South African Computer Journal
Number 22
March 1999

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels
Since 1992 there has been a biennial Winter School on Formal and Applied Computer Science (WOFACS) at the University of Cape Town. Each of these have resulted in a Proceedings volume: SACJ 9, 13, 19, and the volume you hold in your hand right now. All WOFACS events have had more or less the same structure. A group of eminent academics come to Cape Town during the winter vacation, and each of them offers, over a 2-week period, a course of 10 lectures on a particular topic. These short courses are pitched at about Honours level, and have some evaluation mechanism built in: short tests, or exercises, or assignments. At a student’s request, and by arrangement with the Head of Department at his/her home institution, these courses can then be offered as part of the student’s Honours degree. In this way WOFACS makes a contribution to beginning postgraduate studies on a wide geographical front. Typically such an event would attract students and young staff members not only from across South Africa, but also from many sub-Saharan African countries. Each WOFACS was organised by the UCT Laboratory for Formal Aspects and Complexity in Computer Science (FACCSLab).

WOFACS 98 had a distinctly international flavour. The entire event was, in fact, three things at once (which explains the somewhat complicated title at the top of this page). Besides being, by our reckoning, the 4th WOFACS, it was also the third in a series of outreach offerings of IFIP Working Group 2.3 on Programming Methodology. All the speakers were from WG2.3, and the entire event was built around the topic of programming methodology. Thirdly, the event was also an offering of the International Institute for Software Technology, situated at the United Nations University in Macao. The three role players, FACCSLab, WG2.3 and UNU/IIST, shared an agenda of trying to service specifically possible participants from disadvantaged communities and other African countries, and the UNU/IIST made available some generous grants for this purpose. In keeping with the tradition of these events there were no course fees: except for a fairly modest registration charge WOFACS has always been a free service to the community.

The speakers at WOFACS 98 and their topics were:

- Prof Dines Bjørner, Technical University Denmark: *Domains and Requirements, Software Architectures and Program Organisation*.
- Prof David Gries, Cornell University: *Logic as a Tool*.
- Prof Michael Jackson: *Problem Frames and Principles of Description*.
- Prof Jayadev Misra, University of Texas: *Toward an Applied Theory of Concurrency*.
- Dr Carroll Morgan, Oxford: *Predicate Transformers and Probabilistic Programs*.

Professor Gries’ course was regarded as foundational, and recommended to all participants. It was offered during the first week only, at double tempo, thus giving the other four courses the opportunity to make use of concepts and techniques introduced there. Each of the speakers made available a Course Reader of their material. These were printed and bound before the event, and were handed out to participants upon registration. WOFACS 98 was attended by more than 60 participants, inter alia from Angola, Malawi, the Congo, Gabon, Cameroon, Malawi and Uganda.

From South African Universities we had representation, besides UCT, also from the University of Stellenbosch, the Transkei, the Qwa-qwa branch of the University of the North, the Witwatersrand, Pretoria, RAU, the North-West, Vista, UNISA, the Mangosothu Technikon and the Eastern Cape Technikon. Cape Town weather can be pretty stormy in July, but there were sufficiently many beautifully clear winter days to allow participants the opportunity to do some
sightseeing and exploring, after lectures or over the weekend.

We are grateful to our financial sponsors, and pleased to acknowledge their contributions.

• UNU/IIST.
• The Foundation for Research Development.
• The Chairman's Fund of Anglo American.

We thank the University of Cape Town for the use of its premises and facilities. We are particularly grateful to the 5 speakers, who put a lot of thought and preparation into their lectures, and tackled with great success the difficult task of conveying state-of-the-art material to a heterogeneous audience. It is only fair that specific thanks should be given to Carroll Morgan, whose idea it was in the first place to have a combined event, and to Dines Bjorner, who kickstarted the fundraising campaign. Finally, I would like to add my personal thanks to my colleagues and staff who worked so hard behind the scenes to make a success of WOFACS 98.
A Logic for the Design of MultiProgramming Systems

Jayadev Misra
Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712
misra@cs.utexas.edu

Abstract
This paper presents a short introduction to the UNITY logic, a fragment of linear temporal logic. The logic was designed to specify safety and progress properties of reactive systems. A version of the UNITY logic appears in [1]. There have been several changes in this logic since then; a full account is available at http://www.cs.utexas.edu/u.sers/psp/newunity.html, and the essential ideas have appeared in [5, 4].

Keywords: Safety, Progress, Invariant, Stable, Fixed point, Liveness, Fairness, Transient, Ensures, Leads-To
Computing Review Categories: D.2.4,F.3.1

1 Introduction

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. 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.

In the second part of this paper, 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.

A logic for UNITY-program composition appears in [1, 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 [3, chapters 5,6] for details.

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[1] in which a program counter is shown explicitly (as a part of the system state). 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. 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.

The primary operator for expressing safety properties is \(\text{co}\) (for constrains). This operator is designed to facilitate reasoning by induction on the number of computation steps. Most safety properties that arise in practice are succinctly expressed using \(\text{co}\). Additionally, \(\text{co}\) has simple manipulation rules which permit easy deduction.
of new properties from the given ones. 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 of this paper. The derived rules for co—the main ones being universal conjunctivity and universal disjunctivity—are given in Section 3. This section also describes the Substitution Axiom and the Elimination Theorem, which are essential devices for constructions of succinct proofs. Safety properties for a number of small systems are formalized and manipulated.

2 The Basic Safety Operator: 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 up to (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 \( \neg p \land q \) holds.

We define

\[
p \ co q \triangleq (\forall t :: \{p\} t \{q\})
\]

where \( t \) is an action of the system (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[2] is

\[
p \ co q \triangleq (\forall t :: p \Rightarrow wlp.t.q)
\]

This formulation allows us to establish the derived rules for co in Section 3.1 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,

\[
p \land q \ co r \land s
\]

is to be interpreted as

\[
(p \land q) \ co (r \land s)
\]

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
   - \( x \) cannot be decreased if it is zero.

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

   \[
x = 0 \ co x \geq 0
\]

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 \( m \)

   \[
x = m \ co x \geq m
\]

Here, \( m \) is a free variable and 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

\[
x \geq m \ co x \geq m
\]

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 Section 3.1.

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

$p \text{ stable} \equiv p \text{ co } p$

$p \text{ invariant} \equiv \text{ initially } p \text{ and } p \text{ stable}$

$e \text{ constant} \equiv (\forall k :: e = k \text{ stable})$

, where $e$ is an expression and $k$ is a free variable of the same type as $e$.

From above, $p \text{ stable}$ means that once $p$ is $\text{true}$ it remains $\text{true}$ forever, because $p \text{ co } p$ is

$(\forall s :: \{p\} s \{p\})$

which means that no transition falsifies $p$. Predicates $\text{false}$ and $\text{true}$ are stable in any program. Stable predicates are ubiquitous, for example: The system is deadlocked, and the number of messages sent along a channel exceeds a certain value. We will see many instances of stable predicates in Section 4.

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

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. The reader should verify that the familiar constants—3, $\text{true}$, ‘HELLO’—are constants according to our definition.

### 2.1 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 that is discussed in Chapter 4. The well-known phrase, “program is deadlock-free,” means that $\neg FP$ is always $\text{true}$; then, it is always possible to change the program state.

### 3 Derived Rules for co

#### 3.1 Basic Rules

All of these rules follow directly from the facts about the predicate transformer, $wlp$. Here, $p, q, p', q', r$ are arbitrary predicates.

- $\text{false co } p$
- $p \text{ co } \text{true}$
- $(\text{conjunction}; \text{disjunction})$

\[
\frac{p \text{ co } q, p' \text{ co } q'}{p \land p' \text{ co } q \land q'}
\frac{p \lor p' \text{ co } q \lor q'}
\]

The conjunction and disjunction rules follow from the conjunctivity and monotonicity properties of $wlp[2]$ 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 $r \text{ co } \text{true}$ and disjoining $\text{false co } r$, respectively, to $p \text{ co } q$—we obtain

- (lhs strengthening)

\[
\frac{p \text{ co } q}{p \land r \text{ co } q}
\]

- (rhs weakening)

\[
\frac{p \text{ co } q}{p \text{ co } q \lor r}
\]
Also, co is transitive

\[ p \land q, q \land r \quad \implies p \land r \]

This is because \( q \Rightarrow r \) from \( q \land r \); then rhs of \( p \land q \) can be weakened to \( p \land r \). 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., \( p \land p \) does not always hold) nor are we allowed to deduce a contrapositive (\( \neg q \land \neg p \) cannot be deduced from \( p \land q \)).

**Proof Format for co** Since the lhs of a co-property can be strengthened and its rhs can be weakened, we can write a deduction in the following format to establish a co-property, say \( p \land q \).

\[
\begin{align*}
  \Leftrightarrow & \quad \text{justification} \\
  \Rightarrow & \quad \vdots \\
  \Rightarrow & \quad r \\
  \Rightarrow & \quad \text{justification} \\
  \Rightarrow & \quad s \\
  \Rightarrow & \quad \text{justification} \\
  \Rightarrow & \quad t
\end{align*}
\]

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.

### 3.2 Rules for the special cases

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

(\text{stable conjunction, stable disjunction})

\[ p \land q, r \quad \text{stable} \]
\[
\begin{align*}
  p \lor r \land q \lor r \\
  p \lor q \land r
\end{align*}
\]

A special case of the above is,

\[ p \text{ stable}, q \text{ stable} \]
\[
\begin{align*}
  p \lor q \text{ stable} \\
  p \land q \text{ stable}
\end{align*}
\]

Similarly,

\[ p \text{ invariant}, q \text{ invariant} \]
\[
\begin{align*}
  p \land q \text{ invariant}
\end{align*}
\]

(\text{constant formation})

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

### 3.3 Substitution Axiom

An invariant may be replaced by true, and vice versa, in any property of a program.

Substitution axiom allows us to deduce properties that we cannot deduce directly from the definition. For instance, given that \( p \land q \) and that \( J \) invariant, we can conclude

\[ p \land J \land q, p \land q \land J, p \lor J \land q \land J, p \lor \neg J \land q \land J, \]

etc.

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

\[
\begin{align*}
  p \text{ invariant}, & \quad \text{given} \\
  p \land q \text{ invariant}, & \quad p = p \land q, \text{ since } p \Rightarrow q \\
  q \text{ invariant}, & \quad \text{replace } p \text{ by true using the substitution axiom}
\end{align*}
\]
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."

3.4 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 $p \rightarrow q$ we can deduce, for program variable $x$ and free variable $m$,

- $p \land x = m \rightarrow q$ and,
- $p \rightarrow q \lor x \neq m$

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 $p[x := m]$ 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,

$$p = (\exists m : p[x := m] : x = m)$$

**Theorem (Elimination Theorem)**

$x = m \rightarrow q$, where $m$ is free

\[ p \rightarrow (\exists m :: p[x := m] \land q) \]

**Proof:**

- $x = m \rightarrow q$, premise
- $p[x := m] \land x = m \rightarrow p[x := m] \land q$, stable conjunction with $p[x := m]$
- $(\exists m :: p[x := m] \land x = m) \rightarrow (\exists m :: p[x := m] \land q)$, disjunction over all $m$
- $p \rightarrow (\exists m :: p[x := m] \land q)$, simplifying the lhs

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 $u, v = m, n \rightarrow u, v = m, n \lor (m > n \land u, v = m - 1, n)$

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

- $u \geq v$
- $\rightarrow (\exists m, n :: (m \geq n \land u, v = m, n) \lor (m \geq n \land m > n \land u, v = m - 1, n))$
- $\Rightarrow (\exists m, n :: u \geq v \lor u \geq v)$
- $\Rightarrow (u \geq v)$

4 Safety Properties in Program Design

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 Section 3 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.

4.1 Nonoperational Descriptions of Algorithms

It is often preferable to describe an algorithm not by a program text but by its properties. There are several advantages to the latter approach: (1) we can express a family of algorithms by one set of properties, because many implementation details can be ignored while writing the properties, (2) it is usually easier to prove facts
about an algorithm starting with its properties than from its code, and (3) it is often easier to understand an
algorithm from its properties than from its code. We choose a very simple problem from sequential programming—
computing the maximum of a set of numbers—to illustrate these aspects. Section 4.2 contains a small exercise
of this nature.

The typical algorithm to compute the maximum value, \( v \), of a nonempty finite set, \( S \), of integers is to (1)
initially, assign a very small value, \(-\infty\), say, to \( v \) and (2) then scan the elements of \( S \) in some fixed order updating
\( v \) whenever the scanned element has a larger value. Instead of expressing this algorithm in the notation of a
programming language, we describe it by its properties, by focusing on the allowable changes to \( v \).

Using a free variable \( m \) that ranges over integers and \(-\infty\),

\[
\begin{align*}
\text{initially } v &= -\infty \\
v = m \co v = m &\lor (v \in S \land v > m)
\end{align*}
\]

The initial condition is as described above. The \( \co \)-property says that \( v \) is only changed to a higher value that
is also in \( S \). This description ignores the order in which the elements of \( S \) are scanned, leaving open a number
of possibilities for implementation (one of which we discuss later in this subsection). Now, we can deduce several
properties.

- \( v \) is nondecreasing, i.e., for any \( n \)
  \[
  v \geq n \quad \text{stable}
  \]

- \( v \) never exceeds the maximum of \( S \)
  \[
  v \leq M \quad \text{invariant}, \quad \text{where } M = (\max x : x \in S : x)
  \]

- \( v \in S \) stable

The proofs of all these properties appeal to the elimination theorem; we show one below.

\[
\begin{align*}
v \leq M \quad \text{invariant:}
\end{align*}
\]

\[
\begin{align*}
\text{initially } v &\leq M \\
\text{from } v = -\infty
\end{align*}
\]

To show that \( v \leq M \) stable,

\[
\begin{align*}
v \leq M \\
\co \{\text{elimination theorem on: } v = m \co v = m \lor (v \in S \land v > m)\}
\end{align*}
\]

\[
\begin{align*}
(\exists m :: (m \leq M \land v = m) \lor (m \leq M \land v \in S \land v > m))
\Rightarrow \{\text{weakening}\}
\end{align*}
\]

\[
\begin{align*}
(\exists m :: v \leq M \lor v \in S)
\Rightarrow \{\text{simplifying}\}
\end{align*}
\]

\[
\begin{align*}
v \leq M \lor v \in S
\Rightarrow \{v \in S \Rightarrow v \leq M\}
\end{align*}
\]

\[
\begin{align*}
v \leq M
\end{align*}
\]

Note that we can't yet prove that \( v \) will eventually equal \( M \); wait until we develop the theory of progress in
Chapter 4.

A refinement of this algorithm is the following. The set \( S \) is represented by an array \( A[0..] \); the elements
of \( A \) are scanned in the order \( A[0], A[1] \ldots \) updating \( v \) appropriately. We again describe this algorithm by its
properties, i.e., by the way its variables are manipulated. Let \( s \) be the index into \( A \) upto (but not including)
which the elements have been scanned. Then,

\[
\begin{align*}
\text{initially } v, s &= -\infty, 0 \\
v, s = m, k \co v, s = m, k &\lor v, s = \max(m, A[k]), k + 1
\end{align*}
\]

where \( m \) ranges over integers and \(-\infty\), and \( k \) ranges over the indices of \( A \). We can prove that \( v \) is the maximum
over the scanned segment, i.e.,

\[
\begin{align*}
v &= (\max i : 0 \leq i < s : A[i])
\end{align*}
\]

Initially the above holds, since \(-\infty = (\max i : 0 \leq i < 0 : A[i])\). We now prove the stability of \( v = (\max i : 0 \leq i < s : A[i])\).
The remaining proof obligation is, rewriting (CMT4),
\[ v = (\max i : 0 \leq i < s : A[i]) \]
\[ \text{co} \{ \text{elimination theorem} \} \]
\[ (\exists m, k :: [m = (\max i : 0 \leq i < k : A[i]) \land v, s = m, k] \]
\[ \lor [m = (\max i : 0 \leq i < k : A[i]) \land v, s = \max(m, A[k]), k + 1] \]
\[ \} \]
\[ \Rightarrow \{ \text{arithmetic} \} \]
\[ v = (\max i : 0 \leq i < s : A[i]) \]
Note that for this trivial problem the formal proof is trivially simple.

4.2 Common Meeting Time

This problem has been discussed in Chandy and Misra[1, Section 1.4]. The purpose of this example, much like the one in Section 4.1, 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; \( g(t) \) is similarly defined. Time \( t \) is acceptable to \( F \) if and only if \( f(t) = t \). Time \( t \) is a common meeting time if and only if it is acceptable to both \( F \) and \( G \), i.e., \( f(t) = t \land g(t) = t \). 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 [1]. 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 \( m, n \)
\[ m \leq n \Rightarrow f(m) \leq f(n) \]  
\[ m \leq n \Rightarrow g(m) \leq g(n) \]  
\[ \text{CMT1} \]

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 8.2. 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.
\[ \text{initially } t = 0 \]  
\[ t = m \text{ co } t \leq \max(f(m), g(m)) \]  
\[ \text{CMT2} \]

Let \( \text{com}(n) \) denote that \( n \) is a common meeting time, i.e., \( f(n) = n \land g(n) = n \). We prove, from CMT1-CMT3, that \( t \) exceeds no common meeting time, i.e., for any \( n \)
\[ \text{com}(n) \Rightarrow t \leq n \]  
\[ \text{CMT4} \]

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),
\[ (\neg \text{com}(n) \lor t \leq n) \text{ stable} \]  
\[ \text{CMT5} \]
which we proceed to prove.
\[ \neg \text{com}(n) \lor t \leq n \]
\[ \text{co} \{ \text{elimination theorem on (CMT3) where } p \text{ is } \neg \text{com}(n) \lor t \leq n \} \]
\[ (\exists m :: (\neg \text{com}(n) \lor m \leq n) \land t \leq \max(f(m), g(m))) \]
\[ \equiv \{ \text{write the disjunction as } \neg \text{com}(n) \lor (\text{com}(n) \land m \leq n) \text{ and expand } \} \]
\[ (\exists m :: [\neg \text{com}(n) \land t \leq \max(f(m), g(m))] \lor [\text{com}(n) \land m \leq n \land t \leq \max(f(m), g(m))] \]
\[ \Rightarrow \{ \text{the first disjunct implies } \neg \text{com}(n), \text{ the second disjunct implies (see below) } t \leq n \} \]
\[ (\exists m :: (\neg \text{com}(n) \lor t \leq n)) \]
\[ \Rightarrow \{ \text{predicate calculus } \} \]
\[ \neg \text{com}(n) \lor t \leq n \]
This establishes (CMT5). Now we prove the result claimed in the above proof:
\[ \text{com}(n) \land m \leq n \land t \leq \max(f(m), g(m)) \Rightarrow t \leq n \]
The strategy given by (CMT3) is quite general. It subsumes the following strategies.

\[
\begin{align*}
t & = m \\
\text{CMT0:} & \quad t := m \text{ or } t := \max(f(t), g(t)) \\
\text{CMT1:} & \quad t := \max(f(t), g(t)) \\
\text{CMT2:} & \quad t := t + 1 \quad \text{if } t + 1 \leq \max(f(t), g(t))
\end{align*}
\]

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

\[
t = m \text{ and } m \leq t \leq \max(f(m), g(m))
\]

(CMT3')

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

\[
\begin{align*}
n & \leq f(n) \land n \leq g(n)
\end{align*}
\]

(CMT0)

The reader can prove (by applying elimination theorem to CMT3') that \( t = n \land \text{com}(n) \) 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 Introduction to Progress

Safety properties allow 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 that causes no state change—a program consisting of a \texttt{skip} statement, for instance—satisfies all the safety properties. Thus, safety properties alone are insufficient as a basis of program design. 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.

In this section, we study a class of properties known as \textit{progress}. Progress properties state that "the program does some good." For instance, "I press the switch and then the light is on" is an example of a progress property. 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.

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 \( q \) 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, however, that this is a far cry from the absolute performance guarantees that we sought. 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.

6 Transient Predicate

A predicate is transient if it is guaranteed to be falsified by execution of a single (atomic) transition. 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 transition systems; such systems can be represented by a UNITY program[1]. We make the following assumptions about such systems.

- There is at least one transition. Recall that \( \text{skip} \) is included as an action in every system and, hence, this requirement is always met. (We do not show \( \text{skip} \), explicitly, in the programs we write.)
- Each transition terminates, i.e., if the transition is started in any state where it is enabled, it completes in finite time. It is easy to check termination for simple transitions, such as the ones that can be represented by assignment of values to variables. For more intricate transitions, for instance, where a transition is an entire program, the methods of this chapter have to be applied recursively to first prove terminations of the individual transitions and then deduce progress properties of the system.
- An execution is an infinite sequence of transitions executions. In an execution each statement is executed infinitely often. Executing a statement in a state where its guard is \( \text{false} \) causes no state change.

For a terminating transition \( s \), we have the Law of Excluded Miracle[2]

\[
\{p\} s \{\text{false}\}
\]

\[\neg p\]

i.e., the postcondition of a transition is \( \text{false} \) only if the precondition is \( \text{false} \). Using the substitution axiom, this can be interpreted as “the resulting state of a transition is unreachable only if the transition is started in an unreachable state.”

Define,

\[ p \text{ transient} \Delta (\exists s :: \{p\} s \{\neg p\}) \]

where \( s \) is over all transitions in the system.

The following operational argument shows that eventually \( \neg p \) holds given that \( p \) is transient. Let \( t \) be a transition 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 \( \{p\} t \{\neg p\} \), following the execution of \( t \), \( \neg p \) holds. (Note that the execution of \( t \) is assumed to terminate.)

7 ensures, leads-to

7.1 ensures

The informal meaning of \( p \text{ ensures} q \) (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) transition which guarantees to establish \( q \) starting in any \( p \)-state. Formally,

\[ p\ en q \triangleq (p \land \neg q \text{ co} p ) \\

\]
Consider a state in which \( p \) holds and \( q \) does not. Since \( p \land \neg q \) is transient, it is eventually falsified. From the co-property, whenever \( (p \land \neg q) \) is falsified, \( (p \lor q) \) holds. Hence, whenever \( (p \land \neg q) \) is falsified, \( \neg(p \land \neg q) \land (p \lor q) \), i.e., \( q \), holds.

### 7.2 leads-to

The informal meaning of \( p \rightarrow q \) (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 \( \text{ensures} \), that \( p \) remains true until \( q \) holds.

The definition of \( p \rightarrow q \), given below, is recursive. For the basis, we deduce \( p \rightarrow q \) from \( p \land q \). The transitivity rule \((p \rightarrow r \text{ can be deduced from } p \rightarrow q \text{ and } q \rightarrow r)\) is justified as follows. From \( p \rightarrow q \), once \( p \) holds \( q \) will hold, and from \( q \rightarrow r \), once \( q \) holds \( r \) will hold. Therefore, once \( p \) holds \( r \) will hold. The disjunctivity rule (given \( p \rightarrow q \), for all \( p \) in a certain set, deduce \( P \rightarrow q \) where \( P \) is the disjunction of the predicates in that set) is justified, because any state that satisfies \( P \) also satisfies some predicate, \( p \), in the given set and hence, starting from this state \( q \) will eventually be established, from \( p \rightarrow q \). An alternative definition of \( \text{leads-to} \) that eliminates \( \text{en} \) appears in Exercise 4. Exercise 19 shows that the transitivity and the disjunction rules may be combined into a single rule.

\[
\begin{align*}
\text{(basis)} \\
& p \land q \\
\hline
\text{(transitivity)} \\
& p \rightarrow q, q \rightarrow r \\
& \hline \\
& p \rightarrow r \\
\text{(disjunction)} \\
&(\forall p: p \in S : p \rightarrow q) \\
&(\exists p: p \in S : p) \rightarrow q
\end{align*}
\]

As usual, the substitution axiom applies.

The form of this definition—using inference rules—differs from the way we defined \( \text{co} \), transient and \( \text{en} \). The current definition is recursive, and \( \rightarrow \) can be understood as an extreme solution (a least fixpoint) of an equation. 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 derived rules for \( \rightarrow \) using structural induction over its definition (see Sections 4.4.4, 4.6.1).

**Note:** The only mention of \( \text{en} \) is in the basis rule. The premise of that rule could be replaced by the definition of \( p \land q \rightarrow p \land \neg q \land \text{co} \land (p \lor q) \land (p \land \neg q) \text{ transient} \) — thus eliminating \( \text{en} \) from our theory.

**A Note on Quantification:** The range of quantification involving \( \rightarrow \) (or even \( \text{en} \)) can be manipulated as follows.

\[
(\forall k : r : p \rightarrow q) \equiv (\forall k : p \land r \rightarrow q)
\]

**Examples:** In the following, variables \( x, y \) are integers and \( S, T \) are finite sets of integers.

1. A hungry philosopher eats. Let \( h, e \) denote, respectively, that a particular philosopher is hungry or eating.

\[
h \rightarrow e
\]

2. Variable \( x \) changes eventually. For every integer \( m \),

\[
x = m \rightarrow x \neq m
\]

This can be written equivalently as (see Exercise 7d)

\[
\text{true} \rightarrow x \neq m
\]

SART/SACJ, No 22, 1999
3. Variable $x$ grows without bound. For every integer $m$

$$\text{true} \mapsto x > m$$

Note that the above property is an abbreviation for

$$(\forall m :: \text{true} \mapsto x > m).$$

It should not be confused with

$$\text{true} \mapsto (\forall m :: x > m)$$
which happens to be nonsense.

4. Every integer is added to $S$ eventually. For every integer $m$,

$$\text{true} \mapsto m \in S$$

It is not possible to conclude from the above that $S$ grows eventually, because there is no restriction on items being removed from $S$. We impose such a restriction: For any fixed set of integers $U$,

$$S \supseteq U \text{ stable}$$

It can now be shown that $S$ grows without bound. However, it does not follow from the above two properties that $S$ will eventually include all integers. A possible scenario is that initially $S$ is empty and number $m$ is added to $S$ at the $m^{th}$ step, $m \geq 0$, (and no number is ever removed from $S$). Then, $S$ becomes arbitrarily large though it remains finite after a finite number of steps.

5. Variables $x, y$ do not retain their values forever if they differ. For all integers $m, n$

$$x, y = m, n \land m \neq n \Rightarrow x, y \neq m, n$$

or, equivalently, $(\forall m, n : m \neq n : x, y = m, n \rightarrow x, y \neq m, n)$.

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

6. Every element common to $S, T$ is eventually removed from both sets.

$$m \in (S \cap T) \Rightarrow m \notin (S \cup T)$$

It cannot be deduced from the above that $S, T$ will eventually become disjoint.

7. Predicate $p$ holds infinitely often.

$$\text{true} \mapsto p$$

or, equivalently (see Exercise 7d)

$$\neg p \mapsto p$$

8. If $p$ remains $\text{true}$ forever then $q$ holds eventually.

Another way of expressing this property is to say that eventually either $p$ is $\text{false}$ or $q$ holds:

$$\text{true} \mapsto \neg p \lor q$$

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

$$(\text{true} \mapsto p) \Rightarrow (\text{true} \mapsto q)$$

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.

10. 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, $FP$.

$$\text{initial-condition} \mapsto FP$$
7.3 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 is 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 inference rules for $\rightarrow$. 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 [1, chapter 3].

Lightweight Rules

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

The remaining lightweight rules have counterparts for co.

- (lhs-strengthening, rhs-weakening) $\frac{p \iff q}{p' \land p \Rightarrow q}$, $\frac{p \Rightarrow q \lor q'}{p \Rightarrow q \lor s}$

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

$\frac{(\forall i :: p_i \Rightarrow q_i)}{\exists i :: p_i \Rightarrow (\exists i :: q_i)}$

- (cancellation) $\frac{p \Rightarrow q \lor r, r \Rightarrow s}{p \Rightarrow q \lor s}$

We deduce from the implication rule that for any predicate $p$, $p \Rightarrow p$ and $false \Rightarrow p$.

The lhs-strengthening and the rhs-weakening rules—also valid for co-properties—are used extensively in proofs. The disjunction rule, given previously, is slightly more general than the one given here (for an empty set of predicates, the previous rule yields $false \Rightarrow q$ for any $q$, and the current rule yields $false \Rightarrow false$). We will not distinguish the two rules by name; it should be obvious in any application which rule is being considered. The cancellation rule played little role in manipulating the co-properties; it is, however, used heavily in progress proofs. The cancellation rule reduces to transitivity when $q$ is $false$. Note that there is no conjunction rule for $\Rightarrow$ analogous to the one for co (see Exercise 12c).

Heavyweight Rules

- (impossibility) $\frac{p \Rightarrow false}{\neg p}$

- (PSP) $\frac{p \land s \Rightarrow (q \land r) \lor (\neg r \land s)}{p \land s \Rightarrow p \lor s}$

- (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 $W$.

$\frac{(\forall m :: p \land M = m \Rightarrow (p \land M < m) \lor q)}{p \Rightarrow q}$

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 $i$.

$\frac{(\forall i :: p_i \Rightarrow q_i \lor b)}{(\forall i :: p_i) \Rightarrow (\forall i :: q_i) \lor b}$
Special Issue

The impossibility rule says that a state satisfying false is reachable only from an unreachable state (read the consequent of the rule, "¬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 co s—and a progress proof—establishing p q—which are then combined. This rule is so important that it should be memorized before attempting serious progress proofs.

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 q holds.) Some common examples of well-founded relations are: less-than relation over positive integers (or natural numbers), lexicographic order over tuples (where the corresponding items in all tuples are totally ordered), proper prefix or proper subsequence relation over finite sequences and proper 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 co, analogous to the rule for co-properties (Exercise 12c). 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. Exercise 16 asks you to show that the rule is not valid for an infinite pair of predicates, pi and q; certain generalizations of this rule appear in Exercise 17.

8 Progress Properties in Program Design

8.1 Nonoperational Descriptions of Algorithms

We consider the algorithm for computing the maximum of a nonempty set of numbers, S, as described in Section 4.1. The safety properties that we had postulated are as follows. Here, v is the variable in which the maximum is being computed and m is any integer.

\[ \text{initially } v = -\infty \] \hspace{1cm} (ND1)
\[ v = m \iff v = m \lor (v \in S \land v > m) \] \hspace{1cm} (ND2)

We derived a number of safety properties including

\[ v \leq M \] \hspace{1cm} (ND3)

where M is the maximum in S, i.e., \( M = \max x : x \in S : x \). Now, we add the following progress property, where x is free.

\[ m \in S \iff v \geq m \] \hspace{1cm} (ND4)

which says that eventually v is at least m for any m in S. It is easy to establish that v will equal M eventually.

Proof of true \( v \Rightarrow v = M \):

\[ m \in S \iff v \geq m \] \hspace{1cm} , (ND4)
\[ M \in S \iff v \geq M \] \hspace{1cm} , instantiating m by M
true \( v \geq M \) \hspace{1cm} , substitution axiom on the lhs (S is nonempty)
true \( v = M \) \hspace{1cm} , substitution axiom, conjoining (ND3) on the rhs

A refinement of this algorithm was considered in Section 4.1. Let S be represented by an array, A, and s be an index into A. We had

\[ \text{initially } v, s = -\infty, 0 \text{ and} \]
\[ v = (\max i : 0 \leq i < s : A[i]) \] \hspace{1cm} (ND5)
\[ v = (\max i : 0 \leq i < s : A[i]) \] \hspace{1cm} (ND6)

We will show that v acquires the maximum value in A given that the index s increases eventually as long as the end of the array is not reached. Formally, for all \( k, 0 \leq k < N \) (where A has \( N, N \geq 0 \), elements), assume

\[ s = k \iff s = k + 1 \] \hspace{1cm} (ND7)

Proof of initial condition \( v \Rightarrow v = (\max i : 0 \leq i < N : A[i]) \):

\[ s \leq N \iff s = N \] \hspace{1cm} , from (ND7) (see Exercise 14a)
\[ s = 0 \iff s = N \] \hspace{1cm} , strengthen lhs of the above using \( N \geq 0 \)
\[ s = 0 \iff v = (\max i : 0 \leq i < N : A[i]) \] \hspace{1cm} , conjoin (ND6) to rhs using substitution axiom.

\[ \text{initial condition } v = (\max i : 0 \leq i < N : A[i]) \] \hspace{1cm} , strengthen lhs using (ND5)
8.2 Common Meeting Time

The Common Meeting Time problem was discussed in Section 4.2. 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 numbers $m, n$

$$m \leq n \Rightarrow f(m) \leq f(n)$$
$$m \leq n \Rightarrow g(m) \leq g(n)$$

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

- Initially $t = 0$ (CMT2)
- $t = m \Rightarrow t \leq \max(f(m), g(m))$ (CMT3)

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

$$\text{com}(n) \Rightarrow t \leq n$$ (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.

$$\neg \text{com}(t) \land t = m \Rightarrow t > m$$ (CMT6)

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

$$(\exists n :: \text{com}(n)) \Rightarrow \text{com}(t)$$ (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 $m, n$

- $\neg \text{com}(t) \land t = m \Rightarrow t > m$, rewrite (CMT6)
- $\text{com}(t) \land t = m \Rightarrow \text{com}(t)$, implication
- $t = m \Rightarrow t > m \lor \text{com}(t)$, disjunction of the above two
- true $\Rightarrow t > n \lor \text{com}(t)$, induction on integers
- $\text{com}(n) \Rightarrow (\text{com}(n) \land t > n) \lor \text{com}(t)$, stable conjunction with $\text{com}(n)$
- $\text{com}(n) \Rightarrow \text{com}(t)$, use CMT4 to cancel the first disjunct in the rhs
- $$(\exists n :: \text{com}(n)) \Rightarrow \text{com}(t)$$, disjunction over all $n$.

This proof is invalid if $t$ is of type real because the induction step in the above proof is then invalid. The reader can construct a counterexample to (CMT7) 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 $\text{com}(t)$ 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 numbers $n$

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

then the programs (P1,P2) in Section 4.2 satisfy

$$\neg \text{com}(t) \land t = m \Rightarrow t > m$$

and, hence, CMT6 as well.

9 Synopsis

There is ample reason to believe that $\text{co}$ would be adequate for expressing the usual kinds of safety properties for reactive systems. The manipulation rules for $\text{co}$ are simple and effective. The examples in the paper—particularly, common meeting time (5.1)—were handled by extremely concise proofs; it is difficult to see how intuitive reasoning could be any cheaper! Although our examples often used UNITY-style programs, the theory is applicable to any action system.

The definition of leads-to and the promulgation of its manipulation rules 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. 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.

Acknowledgements

This material is based in part upon work supported by the National Science Foundation Awards CCR-9504190 and CCR-9803842.
References


Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal’s readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines:

• Use wide margins and $1\frac{1}{2}$ or double spacing.
• The first page should include:
  • the title (as brief as possible)
  • the author’s initials and surname
  • the author’s affiliation and address
• an abstract of less than 200 words
• an appropriate keyword list
• a list of relevant Computing Review Categories
• Tables and figures should be numbered and titled.
• References should be listed in alphabetic order of the (first) author’s surname, and should be cited in the text according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page, http://www.cs.up.ac.za/sacj which gives a number of examples.

SACJ is produced using the $\LaTeX$ document preparation system, in particular $\LaTeX$ 2e. Previous versions were produced using a style file for a much older version of $\LaTeX$, which is no longer supported. Please see the website for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs

Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.