guarantees could be made about the individual variables—x, y or z—though, under minimal progress.

Weak fairness meets most of our criteria for a useful notion of fairness. It is powerful enough so that starvation-free solutions can be designed, and it is simple enough that a reasonably effective theory for reasoning about it can be developed. It has nice compositional properties[5, Chapter 7].

A typical example of the application of strong fairness is in implementing a "strong" semaphore. It is required that if the semaphore value exceeds 0 infinitely often then every process waiting for the semaphore be granted the semaphore. Under the weak fairness requirement a waiting process may never be granted the semaphore (because the guard—that the semaphore value exceeds 0—is not continuously *true*, but it is *true* infinitely often.) We take up this example in Section 5.5.

Observe that in a program if all actions are strongly fair then they are also weakly fair, and a weakly fair program meets the minimal progress condition. Thus, any progress property that can be deduced under some fairness assumptions is valid under stronger fairness assumptions.

11 Transient Predicate

A predicate is *transient* if it is guaranteed to be falsified by execution of a single (atomic) action. The formal definition depends on the form of fairness assumed for program execution. This is the only point in our theory where a definition of an operator depends on the form of fairness. Other progress operators are defined using transient predicates; their definitions and derived rules are independent of the underlying fairness. Thus, our progress proofs are largely shielded from having to argue about specific fairness properties of programs. We consider different notions of fairness and define transient predicates for each case.

There is a variety of language features for process synchronization, mutual exclusion, process communication, etc. It is possible, though cumbersome, to define transient predicates for programs that include these programming language features. For simplicity, we will restrict ourselves to action systems; such systems can be represented by a UNITY program[5]. We make two assumptions about such systems.

- There is at least one action. Recall that *skip* is included as an action in every system and, hence, this requirement is always met. (We do not show *skip*, explicitly, in the programs we write.)

- Each action terminates, i.e., if the action is started in any state where it is enabled, it completes in finite time. It is easy to check termination for simple actions, such as the ones that can be represented by assignment of values to variables. For more intricate actions, for instance, where an action is an entire program, the methods of this paper have to be applied recursively to first prove terminations of the individual actions and then deduce progress properties of the system.

For a terminating action s, we have the Law of Excluded Miracle[11]

$$\frac{\{p\} \;\; s \;\; \{\mathit{false}\}}{\neg p}$$

i.e., the postcondition of an action is *false* only if the precondition is *false*. Using the substitution axiom, this can be interpreted as "the resulting state of an action is unreachable only if the action is started in an unreachable state."

11.1 Minimal Progress

We consider a program of guarded commands, where every non*skip* command is of the form

$$\langle [\!]\; i \;::\; g_i \;\rightarrow\; s_i \rangle$$

or, $\langle [\!]\; i \;::\; s_i \;\;\text{if}\;\; g_i \rangle$ (in UNITY)

Predicate p is **transient** if,

(1) whenever p holds some command has a *true* guard: $p \;\Rightarrow\; \langle \exists\, i \;::\; g_i \rangle$

and, (2) executing *any* command in a state where p holds falsifies p:
$$\langle \forall\, i \;::\; \{p \wedge g_i\} \;\; s_i \;\; \{\neg p\} \rangle$$

Thus, a transient predicate, if *true* before a step, is *false* after the step. This definition may seem overly restrictive; can we not require that a predicate be falsified in a finite number of steps? The operator *leads-to* will be used to express such facts. Also, our requirement (2), that *every* enabled non*skip* action falsify p is essential; without such a requirement a possible execution may consist only of actions that never falsify p.

Example: We consider the running example from Section 10, which we reproduce below.

Program Fairness
$\alpha \;::\; x \;:=\; x+1$
$[\!]\; \beta \;::\; y \;:=\; y+1$
$[\!]\; \gamma \;::\; z \;:=\; z+1 \qquad \text{if } x \neq y$
end {**Fairness**}

First, we establish that for any integer k,

$x+y+z=k$ transient

Following are the proof obligations and they are easily established. For any integer k,

1. $x+y+z=k \;\Rightarrow\; \mathit{true}$

2. $\{x+y+z=k\}$ x := $x+1$ $\{x+y+z \neq k\}$
$\{x+y+z=k\}$ y := $y+1$ $\{x+y+z \neq k\}$
$\{x+y+z=k \wedge x \neq y\}$ z := $z+1$ $\{x+y+z \neq k\}$

We can similarly show that $x = y$ transient. Next, we attempt proving, for any k, $x = k$ transient. We know, from operational arguments, that this property does not hold. The corresponding proof obligations that cannot be established are

$$\{x=k\} \;\; y \;:=\; y+1 \;\; \{x \neq k\}$$
$$\{x=k \;\wedge\; x \neq y\} \;\; z \;:=\; z+1 \;\; \{x \neq k\}$$

We leave it to the reader to show that neither $x + y = k$ nor $x + z = k$ can be shown transient. □

11.2 Weak Fairness

A transient predicate is falsified by *every* "enabled" action under minimal progress. Under weak fairness, however, it is sufficient to have a single action falsify the predicate. Define,

$$p \;\textbf{transient} \;\triangleq\; \langle \exists\, s \;\; :: \;\; \{p\} \;\; s \;\; \{\neg p\} \rangle$$

where s is over all actions in the system.

The following operational argument shows that eventually $\neg p$ holds given that p is transient. Let t be an action that falsifies p. From the weak fairness condition, t is executed eventually. If $\neg p$ holds prior to the execution of t then the proof is over. Otherwise, p holds prior to the execution of t, and, from $\{p\}$ t $\{\neg p\}$, we conclude that $\neg p$ holds following the execution of t. (Note that the execution of t is assumed to terminate.)

Example: We consider the program **Fairness** of Section 10. The following predicates can be shown to be transient. For any integer k,

$$x=k \;,\; y=k \;,\; x+y=k \;,\; y+z=k \;,\; x+z=k \;,\; x+y+z=k$$

The predicate $z = k$ cannot be shown transient, because we cannot display an appropriate statement. The only statement that modifies z is

$$z \;:=\; z+1 \qquad \text{if} \quad x=y$$

and, this statement does not satisfy

$$\{z=k\} \;\; z \;:=\; z+1 \qquad \text{if} \quad x=y \;\; \{z \neq k\}$$

The reader may also show that $x \leq k$ cannot be proven to be transient. □

11.3 Strong Fairness

Transient predicates for strong fairness can be defined using a recursive definition employing *leads-to*; see [18] for details (in this reference the progress operator *ensures* is defined for strong fairness; transient predicates and *ensures* are related—see Section 12.1). We do not plan to employ this definition nor consider strong fairness in any detail. We give a rule in Section 13.5—that is sound but incomplete—to prove progress properties under strong fairness.

11.4 Derived Rules

We establish two simple rules about transient predicates that hold under either notion of fairness—minimal progress or weak fairness. These rules are primarily used in proving the derived rules for *leads-to*; these are rarely used in deducing properties of specific programs.

- The only predicate that is both stable and transient is *false*:

$$(p \text{ stable } \wedge\ p \text{ transient}) \ \equiv\ \neg p$$

- (strengthening) $$\frac{p \text{ transient}}{p\ \wedge\ q \text{ transient}}$$

Additionally, the substitution axiom can be used as usual.

It is easy to see from the definition of transient predicates that *false* is transient; it is known that *false* is stable. Therefore, in order to establish the first rule, we show, below, that

$$(p \text{ stable } \wedge\ p \text{ transient}) \ \Rightarrow\ \neg p$$

Proof of (p stable $\wedge$ p transient) $\Rightarrow$ $\neg p$ (under minimal progress)

For any statement with guard g_i and body s_i:

$\{p \wedge g_i\}\ s_i\ \{p\}$	, p stable
$\{p \wedge g_i\}\ s_i\ \{\neg p\}$	, p transient
$\{p \wedge g_i\}\ s_i\ \{\mathit{false}\}$	, conjunction of the above two
$\neg(p \wedge g_i)$	, from the Law of Excluded Miracle
$p \Rightarrow \neg g_i$	, simplifying the above
$p \Rightarrow \langle \forall\ i\ ::\ \neg g_i \rangle$	, conjoining over all i
$p \Rightarrow \langle \exists\ i\ ::\ g_i \rangle$	, definition of p transient
$\neg p$	, conjoining the above two □

Proof of (p stable) $\wedge$ (p transient) $\Rightarrow$ $\neg p$ (under weak fairness)

From the definition of p transient, there is a statement t such that

$\{p\}\ t\ \{\neg p\}$	, p transient
$\{p\}\ t\ \{p\}$	, p stable
$\{p\}\ t\ \{\mathit{false}\}$	, conjunction of the above two
$\neg p$	, the Law of Excluded Miracle □

Proof of the strengthening rule (under minimal progress)

$p \Rightarrow \langle \exists\ i\ ::\ g_i \rangle$	, p transient
$p \wedge q \Rightarrow \langle \exists\ i\ ::\ g_i \rangle$	, predicate calculus

Also, for a statement with guard g_i and body s_i:

$\{p \wedge g_i\}\ s_i\ \{\neg p\}$	, p transient
$\{p \wedge q \wedge g_i\}\ s_i\ \{\neg p \vee \neg q\}$	, strengthening the lhs and weakening the rhs

Hence, $p \wedge q$ transient $\square$

Proof of the strengthening rule (under weak fairness)

There is a statement t such that

$\{p\}\ t\ \{\neg p\}$	, p transient
$\{p \wedge q\}\ t\ \{\neg p \vee \neg q\}$	, strengthening the lhs and weakening the rhs
$p \wedge q$ transient	, definition of transient $\square$

Discussion

The notion of stability (and its generalization in **co**) was fundamental in developing our theory of safety; a stable predicate is guaranteed never to be falsified. The notion of transient, fundamental to a theory of progress, is almost the opposite of stability; a transient predicate is guaranteed to be falsified. A predicate can be established stable only by examining *all* actions in a system (i.e., a universal quantification appears in its definition). Under weak fairness, a predicate can be established transient by examining *some* action in the system (i.e., the definition involves existential quantification).

It is an interesting research question to identify problem areas where stability and transience are the only useful logical notions. For such cases, special purpose theories may be efficient in deriving system properties.

12 ensures, leads-to

Our primary progress operator is *leads-to*. It is the transitive, disjunctive closure of an operator, *ensures*. It is possible to eliminate *ensures* from our theory, replacing it by **co**-properties and transient predicates. However, to maintain continuity with [5], we introduce *ensures* though its role is now considerably diminished.

12.1 ensures

The informal meaning of p *ensures* q (abbreviated p **en** q) is: If p holds at any point in the computation it will continue to hold as long as q does not hold; eventually q holds; further, there is one (atomic) action which guarantees to establish q starting in any p-state. Formally,

$$p \textbf{ en } q \triangleq (p \wedge \neg q \textbf{ co } p \vee q) \wedge (p \wedge \neg q \text{ transient})$$

It follows from the **co**-property in the above definition that once p holds it continues to hold as long as q does not. Now, we justify in operational terms that once p holds, q holds eventually ("eventually" includes the present moment). Consider a state in which p holds and q does not. Since $p \wedge \neg q$ is transient, it is eventually falsified. From the **co**-property, whenever $(p \wedge \neg q)$ is falsified, $(p \vee q)$ holds. Hence, whenever $(p \wedge \neg q)$ is falsified, $\neg(p \wedge \neg q) \wedge (p \vee q)$, i.e., q, holds.

12.2 leads-to

The informal meaning of $p \mapsto q$ (read: p *leads-to* q) is "if p holds at any point in the computation then q will hold" (here 'will' applies to the current point as well as the future). There is no guarantee, unlike *ensures*, that p remains *true* until q holds.

The definition of $p \mapsto q$, given below, is recursive. For the basis, we deduce $p \mapsto q$ from p **en** q. The transitivity rule is justified as follows. From $p \mapsto q$, once p holds q will hold, and from $q \mapsto r$, once q holds r will hold. Therefore, once p holds r will hold. The disjunction rule says that from any state that satisfies some predicate p in a given set, q will eventually be established, provided $p \mapsto q$ for every p in that set. An alternative definition of *leads-to* that eliminates **en** appears in Misra[34, Chapter 4, Exercise 4]. Also, Misra[34, Chapter 4, Exercise 19] shows that the transitivity and the disjunction rules may be combined into a single rule.

(basis)
$$\frac{p \ \mathbf{en}\ q}{p \mapsto q}$$

(transitivity)
$$\frac{p \mapsto q \ , \ q \mapsto r}{p \mapsto r}$$

(disjunction) In the following, S is any set of predicates.

$$\frac{\langle \forall\ p\ :\ p \in S\ :\ p \mapsto q \rangle}{\langle \exists\ p\ :\ p \in S\ :\ p \rangle \mapsto q}$$

As usual, the substitution axiom applies.

The form of this definition differs from the way we defined **co**, **transient** and **en**. The current definition is recursive, and $\mapsto$ can be understood as an extreme solution (a least fixpoint) of an equation, a topic that we explore in Section 14.2. The inference rules provide important guidelines for structuring progress proofs: Either a proof follows directly from the program text (in the basis case), or it has to be structured as a transitive or a disjunctive proof. These rules can also be used to establish the derived rules for $\mapsto$ using structural induction over its definition[5, chapter 3].

In Section 14.3, we show that the disjunction rule over a finite set, though useful in practice, can be deduced from the basis and transitivity rules. Therefore, the real power of the disjunction rule lies in its application to an infinite set of predicates.

Note: The only mention of **en** is in the basis rule. The premise of that rule could be replaced by the definition of p **en** q—$p \wedge \neg q$ **co** $p \vee q$, $p \wedge \neg q$ **transient**—thus eliminating **en** from our theory. □

Examples: In the following, variables x, y are integers.

1. A hungry philosopher eats. Using h, e to denote, respectively, that a particular philosopher is hungry or eating, we say $h \mapsto e$.
2. Variable x changes eventually. For every integer m, $x = m \mapsto x \neq m$. This can be written equivalently as $true \mapsto x \neq m$.
3. Variable x grows without bound. For every integer m, $true \mapsto x > m$. This is an abbreviation for $\langle \forall\ m\ ::\ true \mapsto x > m \rangle$. It should not be confused with $true \mapsto \langle \forall\ m\ ::\ x > m \rangle$, which happens to be nonsense.
4. Variables x, y do not retain their values forever if they differ. For all integers m, n

   $$x, y = m, n \ \wedge\ m \neq n \ \mapsto\ x, y \neq m, n$$

   or, equivalently, $\langle \forall\ m, n\ :\ m \neq n\ :\ x, y = m, n \mapsto x, y \neq m, n \rangle$.

   Observe that there is no guarantee for x, y to ever become equal.

5. Predicate p holds infinitely often. We say, $true \;\mapsto\; p$ or, equivalently, $\neg p \;\mapsto\; p$.

6. If p remains *true* forever then q holds eventually.

   Another way of expressing this property is to say that eventually either p is *false* or q holds:

   $$true \;\mapsto\; \neg p \;\vee\; q$$

7. If p holds infinitely often (in all executions) then so does q.

   $$(true \;\mapsto\; p) \;\Rightarrow\; (true \;\mapsto\; q)$$

   This property does *not* say that "In any execution, if p holds infinitely often then so does q." This latter property is stronger than our formulation. This is because, if in some executions p holds infinitely often and in some other executions p holds finitely often, the first formulation makes no guarantees about q in any execution; the second formulation requires q to hold infinitely often whenever p does.

8. A given program "terminates," i.e., starting in any state that satisfies the initial condition eventually a state is reached that satisfies the fixed point predicate, FP.

   $$\text{initial-condition} \;\mapsto\; FP$$

12.3 Derived Rules

Effective applications of the following derived rules can shorten progress proofs by an order of magnitude (I speak from personal experience). The rules are divided into two classes, *lightweight* and *heavyweight*. The former includes rules whose validity is easily established; the latter set of rules are not entirely obvious. Each application of a heavyweight rule goes a long way toward completing a progress proof.

The lightweight rules can be proven directly from the infererence rules for $\mapsto$. The heavyweight rules often require induction on the structure of the progress proofs in the premises. Proofs of these rules are similar to those in [5, chapter 3].

Lightweight Rules

- (implication) $$\frac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$$

The remaining lightweight rules have counterparts for **co**.

- (lhs-strengthening, rhs-weakening) $$\frac{p \;\mapsto\; q}{\begin{array}{l} p' \;\wedge\; p \;\mapsto\; q \;, \\ p \;\mapsto\; q \;\vee\; q' \end{array}}$$

  - (disjunction) In the following, i is quantified over any arbitrary set, and p_i, q_i are predicates.

  $$\frac{\langle \forall\, i \;::\; p_i \;\mapsto\; q_i \rangle}{\langle \exists\, i \;::\; p_i \rangle \;\mapsto\; \langle \exists\, i \;::\; q_i \rangle}$$

- (cancellation) $$\frac{p \;\mapsto\; q \;\vee\; r \;,\; r \;\mapsto\; s}{p \;\mapsto\; q \;\vee\; s}$$

We deduce from the implication rule that for any predicate p,

$$p \;\mapsto\; p \quad \text{and} \quad \mathit{false} \;\mapsto\; p \;.$$

The lhs-strengthening and the rhs-weakening rules—also valid for **co**-properties—are used extensively in proofs. The cancellation rule played little role in manipulating the **co**-properties; it is, however, used quite often in progress proofs. The cancellation rule reduces to transitivity when q is *false*. Note that there is no conjunction rule for $\mapsto$ analogous to the one for **co**.

Heavyweight Rules

- (impossibility) $$\frac{p \;\mapsto\; \mathit{false}}{\neg p}$$

- (PSP) $$\frac{p \;\mapsto\; q \;, \quad r \;\mathbf{co}\; s}{p \;\wedge\; r \;\mapsto\; (q \;\wedge\; s) \;\vee\; (\neg r \;\wedge\; s)}$$

- (induction) Let M be a total function from program states to the set W. Let $(W, <)$ be well-founded. The variable m in the following premise ranges over W.

$$\frac{\langle \forall\, m \;\; :: \;\; p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q \rangle}{p \;\mapsto\; q}$$

where p, q do not name m.

- (completion) Let i take on a finite set of values, and let p_i, q_i be predicates for each i.

$$\frac{\langle \forall\, i \;\; :: \;\; p_i \;\mapsto\; q_i \;\vee\; b \quad q_i \;\mathbf{co}\; q_i \;\vee\; b \rangle}{\langle \forall\, i \;\; :: \;\; p_i \rangle \;\mapsto\; \langle \forall\, i \;\; :: \;\; q_i \rangle \;\vee\; b}$$

The impossibility rule says that a state satisfying *false* is reachable only from an unreachable state (read the consequent of the rule, "$\neg p$ invariant").

The PSP rule (for Progress-Safety-Progress) is perhaps the most widely used rule in progress proofs. It allows us to structure a progress proof as a safety proof—establishing r **co** s—and a progress proof—establishing $p \;\mapsto\; q$—which are then combined. This rule is so important that it should be memorized before attempting serious progress proofs. The way to memorize the rule is: the left side is the conjunction of the two left sides, and the right side consists of two disjuncts, one is the conjunction of the two right sides and the other is obtained from the **co**-property alone by conjoining its two sides (while negating its left side).

Function M in the induction rule is called a *variant function* or a *metric*. The premise of the induction rule says that from any state in which p holds eventually a state is reached where p still holds and the metric has a lower value, or q is established. Since M takes values from a well-founded set, its value cannot decrease indefinitely. Therefore, q is eventually established starting in any state where p holds. (Note: It is sufficient to require that M's value be in W whenever $p \;\wedge\; \neg q$ holds.) Some common examples of well-founded relations are: less-than relation over positive integers (and natural numbers), lexicographic order over sequences of bounded length (where the order on the sequence items is, itself, well-founded), prefix or subsequence relation over sequences and subset relation over finite sets.

The completion rule is a way to take conjunctions of progress properties. As we have remarked earlier, there is no conjunction rule for $\mapsto$, analogous to the rule for **co**-properties. Under additional assumptions about the predicates in the rhs of the *leads-to*'s (given by the **co**-properties), such a conjunction rule is valid.

12.4 Corollaries of the Derived Rules

The following corollary, like its namesake for **co**, permits us to conjoin a stable predicate to both sides of a *leads-to* property.

- (stable conjunction) $$\frac{p \;\mapsto\; q \;,\; r \quad \text{stable}}{p \;\wedge\; r \;\mapsto\; q \;\wedge\; r}$$

Proof: Set s to r in the PSP rule. □

- (Corollary of Induction) In the following corollary the range of m (in the premise of the induction rule) is restricted to a predicate $r.m$. Suppose "$<$" is a well-founded order over r.

$$\frac{\begin{array}{l}\langle\forall\, m \;:\; r.m \;: \\ \qquad p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q \\ \rangle\end{array}}{p \;\mapsto\; (p \;\wedge\; \neg r.M) \;\vee\; q}$$

Proof:

$p \;\wedge\; M = m \;\wedge\; r.m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q$, premise
$p \;\wedge\; M = m \;\wedge\; r.M \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q$, rewriting lhs
$p \;\wedge\; M = m \;\wedge\; \neg r.M \;\mapsto\; p \;\wedge\; \neg r.M$, implication
$p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; (p \;\wedge\; \neg r.M) \;\vee\; q$
, disjunction
$p \;\mapsto\; (p \;\wedge\; \neg r.M) \;\vee\; q$, induction □

An important special case arises when M is integer valued, $r.m$ is of the form $m > L$, for some lower bound L, and "$<$" is the standard less-than relation over integers. Then, we conclude from the above corollary

- (induction on integers) $$\frac{\begin{array}{l}\langle\forall\, m \;:\; m > L \;: \\ \qquad p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M < m) \;\vee\; q \\ \rangle\end{array}}{p \;\mapsto\; (p \;\wedge\; M \le L) \;\vee\; q}$$

If $r.m$ imposes no lower bound on m—for instance, $r.m \;\equiv\; \mathit{true}$—then the conclusion holds for any L.

Using "greater-than" in place of "less-than," we have the analogous

- (induction on integers) $$\frac{\begin{array}{l}\langle\forall\, m \;:\; m < L \;: \\ \qquad p \;\wedge\; M = m \;\mapsto\; (p \;\wedge\; M > m) \;\vee\; q \\ \rangle\end{array}}{p \;\mapsto\; (p \;\wedge\; M \ge L) \;\vee\; q}$$

13 Applications

We consider a few small examples in which we employ the progress operators for specifications and deductions of program properties. The emphasis is on converting verbal descriptions to formal specifications and using the derived rules for deductions. In most cases, we also supply the typical verbal arguments that justify the deductions and contrast them with formal proofs.

13.1 Common Meeting Time

The Common Meeting Time problem was discussed in Section 5.1. We had functions f, g that mapped nonnegative reals to nonnegative reals. Now, we tighten the requirements on f, g: These functions map natural numbers to natural numbers. Thus, for all natural m, n

$$\begin{aligned} m \leq n \;\Rightarrow\; f(m) \leq f(n) \\ m \leq n \;\Rightarrow\; g(m) \leq g(n) \end{aligned} \qquad \text{(CMT1)}$$

A variable t—previously of type real, now of type natural number—is postulated to satify

$$\textbf{initially } t = 0 \qquad \text{(CMT2)}$$
$$t = m \textbf{ co } t \leq \max(f(m), g(m)) \qquad \text{(CMT3)}$$

We had established earlier, from CMT1– CMT3, that t exceeds no common meeting time:

$$com(n) \;\Rightarrow\; t \leq n \qquad \text{(CMT4)}$$

The essential safety property, (CMT3), can be implemented by program *skip*, that does not change t. In order to guarantee that t eventually equals the earliest common meeting time, we add a progress requirement: If t is not a common meeting time then it increases eventually.

$$\neg com(t) \;\wedge\; t = m \;\mapsto\; t > m \qquad \text{(CMT6)}$$

We show that eventually t will be equal to a common meeting time, if there is one.

$$\langle \exists\, n \;::\; com(n) \rangle \;\mapsto\; com(t) \qquad \text{(CMT7)}$$

Together (CMT4) and (CMT7) imply that if there is a common meeting time, t will eventually equal the earliest common meeting time, if one exists.

Proof of (CMT7): For natural numbers m, n

$\neg com(t) \;\wedge\; t = m \;\mapsto\; t > m$, rewrite (CMT6)
$com(t) \;\wedge\; t = m \;\mapsto\; com(t)$, implication
$t = m \;\mapsto\; t > m \;\vee\; com(t)$, disjunction of the above two
$true \;\mapsto\; t > n \;\vee\; com(t)$, induction on integers
$com(n) \;\mapsto\; (com(n) \;\wedge\; t > n) \;\vee\; com(t)$
, stable conjunction with $com(n)$
$com(n) \;\mapsto\; com(t)$, use CMT4 to cancel the first disjunct in the rhs
$\langle \exists\, n \;::\; com(n) \rangle \;\mapsto\; com(t)$, disjunction over all n. □

This proof is invalid if t is of type real. Then the induction step in the above proof is invalid. The reader can construct a counterexample to (CMT8) by having t increase extremely slowly, say, by $1/2^i$ in the i^{th} step.

One way to implement the progress condition, CMT6, is to increment t by 1 in each step and check to see if $com(t)$ holds. The monotonicity condition on f, g – given by CMT1– is far too weak to permit many other strategies. If the functions are also ascending, i.e., for all natural n

$$n \leq f(n),\ n \leq g(n)$$

then the programs (P1,P2) in section 5.1, satisfy

$$\neg com(t) \wedge t = m \ \ \textbf{en} \ \ t > m$$

and, hence, CMT6 as well.

13.2 Token Ring

We have considered the properties of a token ring in Section 5.2. The following safety properties were postulated there.

Notation: We write $h_i/e_i/t_i$ to denote that process i is hungry / eating / thinking. These predicates are mutually exclusive and $h_i \ \vee \ t_i \ \vee \ e_i \equiv true$. The position of the token is in the variable p, i.e., $p = i$ (as in TR5) denotes that process i holds the token. In the following, i ranges over all processes.

$$\begin{array}{ll} \textbf{initially}\ e_i \ \Rightarrow \ p = i & \text{(TR0)} \\ e_i \ \textbf{co}\ e_i \ \vee \ t_i & \text{(TR1)} \\ t_i \ \textbf{co}\ t_i \ \vee \ h_i & \text{(TR2)} \\ h_i \ \textbf{co}\ h_i \ \vee \ e_i & \text{(TR3)} \\ h_i \ \wedge \ p \neq i \ \textbf{co}\ h_i & \text{(TR4)} \\ p = i \ \textbf{co}\ p = i \ \vee \ \neg e_i & \text{(TR5)} \end{array}$$

It was possible (in Section 5.2) to deduce mutual exclusion, a safety property, from (TR0–TR5). Now, we postulate some progress properties and establish the absence of starvation.

First, we require that a hungry tokenholder transit to eating.

$$h_i \ \wedge \ p = i \ \mapsto \ e_i \qquad \text{(TR6)}$$

Next, we require that the token move from the current tokenholder to its right neighbor: In the following, i' is the right neighbor of i.

$$p = i \ \mapsto \ p = i' \qquad \text{(TR7)}$$

Note that (TR7) does not require the token to go from i to i' directly.

We establish absence of starvation for any j, $0 \leq j < N$. That is, we show for any j, $0 \leq j < N$,

$$h_j \ \mapsto \ e_j \qquad \text{(TR8)}$$

Proof of (TR8): Consider an arbitrary j, $0 \leq j < N$. First, we show that j eventually holds the token, i.e.,

$$true \ \mapsto \ p = j \qquad \text{(TR9)}$$

The proof is by induction over (TR7). To apply the induction rule we define, $i' \prec i$ for all process indices i, where $i \neq j$. The relation $\prec$ induces a linear order over the processes,

$$j \prec \ldots i' \prec i \ldots \prec j'.$$

We can rewrite (TR7) as

$$\langle \forall\, i \;::\; p = i \;\mapsto\; p \prec i \;\vee\; p = j \rangle$$

Apply the induction rule (since $\prec$ is a linear order over a finite set) to get (TR9). Now, we are ready to prove (TR8).

$$\begin{array}{ll}
h_j \;\mathbf{co}\; h_j \;\vee\; e_j & \text{, from (TR3) using } j \text{ for } i \\
true \;\mapsto\; p = j & \text{, (TR9)} \\
h_j \;\mapsto\; (h_j \;\wedge\; p = j) \;\vee\; e_j & \text{, PSP } \{\text{use } (\neg h_j \;\wedge\; e_j) = e_j\} \\
h_j \;\wedge\; p = j \;\mapsto\; e_j & \text{, from (TR6) using } j \text{ for } i \\
h_j \;\mapsto\; e_j & \text{, cancellation on the above two}
\end{array}$$

The standard verbal argument for this problem follows the above proof steps closely. The formalism allows us to combine a few special cases. The role of induction—if every process relinquishes the token eventually, then every process acquires the token eventually—is made explicit in our proof. Note that the proof is entirely independent of the kind of fairness assumed in (TR6) and (TR7).

13.3 Unordered Channel

We consider a channel, directed from one process to another, along which messages are sent from the former to the latter. It is required to design a protocol by which every message sent is delivered eventually, though the order of delivery may be different from the order of transmission. The following scheme implements the protocol. Every message sent along the channel is assigned a *sequence-number* (a natural number) and the sequence-numbers are strictly increasing in the order of transmission. A message that has the lowest sequence-number in the channel at any point in the computation is eventually delivered. We claim that this scheme guarantees delivery of every message sent. Note, however, that the delivery order may not be monotonic in sequence-numbers.

The proposed scheme can be described by the following properties. In this description, s is the set of sequence-numbers of the messages in the channel, and x, y are arbitrary sequence-numbers. The lowest sequence number in s is $\min .s$, where for an empty set s we take $\min .s$ to be ∞. If $\min .s$ has a value x then $\min .s$ cannot decrease as a result of adding a number to s, from the monotonicity of sequence numbers. Also, removing a number from s does not decrease $\min .s$. Hence, the safety property, UC1, says that $\min .s$ never decreases as long as s is nonempty. The progress property, UC2, says that the smallest element of s is eventually removed from s. Our goal, UC3, is to show that every sequence-number is eventually outside s. We assume throughout that s is a finite set.

$$\min .s = x \;\mathbf{co}\; \min .s \geq x \tag{UC1}$$
$$\min .s = x \;\mapsto\; x \notin s \tag{UC2}$$

Show

$$true \;\mapsto\; y \notin s \tag{UC3}$$

Our proof formalizes the following argument. From UC1, $\min .s$ never decreases. From UC2, if $x = \min .s$ then x is eventually removed; therefore, $\min .s$ increases eventually. It follows that $\min .s$ increases without bound as long as s is nonempty; hence, every number is eventually outside s.

$$\begin{array}{ll}
\min .s = x \;\mapsto\; (x \notin s \;\wedge\; \min .s \geq x) \;\vee\; (\min .s \neq x \;\wedge\; \min .s \geq x) & \\
& \text{, PSP on UC1, UC2} \\
\min .s = x \;\mapsto\; \min .s > x & \\
& \text{, simplifying the rhs} \\
true \;\mapsto\; \min .s > y & \text{, induction on integers (using } y \text{ as the upper bound)} \\
true \;\mapsto\; y \notin s & \text{, weakening the rhs using } \min .s > y \;\Rightarrow\; y \notin s
\end{array}$$

13.4 Dynamic Graphs

In the problem treated in Section 5.9, a finite directed graph could be modified by directing all incident edges on a node towards that node; the effect of this operation was to make that node a bottom node. We showed that this operation does not create new paths to nonbottom nodes, i.e., for arbitrary nodes u, v, with $u\ R\ v$ denoting that there is a path from u to v and $v.\perp$ denoting that v is bottom

$$\neg u\ R\ v \ \mathbf{co}\ \neg u\ R\ v\ \vee\ v.\perp \tag{G1}$$

We showed that one of the consequences of (G1) is that once the graph becomes acyclic, it remains acyclic. We assume, henceforth, that the graph is initially acyclic; therefore, it is always acyclic, i.e., for all u

$$\neg u\ R\ u \tag{G2}$$

Now, we add a progress requirement. Call a node a *top* node if it has no incoming edge (recall that a *bottom* node has no outgoing edge; an isolated node is both top and bottom). We require that the edge redirection be applied to every top node eventually. The effect of edge redirection on a top node is to make it a bottom node. Therefore, using $v.\top$ to denote that v is a top node,

$$v.\top\ \mapsto\ v.\perp \tag{G3}$$

We will show that every node eventually becomes a bottom node, i.e.,

$$\mathit{true}\ \mapsto\ u.\perp \tag{G4}$$

The main arguments behind the proof of (G4) are as follows. For any node u, consider the set of *ancestors* of u, i.e., the nodes that have paths to u. This set does not grow as long as u is nonbottom, a safety property that we will establish from G1. Next, since the graph is acyclic, either u is top or u has an ancestor, v, that is top. In the first case, from G3, u will become bottom. In the second case, from G3, v will become bottom and therefore cease to be an ancestor of u; hence, the ancestor set of u decreases in size eventually. Since the ancestor set cannot decrease forever, u will become top and then, from G3, eventually become bottom.

In the following, $u.an$ denotes the ancestor set of u, i.e.,

$$v \in u.an\ \equiv\ v\ R\ u \tag{G5}$$

We will use the following facts about top and bottom. A node is top iff its ancestor set is empty (G6) and a bottom node does not belong to any ancestor set (G7).

$$u.\top\ \equiv\ u.an = \phi \tag{G6}$$
$$v.\perp\ \Rightarrow\ v \notin u.an \tag{G7}$$

First, we establish that the ancestor set does not grow as long as a node remains nonbottom. In the rest of this argument, S is any fixed set of nodes.

Lemma: $u.an = S\ \mathbf{co}\ u.an \subseteq S\ \vee\ u.\perp$ (G8)

Proof:

$\neg v\ R\ u\ \mathbf{co}\ \neg v\ R\ u\ \vee\ u.\perp$	, G1 with u, v interchanged
$v \notin u.an\ \mathbf{co}\ v \notin u.an\ \vee\ u.\perp$	, using G5 to rewrite $\neg v\ R\ u$
$\langle \forall\ v\ :\ v \notin S\ :\ v \notin u.an\rangle\ \mathbf{co}\ \langle \forall\ v\ :\ v \notin S\ :\ v \notin u.an\rangle\ \vee\ u.\perp$	
	, conjunction over all v, $v \notin S$
$u.an\ \subseteq S\ \mathbf{co}\ u.an\ \subseteq\ S\ \vee\ u.\perp$	, simplifying the two sides
$u.an = S\ \mathbf{co}\ u.an\ \subseteq\ S\ \vee\ u.\perp$	, strengthening lhs $\square$

Lemma: (G4) $true \;\mapsto\; u.\bot$

Proof:

$v.\top \;\mapsto\; v.\bot$	, G3
$u.an = S \;\wedge\; v.\top \;\mapsto\; v \notin u.an$	, strengthening lhs and weakening rhs (using G7)
$u.an = S \;\wedge\; v.\top \;\wedge\; v \in S \;\mapsto\; v \notin u.an \;\wedge\; v \in S$	
	, stable conjunction with $v \in S$, where S is a subset of nodes
$u.an = S \;\wedge\; v.\top \;\wedge\; v \in S \;\mapsto\; u.an \neq S$	
	, weakening rhs
$u.an = S \;\wedge\; \langle \exists\, v \;::\; v.\top \;\wedge\; v \in S \rangle \;\mapsto\; u.an \neq S$	
	, disjunction over v
$u.an = S \;\wedge\; S \neq \phi \;\mapsto\; u.an \neq S$	, an acyclic graph S has a top node $\equiv\; (S \neq \phi)$
$u.an = S$ **co** $u.an \subseteq\; S \;\vee\; u.\bot$	, from Lemma (G8)
$u.an = S \;\wedge\; S \neq \phi \;\mapsto\; u.an \subset\; S \;\vee\; u.\bot$	
	, PSP and weakening rhs
$u.an = S \;\wedge\; S = \phi \;\mapsto\; u.\bot$	, $u.an = \phi \;\Rightarrow\; \{\text{G6}\}\; u.\top \;\mapsto\; \{\text{G3}\}\; u.\bot$
$u.an = S \;\mapsto\; u.an \;\subset\; S \;\vee\; u.\bot$	, disjunction of the above two
$true \;\mapsto\; u.\bot$	, induction (subset relation over finite sets is well founded) □

13.5 Strong Semaphore: An Example of Strong Fairness

The standard example to illustrate the concept of strong fairness is a binary semaphore that is shared by two processes. Let x, y denote the number of times that the two processes, respectively, are granted the semaphore (i.e., successfully complete their P-operations on the semaphore). We assume that each process releases the semaphore (i.e., performs a V-operation) eventually. Boolean variable s is *true* iff the semaphore is not granted currently to either process. An outline of the program that manages the semaphore is as follows.

$$
\begin{array}{lllll}
 & \alpha & :: & s, x := \mathit{false}, x+1 & \text{if } s \\
[\!] & \beta & :: & s, y := \mathit{false}, y+1 & \text{if } s \\
[\!] & \gamma & :: & s := \mathit{true} &
\end{array}
$$

Under the weak fairness assumption we can establish only that $x + y$ increases without bound; we can make no such guarantee for either x or y. Now, we impose the following strong fairness condition for statement α: If the guard of α (i.e., s) is infinitely often *true* then α is executed infinitely often. We assume weak fairness for the remaining statements, β, γ. Our goal is to show that x increases without bound under this strong fairness condition.

The strong fairness condition can be added as an axiom to the program; for any integer k,

$$(true \;\mapsto\; s) \;\Rightarrow\; (x = k \;\mapsto\; x = k+1) \qquad \text{(SF)}$$

So, we regard our system as consisting of the program (with weak fairness condition) plus the axiom SF. It is then straightforward to show that for any integer m

$$true \;\mapsto\; x > m$$

The proof is as follows.

$true$ **en** s	, from the program text
$true \;\mapsto\; s$	, definition of $\mapsto$
$x = k \;\mapsto\; x = k+1$	, from above using SF
$true \;\mapsto\; x > m$	, induction on integers

This treatment of strong fairness, though sound, is incomplete. To see this we consider the following program where x, y are integers and b is boolean, and all statements are executed under strong fairness.

$$\begin{array}{llllll} & \alpha & :: & x := x+1 & \text{if} & b \\ [] & \beta & :: & y := y+1 & \text{if} & \neg b \\ [] & \gamma & :: & b := \neg b & \text{if} & x = y \end{array}$$

We see that $x + y$ is guaranteed to increase by the following operational argument. Both x, y are nondecreasing. In any execution either b holds infinitely often or $\neg b$ holds infinitely often; therefore, either x or y increases infinitely often and hence, $x + y$ increases without bound. Unfortunately, our proof method cannot be used to prove this result; adding the following strong fairness conditions to the program

$$\begin{array}{l} (true \ \mapsto \ b) \ \Rightarrow \ (x = k \ \mapsto \ x = k+1) \text{ and} \\ (true \ \mapsto \ \neg b) \ \Rightarrow \ (y = n \ \mapsto \ y = n+1) \end{array}$$

does not help us to prove that $x + y$ increases. This is because we can prove neither

$$true \ \mapsto \ b \quad \text{nor} \quad true \ \mapsto \ \neg b$$

(because, if initially $x, y, b = 0, 1, \mathit{false}$ then $(x < y) \wedge \neg b$ persists forever; similarly with $x, y, b = 1, 0, true$ as the initial condition $(x > y) \ \wedge \ b$ persists). The trouble is

$$(true \ \mapsto \ b) \ \Rightarrow \ (x = k \ \mapsto \ x = k+1)$$

merely says that if b holds infinitely often in *all* execution sequences then x will be incremented infinitely often in *all* execution sequences. What we need is a stronger statement: In any execution sequence, if b holds infinitely often then x is incremented infinitely often (in that sequence). Our logic does not permit us to say this.

14 Theoretical Issues

We touch upon a few theoretical issues concerning *leads-to*. In Section 14.1 we show the existence of the "weakest predicate that *leads-to*" q, for any q, and we prove some of its properties. We give a fixpoint characterization of this predicate in Section 14.2. The role of the disjunction rule is examined in Section 14.3; We show that the validity of disjunction over a finite set of predicates is derivable from the basis and transitivity rules; therefore, the main use of the disjunction rule is in its application to an infinite set of predicates.

14.1 *wlt*

For a predicate q, let $wlt.q$ be the weakest predicate that *leads-to* q. The definition of wlt is

$$wlt.q \ \triangleq \ \langle \exists \, p \ : \ p \ \mapsto \ q \ : \ p \rangle \tag{W1}$$

We show that $wlt.q$ is indeed the weakest predicate leading to q, i.e.,

$$p \ \mapsto \ q \ \equiv \ (p \ \Rightarrow \ wlt.q) \tag{W2}$$

To see this, we first prove $(p \ \mapsto \ q) \ \Rightarrow \ (p \ \Rightarrow \ wlt.q)$

$p \mapsto q$, assume
$p \Rightarrow wlt.q$, from (W1)

Conversely, we show $(p \Rightarrow wlt.q) \Rightarrow (p \mapsto q)$
$\langle \forall\, r : r \mapsto q : r \mapsto q \rangle$, trivially
$\langle \exists\, r : r \mapsto q : r \rangle \mapsto q$, disjunction
$wlt.q \mapsto q$, using (W1)
$p \mapsto q$, strengthening the lhs using $p \Rightarrow wlt.q$

This concludes the proof of (W2). We can use (W2) to answer if $p \mapsto q$ for arbitrary p, q, by answering if $p \Rightarrow wlt.q$. This is the preferred method when $wlt.q$ can be computed efficiently; see Kaltenbach[19] for an implementation that uses this approach for finite state programs.

As special cases of (W2), substituting $wlt.q$ for p and using $wlt.q \Rightarrow wlt.q$

$$wlt.q \mapsto q \tag{W3}$$

Also, substituting q for p in (W2) and using $q \mapsto q$,

$$q \Rightarrow wlt.q \tag{W4}$$

Next, we show

$$wlt.q \wedge \neg q \;\mathbf{co}\; wlt.q \tag{W5}$$

Proof of (W5): $wlt.q \wedge \neg q \;\mathbf{co}\; wlt.q$

Given $p \mapsto q$, for any p, q, we show that there is a predicate b satisfying
1. $p \Rightarrow b$
2. $b \mapsto q$
3. $b \wedge \neg q \;\mathbf{co}\; b \vee q$

We establish (W5) from (1,2,3), as follows.
$wlt.q \mapsto q$, from (W3)
Using (1,2,3), with p as $wlt.q$ and q as q, there is a predicate w such that
$1'$. $wlt.q \Rightarrow w$
$2'$. $w \mapsto q$
$3'$. $w \wedge \neg q \;\mathbf{co}\; w \vee q$

Now,
$w \Rightarrow wlt.q$, from ($2'$ and W2)
$w \equiv wlt.q$, from above and $1'$
$wlt.q \wedge \neg q \;\mathbf{co}\; wlt.q \vee q$, from $3'$ replacing w by $wlt.q$
$wlt.q \wedge \neg q \;\mathbf{co}\; wlt.q$, simplifying rhs using (W4)

This establishes (W5). Proofs of (1,2,3) can be constructed by applying induction on the structure of the proof of $p \mapsto q$.

14.2 A Fixpoint Characterization of *wlt*

The following fixpoint characterization of *wlt* is under weak fairness. We define a predicate transformer *we*, that captures the essence of *ensures* and we define *wlt* in terms of *we*. These definitions have been found useful in automatic verifications of finite state programs where the extreme solutions (i.e., weakest or strongest) can be computed iteratively [19].

Predicate $we.p$ is the weakest predicate such that starting in any state satisfying $we.p$, eventually p is established, and $we.p$ holds until p is established. The predicate $we.p$ can be written as a disjunction of several predicates, $(we.p)_t$, one predicate for each action t in the given program. Predicate $(we.p)_t$ has the same meaning as $we.p$, but the additional requirement that p can be established by executing action t with precondition $(we.p)_t$. Specifically, $(we.p)_t$ is the weakest solution of (1) in q; here, $wp.s.q$ is the weakest precondition of s with postcondition q.

$$q \equiv (\langle \forall\ s\ ::\ wp.s.q \rangle \ \wedge\ wp.t.p)\ \vee\ p \tag{1}$$

Now, we can be defined. Below, t is quantified over all actions.

$$we.p \equiv \langle \exists\ t\ ::\ (we.p)_t \rangle \tag{2}$$

Finally, we define $wlt.q$ as the strongest solution in p of (3).

$$p \equiv q\ \vee\ we.p \tag{3}$$

The existence of the weakest solution for (1) and the strongest solution for (3) can be established by appealing to the Knaster-Tarski Theorem; see [17] for details. The definition of wlt given by equation (3) can be shown to be the same as the definition (W1) of Section 14.1. The fixpoint characterization yields a number of properties of we and wlt (we have seen some of these earlier):

$p \Rightarrow we.p$
$wlt.p \equiv p\ \vee\ we.(wlt.p)$
$[(p\ \vee\ we.q) \Rightarrow q] \Rightarrow [wlt.p \Rightarrow q]$
$we.(wlt.p) \Rightarrow wlt.p$
$p \Rightarrow wlt.p$
$(p \Rightarrow q) \Rightarrow (wlt.p \Rightarrow wlt.q)$
$wlt.false \equiv false$
$wlt.(wlt.p) \equiv wlt.p$
$wlt.(p\ \vee\ wlt.q) \Rightarrow wlt(p\ \vee\ q)$

14.3 The Role of the Disjunction Rule

This section is restricted to the study of progress under weak fairness. We have defined $\mapsto$ using three inference rules. The basis and the transitivity rules are intuitively acceptable. The need for the disjunction rule, however, is not easy to see. In this section, we show that the disjunction rule is (1) unnecessary for performing finite disjunctions and (2) necessary for performing infinite disjunctions. One consequence of this observation is that for a finite-state program—where the number of predicates themselves is finite and, hence, disjunction can be performed only over a finite number of predicates—the disjunction rule is unnecessary.

Let $\models\!\!\!\Rightarrow$ (a "poor cousin" of $\mapsto$) be the transitive closure of *ensures*, i.e., $\models\!\!\!\Rightarrow$ is defined by

$$\frac{p\ \mathbf{en}\ q}{p \models\!\!\!\Rightarrow q}$$

and

$$\frac{p \models\!\!\!\Rightarrow q\ ,\ q \models\!\!\!\Rightarrow r}{p \models\!\!\!\Rightarrow r}$$

We show that $\models\!\!\!\Rightarrow$ is finitely disjunctive, i.e.,

$$\frac{p \models\!\!\!\Rightarrow q\ ,\ p' \models\!\!\!\Rightarrow q}{p\ \vee\ p' \models\!\!\!\Rightarrow q}$$

For this proof, we first show

$$\frac{p \mapsto q}{p \vee r \mapsto q \vee r} \tag{1}$$

The demonstration uses induction on the structure of the proof of $p \mapsto q$.

(basis) Assume p **en** q
$p \vee r$ **en** $q \vee r$, direct from the definition of **en**
$p \vee r \mapsto q \vee r$, definition of $\mapsto$

(transitivity) Assume for some b that $p \mapsto b \mapsto q$
$p \vee r \mapsto b \vee r$, induction hypothesis
$b \vee r \mapsto q \vee r$, induction hypothesis
$p \vee r \mapsto q \vee r$, transitivity

Now, we can prove the finite disjunctivity of $\mapsto$.
$p \vee p' \mapsto q \vee p'$, from the premise $p \mapsto q$ and (1) using p' as r
$q \vee p' \mapsto q$, from the premise $p' \mapsto q$ and (1) using q as r
$p \vee p' \mapsto q$, transitivity on the above two

This concludes the proof that $\mapsto$ is finitely disjunctive. Next, we show that $\mapsto$ is *not* infinitely disjunctive.

We show that whenever $p \mapsto q$ can be proven for a program, there is some natural number, k, depending on p, q, such that for any state satifying p there is an execution of length at most k that establishes q. If $p \mapsto q$ is proven by p **en** q, then there is an action t such that

$$\{p \wedge \neg q\} \;\; t \;\; \{q\}$$

For a state satisfying $p \wedge q$, an empty execution sequence establishes q and for a state satisfying $p \wedge \neg q$, the sequence consisting of action t establishes q. Thus, the bound k equals 1, in this case. If $p \mapsto q$ is proven by $p \mapsto r \mapsto q$, then $k_1 + k_2$ is the required bound where k_1 is the bound for $p \mapsto r$ and k_2 for $r \mapsto q$.

Our impossibility result is derived by considering a program that consists of a single statement; the statement decrements by 1 the value of an integer variable, x. It is straightforward to show that in this program $true \mapsto x < 0$ holds. However, $true \mapsto x < 0$ cannot be proven. Because if it can be proven, there is a bound k associated with this proof. That is, for any state—in particular, $x = k$—we can find an execution sequence of length at most k establishing $x < 0$, which is impossible.

15 Synopsis

The major theme of this part of the paper is a definition of *leads-to* and the promulgation of its manipulation rules. The definition used the auxiliary concept of *transient predicate* (which is used to define *ensures*, that forms the basis for the definition of *leads-to*). Transient predicates are defined directly from the program text, for different forms of fairness.

The manipulation rules for *leads-to* consist of about four lightweight and four heavyweight rules. The examples illustrate how these rules can be effectively applied in practice.

Bibliographic Notes

The treatment of progress properties in this paper closely follows the original development described in [5]. The only new element is the introduction of transient predicates and the replacement of "*unless*" by "**co**" in the derived rules, such as the PSP and the completion rules. The minimal progress condition is due to Dijkstra[9]. For a comprehensive treatment of fairness see Francez[13] or Manna and Pnueli[27]. Transient predicates have been used informally before (they are called nonquiescent in ([28], [32]); the present definition is inspired by Cohen[7]. The definition of *ensures* for minimal progress appears in Jutla and Rao[18]. The definition of transient predicate under weak fairness is inspired by Lehmann, Pnueli and Stavi's notion[26] of "helpful transitions." The *leads-to* operator was introduced in Lamport[23]; its interpretation in linear temporal logic is in Owicki and Lamport[35]. Our definition, using inference rules, has facilitated proofs of the derived rules using structural induction. Lamport[25] prescribes deducing the progress properties from the conjunction of the fairness assumption—expressed as a formula in temporal logic—and the safety properties. The graph problem (Section 13.4) is from [29]; the original inspiration for this problem is from Chandy and Misra[3].

Recently, Chandy has proposed combining progress with stability; he writes $p \hookrightarrow q$ to denote that once p holds in the program, q will eventually hold and will continue to hold thereafter. The operator $\hookrightarrow$ has many pleasing properties including lhs strengthening, rhs weakening, infinite disjunction and transitivity. It is particularly interesting that $\hookrightarrow$ is finitely conjunctive. The notion of *wlt* appears in Knapp[20] and in Jutla, Knapp and Rao[17]. The latter paper includes the fixpoint characterization of *wlt* given in Section 6.2. The property (W5) in Section 14.1 is due to Singh[40]. Several varieties of completeness (and incompleteness) results have been established for *leads-to*. Jutla and Rao[18] contains an excellent exposition of what completeness means in this context. They argue that relative completeness in the sense of Cook[8] is all that we can hope for. Such completeness results appear in Cohen[7], Jutla and Rao[18], Knapp[21], Pachl[36], and Rao ([37] and [38]). The fact that the disjunction rule in the definition of *leads-to* is unnecessary for finite-state programs, is due to van de Snepscheut[44]. It has been exploited by Kaltenbach[19] in implementing an automatic verifier for finite state programs. The *leads-to* operator has been extended to probabilistic programs in [38]. Carruth[2] shows how the progress operators can be extended for stating and proving real-time progress properties. His extensions preserve the derived rules stated in this chapter.

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 24(4):181–185, 1985.

[2] Al Carruth. Real-time unity. Technical Report TR94-10, University of Texas at Austin, Austin, Texas, April 1994.

[3] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM*, 6(4):632–646, 1984.

[4] K. Mani Chandy and Jayadev Misra. How processes learn. *Journal of Distributed Computing*, 1:40–52, 1986.

[5] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[6] K. Mani Chandy and Jayadev Misra. *Developments in Concurrency and Communication, ed. C. A. R. Hoare*, chapter Proofs of Distributed Algorithms: An Exercise. University of Texas at Austin Year of Programming. Addison-Wesley, 1990.

[7] Ernie Cohen. *Modular Progress Proofs of Concurrent Programs*. PhD thesis, The University of Texas at Austin, August 1992.

[8] Stephen Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[9] E. W. Dijkstra. Solution of a problem in concurrent programming control. *C.ACM*, 8(9):569, 1965.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[11] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[12] R. W. Floyd. Assigning meanings to programs. In T. Schwartz, editor, *Mathematical Aspects of Computer Science (Proc. Sym. in Applied Math)*, volume 19, pages 19–32. Amer. Math. Soc., 1967.

[13] Nissim Francez. *Fairness*. Springer-Verlag, New York, 1986.

[14] J. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 66 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978. Also appears as *IBM Research Report RJ 2188*, August 1987.

[15] Joseph Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. Technical Report TR RJ 4421, IBM Almaden Research Center, 1989.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *C.ACM*, 12:476–580, 1969.

[17] C. S. Jutla, E. Knapp, and J. R. Rao. A predicate transformer approach to semantics of parallel programs. In *Proc. 8th ACM SIGACT/SIGOPS Symposium on Principles of Distributed Systems (PODC '89)*, pages 249–263, Edmonton, Alberta, Canada, 1989.

[18] Charanjit S. Jutla and J. R. Rao. A methodology for designing proof rules for fair parallel programs. Technical report, IBM T.J. Watson Research Center, 1994.

[19] M. Kaltenbach. Model checking for unity. Technical report, Univ. of Texas at Austin, 1994.

[20] E. Knapp. A predicate transformer for progress. *Information Processing Letters*, 33:323–330, 1989/90.

[21] E. Knapp. *Refinement as a Basis for Concurrent Program Design*. PhD thesis, The University of Texas at Austin, May 1992.

[22] Simon S. Lam and A. Udaya Shankar. Refinement and projection of relational specifications. In *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Plasmolen-Mook, The Netherlands*. LNCS Series, Springer-Verlag, May 1989.

[23] L. Lamport. Proving the correctness of multiprocess programs. *IEEE, Trans. on Software Engineering*, SE-3(2):125–143, March 1977.

[24] L. Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, 1990.

[25] L. Lamport. The temporal logic of actions. Technical Report SRC Research Report Number TR 79, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, December 1991.

[26] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In O. Kariv and S. Even, editors, *Proc. 8th ICALP*, volume 115 of *Lecture Notes in Computer Science*, Acre, Israel, July 1981. Springer-Verlag.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[28] J. Misra. Reasoning about networks of communicating processes. Presented at the Advanced Nato Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, La Colle-Sur-Loup, France, October 1984.

[29] J. Misra. A theorem about dynamic acyclic graphs. Notes on UNITY: 02–88, September 1988.

[30] J. Misra. A foundation of parallel programming. In Manfred Broy, editor, *Proc. 9th International Summer School on Constructive Methods in Computer Science*, volume F 55 of *NATO ASI Series*, pages 397–433, Marktoberdorf, Germany, July, 1988, 1989. Springer-Verlag.

[31] J. Misra. Auxiliary variables. Notes on UNITY: 15–90, July 1990.

[32] J. Misra. Equational reasoning about nondeterministic processes. *Formal Aspects of Computing*, 2(2):167–195, 1990.

[33] J. Misra. Soundness of the substitution axiom. Notes on UNITY: 14–90, March 1990.

[34] Jayadev Misra. Lecture notes on unity proof theory. available as Notes-on-UNITY, 1991-1993. Univ. of Texas.

[35] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[36] Jan Pachl. A simple proof of a completeness result for *leads-to* in the UNITY logic. *Information Processing Letters*, 41:35–38, 1992.

[37] J. R. Rao. On a notion of completeness for the *leads-to*. Notes on UNITY: 24–90, July 1991.

[38] J. R. Rao. *Building on the UNITY Experience: Compositionality, Fairness and Probability in Parallelism*. PhD thesis, The University of Texas at Austin, August 1992.

[39] B. A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[40] Ambuj K. Singh. A theorem relating *leads-to* and *unless*. Notes on UNITY: 04–88, December 1988.

[41] Mark G. Staskauskas. Formal specification and design of a distributed electronic funds-transfer system. *IEEE Transactions on Computers*, pages 1515–1528, Dec. 1988.

[42] Mark G. Staskauskas. *Specification and Verification of Large-Scale Reactive Programs*. PhD thesis, The University of Texas at Austin, May 1992.

[43] Mark G. Staskauskas. An experience in the formal verification of industrial software. *Comm. of the ACM*, (to appear).

[44] Jan van de Snepscheut. Personal communication.

[45] A. J. M. van Gasteren and G. Tel. On the proof of a distributed algorithm. *Information Processing Letters*, 35(6), 1990. Letter to the Editor.