A Proof System for Ada* Tasks

H. BARRINGER AND I. MEARNS ·
Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL

An axiomatic proof system for reasoning about the basic Ada tasking constructs is presented. The proof system is based on an earlier paper by the same authors and deals with safety properties of concurrent Ada programs including freedom from deadlock. The system has been proved to be sound and relatively complete against a denotational semantics for the constructs considered.

Received March 1985

1. INTRODUCTION

In an earlier paper,[11] we presented axioms and proof rules for reasoning about concurrent Ada programs. As we stated in that paper, we submitted our work for publication before the proof system had been formally checked for soundness and relative completeness; we repeat our justification for doing this in appendix B. However, after the paper appeared, we discovered that the system did have some errors and omissions and, therefore, a complete revision of Ref. 11 is presented in this paper.

The organisation of this paper is as follows. Section 2 acknowledges the inspiration for our work and contains some very simple examples of proofs. Section 3 is a formal presentation of the basic (partial correctness) proof system, and section 4 gives a non-trivial example of its use. Section 5 extends the proof system to deal with deadlock, and section 6 outlines the approach taken to prove it sound and relatively complete. Section 7 consists of a comparison with another Ada tasking proof system, and some conclusions are drawn in section 8. Appendix A contains a very brief survey of program proof systems in general, and a critique of our original system is given in Appendix B.

We assume that the reader is familiar with the Ada language reference manual;[17] a more readable introduction to the syntax and informal semantics of Ada is, for example, Barnes.[7] Readers unfamiliar with CSP (Communicating Sequential Processes), see below, are referred to Hoare.[27]

2. SIMPLE EXAMPLES

The basis of our proof system is the CSP proof system of Apt, Francez and de Roever[5] (hereafter referred to as AFdR), with its central notion that concurrently executing processes *cooperate* to achieve some overall result. To prove that this result is reached one first deduces the effect of each process executing in isolation and then considers the inter-process communications; if these satisfy the definition of cooperation then one may conclude that the overall result is the conjunction of the individual processes' post-conditions and a *global invariant*, GI. (See section 3.1 for full definitions.)

The proof system follows the axiomatic method of Hoare; references for readers unfamiliar with Hoare-style proof systems and associated terminology are given in appendix A. In order to give a flavour of the style of proofs, the rest of this section consists of some very simple examples.

First, consider an isolated task body consisting of a single assignment statement, $x := x+1$. If x is initialised to zero, then upon termination of the task, x clearly has the value one, and this may be proved by using Hoare's 'backward assignment axiom' ($\{p[e/x]\}\ x := e\ \{p\}$). This axiom states that for the formula denoted by p to be true after execution of the assignment $x := e$, the formula obtained by replacing every free occurrence of x in p by e, $p[e/x]$, must have been true just before execution of the assignment. Thus, to achieve $x = 1$ after execution of $x := x+1$, we require that $x = 1[x+1/x]$, i.e. $x+1 = 1$, is true before execution. Now, as $x = 0$ implies $x+1 = 1$, we have the desired result. Rather than write out a formal proof in full, it is usual to annotate a program text with assertions. These are formulae that express the values of, and relations between, program variables at points in the program's execution. Assertions are enclosed in curly brackets, and the annotated program is called a 'proof outline'. A formal proof could be constructed, if necessary, from the proof outline and the axioms and rules of the proof system. For our first example, we would thus write

$$\{x = 0\}$$
$$x := x+1;$$
$$\{x = 1\}$$

which states that, if $\{x = 0\}$ is true before execution of the assignment statement, then upon termination of the assignment, $\{x = 1\}$ is true.

Now consider two tasks that rendezvous and then terminate:

```
task body T1 is
begin
  {true}
  T2.E(1);
end; {true}
```

```
task body T2 is
x:integer;
begin
  {true}
  accept E (y: in integer) do
    x := y;
  end;
end: {x = 1}
```

We may consider each task in isolation, and make assertions as shown ('true' is effectively saying nothing). The final assertion of $T2$ is an inspired guess: whether x

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

† Present address: Data Systems Division, Marconi Radar Systems Ltd, Writtle Road, Chelmsford CM1 3BN, England.

really does have the value one or not at that point depends on what calls are made to E. The cooperation test checks that possible calls do not contradict the guess. In this example we may show cooperation by simply using the 'communication' proof rule. Thus, if

$$\{true \wedge y = 1\}$$
$$x := y;$$
$$\{x = 1\}$$

is valid reasoning (which it is), then we may conclude (where T_1 *par* T_2 represents parallel execution of T_1 and T_2) that

$$\{true\}\ T2.E(1)\ \textbf{par accept}\ E \ldots \{x = 1\}.$$

As the tasks cooperate, then upon termination x indeed has the value one. Note that there is no need for a global invariant in this example.

It is not always possible to prove a result using only the original program variables; it may be necessary to introduce 'auxiliary' or 'history' variables. Consider the two tasks:

task body $T1$ **is**	**task body** $T2$ **is**
x:*integer*: = 0;	y:*integer*: = 0;
begin	**begin**
$x := x+1$;	$y := y+1$;
end;	**end**;

We are given that $\{x = y\}$ is true initially, and wish to prove that $\{x = y\}$ is true when the tasks terminate. Assertions that appear in the text of $T1$ may not refer to variables from $T2$, and vice versa; hence, we need to introduce auxiliary variables x',y' initialised to the initial values of x,y respectively:

task body $T1$ **is**	**task body** $T2$ **is**
x,x':*integer*: = 0;	y,y':*integer*: = 0;
begin $\{x = x'\}$	**begin** $\{y = y'\}$
$x := x+1$;	$y := y+1$;
end; $\{x = x'+1\}$	**end**; $\{y = y'+1\}$

We also require a global invariant (that may refer to any variable), namely $\{x' = y'\}$. GI is true initially, and must be true finally as x',y' do not alter. $T1$ and $T2$ trivially cooperate (as they do not communicate), so we may conclude that

$$\{x = x'+1 \wedge y = y'+1 \wedge x' = y'\}$$

is true upon termination of the tasks, i.e. $\{x = y\}$ is also true. In general, auxiliary variables and the global invariant not only relate variables from separate tasks, but also ensure that rendezvous that are possible syntactically, but not semantically, trivially satisfy the cooperations conditions (see section 4 for an example). Variables free in a global invariant may only be altered in so-called 'bracketed sections', i.e. areas of a task immediately surrounding an entry call or accept statement.

3. THE PROOF SYSTEM

3.1. Definitions and assumptions

We confine our attention to a subset of the Ada tasking constructs that consists of:

tasks, entries, entry calls, accept statements and selective wait statements with no *else* part and no delay alternatives.

We do not treat the 'real-time' constructs (delay statements, conditional and timed entry calls), priorities and the abort statement.

We assume the following.

(i) An Ada program consists of a fixed number of tasks, all at the same conceptual level. We ignore all questions of their declaration and activation within some containing unit, and assume that all programs are well formed. We also ignore the declaration of entities associated with a task, such as entries and variables, but assume that all such entities have distinct names.

(ii) A statement within a task body is either an entry call, or an accept, selective wait (restricted as indicated earlier), assignment, **while** loop, if or null statement.

(iii) Tasks do not share variables, and there are no hidden side-effects in expression evaluation or statement execution.

(iv) Entry parameters have no defaults, and their associations are always positional, never named.

(v) There are no identifier clashes (between task names, or names of entities associated with different tasks).

(vi) All constructs terminate normally (i.e. we only consider partial correctness). Normal termination includes execution of a terminate alternative.

(vii) All actual parameters for a given entry call are disjoint. Note that this means that **in out** parameters cannot be handled simply by concatenating them to both **in** and **out** parameter lists.

For the remainder of this paper, T denotes a task, E denotes an entry, S a statement, x a variable (an object that has a value), e an expression and b a Boolean expression (an expression that evaluates to *tt* or *ff*, where *tt* and *ff* denote the truth values). A set of tasks or program is written T_1 **par** ... **par** T_n or Pr (**par** indicates concurrent execution). Assertions, i.e. formulae of some first-order language whose variables and expressions include those of our Ada subset, have typical elements p, q, r; 'true' and 'false' are assertions whose meaning is always *tt* and *ff* respectively. For any assertion q, FV(q) denotes the set of free variables, i.e. those not bound by any quantifier, appearing in q, for example $\mathrm{FV}(\exists y\,.\,x = y) = \{x\}$. The use of FV is also extended to apply to statements as well as assertions, thus $\mathrm{FV}(x := y+1) = \{x, y\}$.

We shall assume that all entry parameters of a given mode may be collected into a single list, which preserves their order and is denoted by *ain*, *aio*, *aout* for actual parameters, *fin*, *fio*, *fout* for formal, and we shall write (*fin*, *fio*, *fout*) for

(*fin*: **in** ...; *fio*: **in out** ...; *fout*: **out** ...).

With this notation then, for example

accept *SOME_ENTRY* (x: **in** *typex*; y: **out** *typey*; z: **in** *typez*) **do**
  $y := x+z$;
end;

may be considered as

accept E (*fin*, *fout*) **do**
  fout (1): = *fin* (1)+*fin* (2);
end;

We also define (cf. Ref. 11) the following notions.

(a) Proof outline or local proof

Considering a single task, T, in isolation, we may derive a formal proof of $\{p\}T\{q\}$, for suitable p, q, using axioms A1–A4, R1–R5 of section 3.2. Formal proofs are

notoriously tedious to produce and to understand, and, following general practice, we shall simply annotate the text of T with appropriate assertions from which, if necessary, a formal proof could be constructed. For a statement, S, of T we supply a *pre-assertion*, pre(S), and a *post-assertion*, post(S); note that these assertions refer solely to the variables of T. The body of an accept statement is not annotated. Intuitively, pre(S) (respectively, post(S)) holds just before (respectively, after) execution of S. We call the annotated text a proof outline or local proof.

(b) Global invariant, GI

An assertion that may include variables drawn from a number of tasks.

(c) Input/output, IO, commands

An IO command is an entry call or an accept statement. Two IO commands *match* if one is a call to an entry for which the other is an accept statement.

(d) Transformation

IO commands may be nested (inside the body of an accept statement). The proof system does not handle nested IO commands directly; should any occur, it is necessary to transform the relevant accept statements, and all matching entry calls, as indicated below. T' is the transformed version of T, E', E'' are some new entry names, *fin'*, *fio'*, *fio''*, *fout'* are some new variable names with appropriate scope, and S' is S with (elements of) *fin'*, *fio'*, *fout'* substituted for (the corresponding elements of) *fin*, *fio*, *fout*.

$T.E$ (*ain*, *aio*, *aout*); **par accept** E (*fin*, *fio*, *fout*) **do** S; **end**; is transformed to

$T'.E'$ (*ain*, *aio*); **par** **accept** E' (*fin*, *fio*) **do**
$T'.E''$ (*aio*, *aout*); *fin'*: = *fin*; *fio'*: = *fio*;
end;
S';
accept E''(*fio''*, *fout*) **do**
fio'': = *fio'*; *fout*: = *fout'*;
end;

(e) Bracketed sections, BS

These are sections of the program, delimited by '<' and '>', of the form

$$S_1; \text{IO}; S_2;$$

where S_1 and S_2 do not include any IO command. Two bracketed sections match if they contain matching IO commands. every IO command in a program must appear in a bracketed section. Assertions pre(BS), post(BS) are identical to pre(S_1), post(S_n), where BS is $S_1; \ldots S_n$. Note that bracketed sections may not be nested: any nesting of IO commands should be removed by program transformation (d). A program is said to be bracketed if all of its IO commands are bracketed.

(f) Auxiliary variables

These are variables introduced into a program solely to express assertions that cannot be stated in terms of the given program variables. We denote the set of such variables, for a given program, by AV.

(g) Cooperation

Given a bracketed program Pr $\triangleq T_1$ **par**...**par** T_n, with no nested IO commands, and given a global invariant, GI, and local proofs $\{p_i\}T_i\{q_i\}$ for all tasks T_i, $1 \leqslant i \leqslant n$, then the local proofs *cooperate* if:

(i) No assertion used in the local proof of T_i has a free variable subject to change in T_j $(i \neq j)$;

(ii) $\{\text{pre}(\text{BS}_1) \wedge (\text{BS}_2) \wedge \text{GI}\}$ BS_1 **par** BS_2 $\text{post}(\text{BS}_1) \wedge \text{post}(\text{BS}_2) \wedge \text{GI}\}$ holds for all matching pairs of bracketed sections BS_1, BS_2;

(iii) no variable free in GI alters outside a bracketed section.

3.2. Axioms and proof rules

In the list of axioms and rules given below, A1, A2 and R1–R4 are the standard axioms and rules associated with sequential programming languages. The other rules are necessary to handle the extra tasking constructs and, where necessary, some explanation is provided. (For convenience, we omit the final ';' from Ada statements.)

A1. Null $\{p\}$ **null** $\{p\}$

A2. Assignment $\{p[e/x]\}\ x: = e\ \{p\}$
where $p[e/x]$ is p with every free occurrence of x replaced by e.

A3. Entry $\{p\}\ T.E(ain, aio, aout)\ \{q\}$
provided $\text{FV}(p) \cap \{aout\} = \emptyset$.

A4. Accept $\{p\}$ **accept** E (*fin*, *fio fout*) **do** S; **end** $\{q\}$
provided FV $(p, q) \cap \{fin, fio, fout\} = \emptyset$.

Notice that the axioms A3 and A4 allow any assertion to be written after an entry call or an accept statement. Intuitively, the local proof outline of a task is asserting what happens when the task is run in isolation and, hence, any entry call or accept statement would block. Now, because the local proof outline is only a partial correctness proof anything may be concluded. However, a successful proof of cooperation of local proofs of tasks (3.1.g) will ensure that assertions concluded after such statements will indeed be true when the task are run together. The restriction on the appearance of free variables in the assertions used in A3 and A4 – i.e., the **out** parameters of a call may not appear in its pre-assertion, and none of the formal parameters of an accept statement may appear in either its pre- or post-assertion – is discussed after presentation of the communication rule R6.

R1. Consequence $\dfrac{p \Rightarrow p_1, \{p_1\}S\{q_1\}.\ q_1 \Rightarrow q}{\{p\}S\{q\}}$
and similarly for T and Pr.

R2. Composition $\dfrac{\{p\}S_1\{q\}, \{q\}S_2\{r\}}{\{p\}S_1; S_2\{r\}}$

R3. Alternation $\dfrac{\{p \wedge b\}S_1\{q\}, \{p \wedge \neg b\}S_2\{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1;\ \textbf{else}\ S_2;\ \textbf{end if}\ \{q\}}$
treating **elsif**...as **else if**...
with appropriate parsing.

R4. Repetition $\dfrac{\{p \wedge b\}S\{p\}}{\{p\}\ \textbf{while}\ b\ \textbf{loop}\ S;\ \textbf{end loop}\ \{p \wedge \neg b\}}$

R5. Selective wait

$$\frac{\{p \wedge b_i\}S_i\{q\}, 1 \leqslant i \leqslant n, p \wedge b_{n+1} \Rightarrow \text{post}(T)}{\{p\}\ \textbf{select when}\ b_1 \Rightarrow S_1;\ \textbf{or}\ldots\textbf{when}\ b_n \Rightarrow S_n;\ \textbf{or when}\ b_{n+1} \Rightarrow \textbf{terminate};\ \textbf{end select}\ \{q\}}$$

where post(T) is the post-assertion of the task, T, containing the selective wait.

For a selective wait statement with no terminate alternative, the above rule (R5) without the premise $p \wedge b_{n+1} \Rightarrow \text{post}(T)$ should be used. Notice that R5

must be used to justify the premises of any other rule that is applied to a statement containing a selective wait. (The introduction of axioms like $\{true\}S;x:=0;\{x=0\}$, where S contains a selective wait is forbidden, cf O'Donnell's counter-examples to the Clint–Hoare Goto rule in ref. 45.)

R6. Communication

$$\frac{\{p[fio/aio] \wedge fin = ain\}\, S\, \{q\}}{\{p\}\, T.E(ain, aio, aout)\ \textbf{par accept}\ E(fin, fio, fout)\ \textbf{do}\ S;\ \textbf{end}\ \{q[aio/fio, aout/fout]\}}$$

provided $FV(p) \cap \{fin, fio, fout, aout\} = \emptyset$, $FV(q) \cap \{fin, aio, aout\} = \emptyset$

$p[fio/aio]$ is p with every occurrence of an element of aio replaced by the corresponding element of fio.

This rule would appear more symmetrical if we had $p[fio/aio, fin/ain]$ in the premise, but this would necessitate having $q[ain/fin, \ldots]$ in the conclusion (in order to preserve *ain*), which clashes with our intuitive idea of parameter passing. We cannot put $p \wedge fio = aio \wedge fin = ain$ in the premise because we could then 'prove', for example:

$\{x = 1\}$
$T.E(x);$ **par accept** $E(y)$ **do** $y := 2$; **end**;
$\{x = 1 \wedge x = 2\}$.

This is also the reason why *aout* cannot appear free in p and for the restrictions imposed in A3 and A4 earlier.

R7. Formation

$$\frac{\{p\}\, S_1; S_3\, \{p_1\}, \{p_1\}\ IO_1\ \textbf{par}\ IO_2\, \{q_1\}, \{q_1\}\, S_2; S_4\, \{q\}}{\{p\} < BS_1 > \textbf{par} < BS_2 > \{q\}}$$

where BS_1 is $S_1; IO_1; S_2$
and BS_2 is $S_3; IO_2; S_4$.

Informally, the execution of a matching pair of bracketed sections, say BS_1 and BS_2, will proceed by executing S_1 and S_3 in parallel, then by performing the rendezvous IO_1**par**IO_2, and finish with the parallel execution of S_2 and S_4. Since S_1 and S_3 (S_2 and S_4, respectively) do not contain any IO commands (3.1.e) and do not share any variables (3.1.iii), the effect of their parallel execution will be the same as their sequential execution (in any order) and so the usual sequential proof rules can be used. R7 uses the arbitrarily chosen execution order of $S_1; S_3$ ($S_2; S_4$).

R8. Parallel composition

$$\frac{\forall i, 1 \leqslant i \leqslant n\text{: local proofs } \{p_i\}\ T_i\ \{q_i\} \text{ cooperate}}{\{p_1 \wedge \ldots \wedge p_n \wedge GI\}\, T_1\ \textbf{par} \ldots \textbf{par}\ T_n\, \{q_1 \wedge \ldots \wedge q_n \wedge GI\}}$$

provided no T_i contains nested IO commands.

R9. Auxiliary variables

$$\frac{\{p\}\, Pr'\, \{q\}}{\{p\}\, Pr\ \{q\}}$$

where $x \in AV \Rightarrow x \notin FV(q)$, and x only appears in Pr′ either in assignments which have x (or $y \in AV$) as the target variable, or as an actual or formal *in* parameter; Pr is obtained from Pr′ by deleting all such assignments and parameters.

R10. Substitution

$$\frac{\{p\}\ Pr\ (q)}{\{p[e/x]\}\ Pr\ \{q\}}$$

provided $x \notin FV(q, Pr)$.

R11. Transformation

$$\frac{\{p\}\, T_1'\ \textbf{par} \ldots \textbf{par}\ T_n'\, \{q\}}{\{p\}\ T_1\ \textbf{par} \ldots \textbf{par}\ T_n\, \{q\}}$$

where T_i' is T_i transformed (as in 3.1(d)) to remove all nested IO commands, and q contains no reference to any formal parameter.

4. EXAMPLE OF USE

The axioms and proof rules may be divided into three groups:

A1–A4, R1–R5	are used to construct local proofs, and the premises of R6 and R7;
R6–R8	are used to combine the local proofs and derive a conclusion about the program as a whole;
R9–R11	relate, back to their original versions, programs that have been altered (by the addition of auxiliary variables, or the removal of nested IO commands) to their original versions.

Thus, given an Ada (subset) program, the general order of procedure is as follows.

(1) Transform the program to remove any nesting of IO commands.

(2) Add auxiliary variables, if necessary, and bracket the resulting program.

(3) Construct local proofs and the global invariant GI.

(4) Establish cooperation conditions (use R6 and R7).

(5) Draw an overall conclusion about the transformed program that includes auxiliary variables (use R8).

(6) Remove auxiliary variables and draw a final conclusion for the transformed program (use R9 and R10).

(7) Use R11 to establish that this result holds for the original program.

We exemplify use of the proof system with a program to partition the union of two disjoint non-empty sets of integers, S'' and L'', into two subsets S' and L' such that $|S'| = |S''|$, $|L'| = |L''|$, $S' \cup L' = S'' \cup L''$ and every element of S' is smaller than any element of L'. This problem, with a CSP solution, is given in Ref. 5 and attributed to Dijkstra. Assuming the availability of obvious set operations max, min, U (union), −, the program consists of the following two tasks.

```
task body SMALL is
  S:set_of_integer := S'';
  maxS, minL:integer;
begin
  maxS := max(S);
  LARGE.GETS(maxS);
  S := S−maxS;
  LARGE.GETL(minL);
  S := S U minL;
  maxS := max(S);
  while maxS > minL loop
    LARGE.GETS (maxS);
    S := S−maxS;
    LARGE.GETL(minL);
    S := S U minL;
    maxS := max(S);
  end loop;
end;
```

```
task body LARGE is
  L:set_of_integer: = L″;
begin
  accept GETS (bigS: in integer) do
    L : = L U bigS;
  end;
  accept GETL (littleL: out integer) do
    littleL : = min(L);
    L : = L − littleL;
  end;
  loop
    select
      accept GETS(bigS: in integer) do
        L : = L U bigS;
      end;
      accept GETL (littleL: out integer)
      do littleL : = min(L);
        L : = L − littleL;
      end;
      or terminate;
    end select;
  end loop;
end;
```

We claim that, if the program terminates, then the final values of S, L satisfy the requirements for S', L'. Informally, task *SMALL* passes *maxS*, the largest element of S, to task *LARGE* in return for *minL*, the smallest element of L, until $maxS < minL$. For convenience, we have set operations performed in the bodies of the *GETS*, *GETL* accept statements; in practice the tasks would probably be coded so that these operations were executed outside a rendezvous.

To prove the claim, using our system, we shall need to introduce auxiliary integer variables hS, hL, to indicate precisely when *minL* is the smallest element of L, and to ensure that the cooperation conditions are satisfied for all matching bracketed sections, including those that match syntactically but never interact in practice (e.g. the first **accept** *GETS* in *LARGE* and the call to *LARGE.GETS* in the loop of *SMALL*). Hence, we arrive at the following annotated program, assuming $|S''| = n$, $|L''| = m$ $(n, m > 0)$:

```
  task body SMALL1 is
    S:set_of_integer: = S″;
    hS:integer: = 0; maxS, minL:integer;
  begin
      {|S| = n ∧ hS = 0 ∧ S = S″ ∧ max(S)∈S}
    maxS : = max(S);
      {|S| = n ∧ hS = 0 ∧ maxS∈S}
<   LARGE1.GETS(maxS);      BS1
      {|S| = n ∧ maxS∈S}
    S : = S − maxS;
      {|S| = n−1}
    hS : = 1;   >
      {|S| = n−1 ∧ hS = 1}
<   LARGE1.GETL(minL);      BS2
      {|S| = n−1 ∧ minL∉S}
    S : = S U minL;
      {|S| = n}
    hS : = 2;   >
      {|S| = n ∧ hS = 2 ∧ max(S)∈S}
    maxS : = max(S);
      {p}, where p ≡ |S| = n ∧ hS = 2 ∧ maxS∈S ∧
      maxS = max(S)
  while maxS > minL loop
      {p ∧ maxS > minL}
<   LARGE1.GETS(maxS);      BS3
      {|S| = n ∧ maxS∈S}
    S : = S − maxS;
      {|S| = n−1}
    hS : = 3;   >
      {|S| = n−1 ∧ hS = 3}
<   LARGE1.GETS(minL);      BS4
      {|S| = n−1 ∧ minL∉S}
    S : = S U minL;
      {|S| = n}
    hS : = 2;   >
      {|S| = n ∧ hS = 2 ∧ max(S)∈S}
    maxS : = max(S);
      {p}
  end loop;
  {p ∧ maxS ⩽ minL}
end; {p ∧ maxS ⩽ minL}

task body LARGE1 is
  L:set_of_integer: = L″; hL:integer: = 0;
begin
    {|L| = m ∧ hL = 0 ∧ L = L″)
< accept GETS(bigS: in integer) do      BL1
      L : = L U bigS;
end;
      {|L| = m+1}
  hL : = 1;   >
      {|L| = m+1 ∧ hL = 1}
< accept GETL (littleL: out integer) do   BL2
      littleL : = min(L);
      L : = L − littleL;
end;
      {|L| = m}
  hL . = 2;   >
    {q}, where q ≡ |L| = m ∧ hL = 2
loop
  {q}
  select
    {q}
<   accept GETS(bigS: in integer) do   BL3
      L : = L U bigS;
    end;
      {|L| = m+1}
    hL : = 3;   >
      {|L| = m+1 ∧ hL = 3}
<   accept GETL (little;L: out integer) do   BL4
      littleL : = min(L);
      L : = L − littleL;
    end;
      {|L| = m}
    hL : = 2;   >
      {q}
  or {q} terminate;
  end select;
    {q}
end loop;
  {q}
end; {q}
```

The global invariant, *GI*, is

$$S \cap L = \emptyset \wedge S \cup L = S'' \cup L'' \wedge hS = hL \wedge hS = 2 \Rightarrow minL < min(L).$$

The local proofs are constructed by a straightforward application of A1–A4, R1–R5; each local proof refers only to variables from the appropriate task, and no

variable free in GI alters outside a bracketed section. Hence the local proofs cooperate, provided that

$$\{\text{pre}(BS_1) \wedge (BS_2) \wedge GI\}\ BS_1\ \textbf{par}\ BS_2\ \{\text{post}(BS_1) \wedge \text{post}(BS_2) \wedge GI\} \quad (4.1)$$

holds for all matching bracketed sections, i.e. $BS1$–$BL1$, $BS1$–$BL3$, $BS2$–$BL2$, $BS2$–$BL4$, $BS3$–$BL3$, $BS3$–$BL1$, $BS4$–$BL4$, $BS4$–$BL2$. (4.1) holds trivially for $BS1$–$BL3$, since

$$(hS = 0 \wedge hL = 2 \wedge hS = hL) \equiv \text{false},$$

and similarly for $BS2$–$BL4$, $BS3$–$BL1$, $BS4$–$BL2$.

Now consider $BS1$–$BL1$. Clearly,

$$\{|S| = n \wedge hS = 0 \wedge maxS \in S \wedge |L| = m \wedge hL = 0 \wedge L = L'' \wedge GI \wedge bigS = maxS\}$$
$$L := L\,U\,bigS;\ \{p1\}$$

where

$$p1 \equiv |S| = n \wedge hS = 0 \wedge maxS \in S \wedge |L| = m+1 \wedge hL = 0 \wedge maxS \in L \wedge (S - maxS) \cap L = \emptyset \wedge S \cup L = S'' \cup L'' \wedge hS = hL \wedge hS = 2 \Rightarrow minL < min(L).$$

Hence, by R6,

$$\{\text{pre}(BS1) \wedge \text{pre}(BL1) \wedge GI\}\ LARGE.GETS(\ldots)\ \textbf{par accept}\ GETS\ldots\{p1\}$$

Clearly, $\{p1\}$ $S := S - maxS$; $hS := 1$; $hL := 1$; $\{\text{post}(BS1) \wedge \text{post}(BL1) \wedge GI\}$. Hence, by $R7$, (4.1) holds for $BS1$–$BL1$, and similarly for $BS3$–$BL3$.

Now consider $BS2$–$BL2$. Clearly,

$$\{|S| = n-1 \wedge hS = 1 \wedge |L| = m+1 \wedge hL = 1 \wedge S \cap L = \emptyset \wedge S \cup L = S'' \cup L'' \wedge min(L) \in L\}$$
$$littleL := min(L);$$
$$L := L - littleL;\ \{q1\}$$

where

$$q1 \equiv |S| = n-1 \wedge hS = 1 \wedge |L| = m \wedge hL = 1 \wedge S \cap L = \emptyset \wedge S \cup L \cup \{littleL\} = S'' \cup L'' \wedge littleL < min(L) \wedge littleL \notin S.$$

Hence, by $R1$ and $R6$,

$$\{\text{pre}(BS2) \wedge \text{pre}(BL2) \wedge GI\}$$
$$LARGE.GETL(\ldots)\ \textbf{par accept}\ GETL\ldots$$
$$\{q1\,[minL/littleL]\}$$

Clearly, $\{q1\,[minL/littleL]\}$ $S := SUminL$; $hS := 2$; $hL := 2$; $(\text{post}(BS2) \wedge \text{post}(BL2) \wedge GI)$. Hence, by $R7$, (4.1) holds for $BS2$–$BL2$, and similarly for $BS4$–$BL4$.

We have now shown that the local proofs cooperate. Let

$$p2 \equiv |S| = n \wedge hS = 0 \wedge S = S'' \wedge maxS \in S \wedge |L| = m \wedge hL = 0 \wedge L = L'' \wedge GI,$$

$$q2 \equiv |S| = |S''| \wedge |L| = |L''| \wedge S \cup L = S'' \cup L'' \wedge S \cap L = \emptyset \wedge max(S) < min(L).$$

By $R8$, $\{p2\}$ $SMALL1$ **par** $LARGE1$ $\{p \wedge maxS \leqslant minL \wedge q \wedge GI\}$,
and this post-condition implies $q2$, as required.

By $R9$, we may remove the assignments to hS and hL in $SMALL1$ and $LARGE1$.

By $R10$, substituting $hS = 0$, $hL = 0$ in $p2$ yields

$$\{|S| = n \wedge S = S'' \wedge |L| = m \wedge L = L'' \wedge S \cap L = \emptyset \wedge S \cup L = S'' \cup L''\}$$
$$SMALL\ \textbf{par}\ LARGE$$
$$\{q2\}.$$

Hence, if the program terminates, the final values of S, L do satisfy the requirements for S', L'.

5. DEADLOCK DETECTION

The deadlock detection rules and definitions of our paper[11] (which are based on those of Ref. 5) need no revision; we repeat them, with slight changes of terminology, as follows.

An Ada (subset) task is *blocked* if it is waiting to execute

(i) an entry call or an accept statement, or
(ii) a selective wait statement.

The corresponding set of *communication capabilities* of a blocked task is then as below.

(i) the bracketed section, BS, surrounding the entry call or accept statement,

(ii) a subset of {any terminate alternative} ∪ {the bracketed sections surrounding the accept statement(s) of the selective wait, S}. The subset consists of those members whose guard, b_i, $1 \leqslant i \leqslant m$, is true, where m is the number of alternatives in S. Let A ($\subseteq \{1,\ldots,m\}$) index these members. As we are only concerned with programs that terminate normally, $A \neq \emptyset$.

A terminated task, T, also has a communication capability, namely

(iii) acknowledgement of termination.

We associate assertions with these possible sets of communication capabilities as follows:

(i) pre(BS),
(ii) $\text{pre}(S) \wedge (\wedge_{j \in A} b_j) \wedge (\wedge\, \neg_{(j \notin A)} \neg b_j)$ $(1 \leqslant j \leqslant m)$,
(iii) post(T).

If all tasks T_i, $1 \leqslant i \leqslant n$, of a program are blocked or terminated, then we may define a corresponding n-tuple of sets of communication capabilities, one set per task, and a corresponding n-tuple of assertions. We call the conjunction of elements of the n-tuple of assertions a *potential deadlock formula* if, considering the n-tuple of sets of communication capabilities, the following clauses apply:

(a) the sets of communication capabilities do not contain a matching pair of bracketed sections,

(b) not all sets of communication capabilities include either a terminate alternative or an acknowledgement of termination.

Theorem 5.1

Given a proof $\{p\}Pr\{q\}$ with global invariant GI, Pr is deadlock-free (relative to p) if, for every potential deadlock formula PDF, $\neg(PDF \wedge GI)$ holds.

Proof. See Mearns.[42]

Before illustrating the use of this theorem, we define, for any given program, the assertion $ALLEND$ to be that

assertion which is true iff all tasks in the program have terminated or are capable of executing a terminate alternative. (*ALLEND* is formally defined in Ref. 42.) *ALLEND* may only appear in a proof outline as an implicit guard on a terminate alternative, or as part of a task's post-assertion. Since *ALLEND* is clearly global, this definition appears to violate the rule that assertions in a task's proof outline may only refer to that task's local variables. However, this rule is only formulated because, in general, one task cannot make assumptions about another task's variables – but, in the case of a terminate alternative, the semantics of Ada guarantee that a terminate alternative will only be executed if *ALLEND* (and any explicit guard) is true.

As an example of the use of theorem 5.1, and the need for *ALLEND*, consider the set partition of section 4. For a potential deadlock formula, *PDF*, that corresponds to *SMALL*1 terminated or waiting to execute a call to *LARGE*1, and *LARGE*1 waiting at an accept or selective wait statement, $\neg(PDF \wedge GI)$ clearly holds, e.g.

$$\text{post}(SMALL1) \wedge \text{pre}(BL4) \wedge GI$$
$$=> hS = 2 \wedge hL = 3 \wedge hS = hL$$
$$=> \text{false},$$

and similarly for *LARGE*1 terminated and *SMALL*1 waiting to execute *BS*1, *BS*2 or *BS*4. However, $\text{pre}(BS3) \wedge \text{post}(LARGE1) \wedge GI$ does not immediately lead to a contradiction, although, clearly, such a situation cannot occur in practice. In order to handle this case, it is necessary to strengthen pre(**terminate**) and post(*LARGE*1) top $\{q \wedge ALLEND\}$; this does not affect the partial correctness proof given in section 4. Since pre(*BS*3) clearly implies $\neg ALLEND$, we have

$$\text{pre}(BS3) \wedge \text{post}(LARGE1) => \text{false},$$

and we may conclude that the set partition program is deadlock-free.

We briefly discuss two other aspects of the proof system.

(i) Failure. The above treatment of deadlock deals with program failure due to a *TASKING_ERROR*. We may handle program failure caused by *PROGRAM_ERROR* (when due to all alternatives in a selective wait being closed) by adding

$$p => (b_1 \vee \ldots \vee b_{n+1})$$

to the premise of *R*5.

(ii) Non-terminating programs. Some concurrent programs, e.g. (components of) operating systems, are not designed to terminate. We have stressed that our system is only for reasoning about partial correctness. However, we claim that the system may also be used to verify certain properties of deadlock-free programs that contain tasks of the form:

```
begin
  loop
    :
    S;
    :
  end loop;
end;
```

If all tasks cooperate, then we may conclude that, if and when statement *S* is executed, then post(*S*) is true. (See lemma 5.3.10 in Ref. 42.) However, the proof system cannot deduce whether *S* will, or will not, eventually execute.

6. SOUNDNESS AND RELATIVE COMPLETENESS

A proof system for a programming language should be sound, i.e. every formula that may be derived using the system is true in some sense, and relatively complete, i.e. every formula that is true in some sense may be derived using the system. (These definitions are somewhat naïve, but will suffice for the purposes of this section; further discussion and references may be found in Apt.[1] Note that in order to prove that a proof system possesses these qualities, it is necessary to have a formal definition or semantics of the programming language in question.

The proof that our proof system is sound and relatively complete, together with the semantics used, is given in the thesis of Mearns.[42] The proof is based on the formal justification for the *AFdR CSP* proof system as given by Apt,[2] which in turn follows the general line of reasoning behind the corresponding results in the thesis of Owicki.[46] The main difference is in the style of semantics used: both Apt and Owicki give an operational semantics in terms of a state transition relation, whereas Mearns' thesis employs a denotational semantics. (In the denotational semantics approach, language constructs are considered to denote familiar mathematical objects; an operational semantics views the meaning of a language construct in terms of the changes that it would induce in some hypothetical machine.)

There is, as yet, no firm consensus of opinion as to how best formally to define concurrent language constructs. The 'official' formal definition of Ada[29] takes a denotational approach, but does not give the full semantics of the tasking constructs. In Mearns' thesis,[42] the semantics is based on the theory of processes, or objects constructed from sets of sequences, as developed by de Bakker and Zucker.[16]

7. RELATED WORK

In this section, we compare our proof system with that of Gerth and de Roever[22] (hereafter referred to by GdR), which is the only other Ada tasking proof system known to us. Like our system, GDR is based on the AFdR CSP proof method.[5] The most significant difference is that an accept statement of the form

$$< \textbf{accept } E(\ldots) \textbf{ do } S_1; > S; < S_2; \textbf{ end}; >$$

is bracketed as shown, where S_1 and S_2 contain no IO commands, and that a 'canonical' proof outline for accept statement bodies is required in local proofs. Our communication and formation rules (*R*6 and *R*7) are replaced by a single 'rendezvous' rule, which permits the cooperation conditions for matching entry calls/accept statements to be verified. The premises of the rendezvous rule seek to establish that, after the start of a rendezvous involving an accept statement of the above form, pre(*S*)[*] and *GI* and pre('entrycall') hold, where [*] indicates substitution of actual for formal parameters; similarly, before the end of the rendezvous, post(*S*)[*] and *GI* and pre('entrycall') are assumed. With certain

restrictions on the parameters, pre(S) and post(S) may be taken from the local proof, and S not considered further in the rendezvous rule premises.

The obvious advantage of their approach is that nested IO commands are handled directly: an accept statement can always be bracketed as above, because S_1 and S_2, in general, need only contain an assignment to an auxiliary variable; hence, nested bracketed sections never occur in their system. The provision of a canonical proof outline for an accept body also saves some work if two or more calls match that accept, but the amount of verification saved is probably not as much as might appear at first sight, since the proofs of cooperation for a given accept in our system are likely, in practice, to be quite similar to each other. We also feel that our bracketing maintains the idea of a rendezvous as a single unity more than their bracketing does, although, of course, theirs emphasises that a rendezvous consists of two inter-task communications. (Hence, the Ada rendezvous may easily be modelled in CSP). When producing Ref. 11 we were convinced that nested IO commands would very rarely be used in practical examples, since the first calling task is certain to be delayed until all the inner rendezvous have been completed; however, some situations can be modelled rather elegantly with nested accept statements – consider the following example (due to J. R. Abrial):

```
task body marriage_agency is
  begin
    accept man (m: in ...; wife: out ...) do
      accept woman (w: in ...; husband: out ...) do
        if introduce (m, w) = fall_in_love
          then wife := w; husband := m;
          else wife := no_luck; husband: = no_good;
        end if;
      end;
    end;
  end;
```

Of course, this example could have been written with woman accepted before man, or even with both cases as alternatives of a selective wait!

GdR also formalises the notion of 'calling chains' that may arise with nested IO commands, and treats general safety properties, including freedom from deadlock. Termination and absence of failure is discussed. Soundness and relative completeness of the system are indicated by Gerth:[21] the method adopted is to translate the Ada subset considered into (an extension of) CSP, and then to show that a proof of an Ada program in the GdR system implies the existence of a proof of the translated program in the AFdR system, and vice versa. Since the AFdR system is sound and relatively complete, then so is the GdR system.

8. CONCLUSIONS

An axiomatic proof system for reasoning about the basic Ada tasking constructs has been presented. The system covers the partial correctness and freedom from deadlock of concurrent Ada programs, and has been proved sound and relatively complete against a denotational semantics.

The justification of our proof system has demonstrated that an earlier version, which was apparently correct, was actually unsound. We suggest that no proof system should ever be used in practice until it has been formally justified. Of course, in the absence of a full official formal definition of Ada, there is no guarantee that our semantics corresponds with the intentions of the authors of the Ada Language Reference Manual (LRM), or that it will agree with a particular compiler writer's interpretation; however, this is an argument for the more widespread use of formal definitions, rather than for the rejection of proof systems. Our semantics appears to agree closely with other formal interpretations of the LRM chapter on tasking, for example see Lovengreen,[36] Li.[35]

Subject to the above photograph, we hope that this work will be of practical use in the production of reliable Ada software. Our system has the usual advantages and disadvantages of formal proof systems, the use of which seems to generate some emotional heat, particularly amongst their detractors (see, for example, Merrill[43] for a discussion of this topic). However, for concurrent Ada programs, some of which will very likely have disastrous consequences should they fail to function as expected, we do not accept that having the option of using a proof system such as ours can be anything but beneficial. We fully endorse the approach of Gries[24] to the subject of formal program development: 'Use theory to provide insight; use common sense and intuition where it is suitable, but fall back on the formal theory for support when difficulties and complexities arise.'

Our earlier paper[11] included tentative proposals for dealing with the Ada 'real-time' constructs. We have not pursued these ideas any further in this paper, since our semantics do not extend to these constructs. To our knowledge little work has been done to date in this area, although recently some interesting results have been achieved by Shyamasundar *et al.*,[50] where a denotational semantics has been produced for a CSP-like language with real-time facilities. As yet, a proof system for such is not available.

We conclude by noting that the most tedious part of using the proof system is checking the cooperation conditions. Apt[4] indicates how to reduce the number of cooperation checks needed when verifying a CSP program, by first performing a 'static analysis' of the program. The basic idea is to consider all possible sequences of communications that can occur if Boolean guards are not interpreted. This identifies matching bracketed sections that can never, in fact, be synchronised, and which need not be considered in the proof. The analysis may also be used to determine possible configurations that are deadlocked. Taylor[51] independently presents a general-purpose algorithm for performing a similar static analysis of concurrent programs (exemplified by Ada). We expect that these results could readily be adapted to our proof system.

Acknowledgements

We gratefully acknowledge helpful comments from Jean-Raymond Abrial, Krzysztof Apt, Rob Gerth, Cliff Jones, Ruurd Kuiper and Joe Stoy. One of us (I.M.) would like to acknowledge the financial support of the U.K. Science and Engineering Research Council during the period of research leading to this paper.

APPENDIX A. PROGRAM PROOF SYSTEMS

In this appendix we briefly review program proof systems in general, thereby placing our proof system in a wider context. Specific comparison with another Ada tasking proof system (Gerth and de Roever)[22] was given in section 7. Note that we do not claim our system is a method for developing Ada tasking programs; neither do we review work concerned with the automation of program verification.

The seminal paper for research into verifying the behaviour of computer programs is Floyd,[20] which introduces *inductive* or *intermediate assertions* (predicates) that are associated with every stage in the flow of control through a program. An assertion characterises the relation between program variables at that point in the program's execution; assertions for adjacent stages are themselves related according to whether an assignment or a test takes place; programs are usually represented as directed graphs. See Manna[37] for a full description.

Probably the best-known approach to program verification is the axiomatic method of Hoare,[26] which gives a proof system for reasoning about triples $\{p\}S\{q\}$, where p, q are assertions (over program variables) and S is (the text of) a program. The intuitive meaning of $\{p\}S\{q\}$ is: if p holds before execution of S, and S terminates, then q holds. The proof system consists of axioms and rules of inference for program statements and combinations of statements. Hoare logic has been used for a large variety of programming languages: see Apt[1, 3] for an extensive survey of published results concerning sequential and non-deterministic constructs.

Recall from section 6 that proof systems should be *sound* (every provable formula is true with respect to some semantics) and *relatively complete* (every true formula, with respect to that semantics, is provable). Clarke[13] identifies certain language constructs (e.g. recursive procedures with procedure parameters and static scope of identifiers) for which it is impossible to obtain a sound and complete Hoare system. However, axioms and rules for the most common sequential language constructs are all proved sound and complete against a denotational semantics by de Bakker.[15]

A proof system may deal with *partial* correctness of a program (if assertion p holds initially, then assertion q holds **if** the program terminates) or *total* correctness (if p holds initially, then q holds finally **and** the program terminates). In the paper[20] of Floyd, termination of a program is proved by showing that each step decreases the value of some expression, whose values are members of a well-founded set (a partially ordered set that contains no infinite decreasing sequence). The same basic idea may be applied to Hoarse systems. In general, total correctness of a program may be shown either in two steps (by using a partial correctness system plus a method of proving termination) or in one step (by using a total correctness system whose rules are formulated to take possible non-termination into account). Following Burstall,[12] Manna and Waldinger[41] propose *intermittent assertions* which may be associated with points in a program text; such an assertion is true at some time (not necessarily always) when control reaches its corresponding point. So 'sometime q at end' means 'eventually control will reach the end of the program and q will hold'; if this assertion is true, and q is the desired post-condition, then the program is totally correct. Gries[23] disputes the claim in Ref. 41 that intermittent assertions are more 'natural' than invariant ones; however, the use of *temporal logic* (a formalism for abstract reasoning about time, and by which intermittent assertions can be formalised) in total correctness systems is growing more widespread, particularly where concurrency is involved.

For Hoare systems, Harel[25] give a single rule for the total correctness of **while** loops. He also shows that the resulting proof system is sound and complete in an *arithmetical* interpretation, i.e. one whose domain includes the natural numbers, the standard Peano functions and the ability to decide if a given symbol represents a natural number or not. (There is no non-trivial Hoare system for total correctness that is sound in all interpretations.[1] Harel's proof system is derived from *dynamic logic*, which is similar to temporal logic in that its augments the classical 'static' predicate calculus operators with additional primitives. These latter primitives enable expression of assertions such as $[a]p$ – 'property p holds at the end of all possible executions of the program a' or $< a > p$ – 'property p holds at the end of some possible execution of program a'. Dynamic logic may also be seen as an extension of *infinitary logic*, which is first-order logic that permits countable disjunctions and conjunctions of formulae. A similar idea is exploited in *algorithmic logic* (see Salwicki[49] for references).

Dijkstra[18] introduces, for a language construct S, a *predicate transformer*, which is a rule describing how to derive the weakest pre-condition that guarantees a given post-condition and termination of S. Other researchers have studied his rules, particularly those dealing with the total correctness of (bounded) non-deterministic constructs: see Harel,[25] for example, where dynamic logic is used to define a semantic model, and Back,[6] where the predicate transformers are expressed in an infinitary logic. Flon and Suzuki[19] present a total correctness proof system for parallel programs in terms of predicate transformers, after recognising that a parallel program has an equivalent non-deterministic form. They show the system is sound and relatively complete by demonstrating that their predicate transformers (i.e. weakest pre-conditions) are, in a semantic model, extremum fixed points of continuous functions over predicates; soundness and relative completeness then follow directly from two general metatheorems concerning predicate transformers and proof rules.

We now consider proof systems that do not involve the transformation of programs, and which are designed explicitly for parallel languages. A pioneering paper is Owicki and Gries.[47] Processes executing concurrently and sharing global variables are first considered in isolation, and Hoare logic used to deduce a post-condition for each process; provided the individual proofs are proved to be 'interference-free' (execution of one process does not invalidate the proof of another process), then the post-conditions may be combined to give the overall effect of the concurrent program. Freedom from deadlock is also provable in their system. A similar approach, using cooperation tests for combining proof outlines, is used in the AFdR CSP proof system,[5] introduced in section 2, and in the CSP proof system of Levin and Gries.[34] In CSP, processes do not share

variables, but Levin and Gries permit shared auxiliary variables (rather than using a global invariant as in the AFdR system); thus their approach requires a sequential proof for each process, a 'satisfaction' proof (which is similar to showing cooperation in AFdR) and a non-interference proof similar to that in the Owicki and Gries approach.

Other proof systems for CSP are those of Misra and Chandy[44] and Zhou and Hoare,[54] which both utilise the notion of a *trace*, or record of values communicated to or from a process. In Ref. 44 assertions are primarily over traces, which may be used as auxiliary variables, if necessary, there is no explicit concept of an auxiliary variable or a global invariant. (Cf. Clint,[14] who states that some form of auxiliary or history variable is indispensable for proofs of certain types of program; in particular, those that accept messages from an outer environment.) In Ref. 54 processes are semantically defined in terms of traces, and the proof rules shown to be sound against this model. Both Refs 44 and 54 permit the formal development of networks of processes, but do not deal (directly) with deadlock freedom. Misra and Chandy claim that freedom from deadlock may be shown by combining the proof system with earlier work by the same authors; Hoare[28] extends the work in Ref. 54 to a total correctness system in which the absence of deadlock may be proven, and Zhou[52] shows that this system is consistent with an operational model. Zhou, in Ref. 53, also develops a similar notion to predicate transformers for communicating processes – the *weakest environment*, we, such that a process P satisfies an assertion R iff we (P, R) is true.

Lamport[30, 31, 32] studies proof systems for shared variable parallel languages. In Ref. 30 he introduces the terms *safety* and *liveness* to refer to properties of multiprocess programs. A safety property states that something (bad) will not happen, and a liveness property states that something (good) will happen. Partial correctness and absence from deadlock are generally taken to be safety properties (that do not depend on whether execution is fair or not), whilst termination and eventual execution of a statement (absence of livelock) are liveness properties (that do depend on fairness). In Ref. 30 processes are represented as graphs with assertions attached to arcs. Safety properties are proved using a similar idea to Ref. 47; to prove liveness it is necessary to find suitable assertions such that, if a process's 'token' (indicating where local control resides at any time) does not eventually move from an arc, then a contradiction arises. Unfortunately, proofs of liveness properties are very hard to construct in this system. In Ref. 31 Lamport eschews assertions in favour of two relations – 'precedes' and 'can influence' – between non-atomic operations, and uses axioms for these relations to prove a mutual exclusion algorithm correct; in Ref. 32 he reverts to assertions, that may include 'location counters' (e.g. 'after S' if control resides after statement S), and gives rules for proving safety properties; and in Owicki and Lamport[48] the underlying ideas on liveness in Ref. 30 are combined with a formal treatment of location counters (using temporal logic). In Ref. 33 Lamport provides a good overview of his use of temporal logic in program specification and verification.

The best-known exponents of temporal logic (for program verification) are Manna and Pnueli, who interpret temporal formulae over infinite, linear and discrete sequences of states. In Ref. 38 they consider concurrent programs that communicate via shared variables. A semantic model of interleaved atomic actions is defined, and various safety and liveness properties of programs are expressed using the formalism of temporal logic and the concept of location counters. In Ref. 39 they present a proof system for proving these properties of parallel programs. It consists of three parts:

(i) 'pure' temporal logic axioms and rules,

(ii) domain axioms (that axiomatise the data types on which the program operates),

(iii) program axioms (that may be considered as giving the 'temporal semantics' of the programming language).

In Ref. 40 the program axioms are generalised to 'interface' with any concurrent programming language through the concepts of atomic transitions, justice and fairness. By defining these concepts for a given language, a relatively complete proof system is easily obtained. The method is exemplified with a shared variable language and CSP. Recently, compositional (and hence syntax-directed) proof systems have been developed in the temporal framework. The papers[9, 10] of Barringer *et al.* demonstrate, in a general style, how compositional temporal proof systems can be obtained for parallel languages based on shared variables and message-based communication mechanisms.

We do not comment on the relative merits, or otherwise, of these concurrent proof systems. An extensive survey and comparison of non-temporal approaches is given by Barringer.[8] As with the semantics of concurrency, there is as yet no consensus as to the best method of verifying parallel programs, although it is generally true that processes communicating via shared variables are more difficult to reason about than processes that utilise explicit message-passing commands.

APPENDIX B. CRITIQUE OF THE ORIGINAL SYSTEM

As mentioned in the introduction, our original proof system[11] was published before it was proved to be sound and (relatively) complete. The justification for this was that it was firmly based on the AFdR system, for which the proofs of soundness and completeness (by Apt[2]) did not appear until later; it was anticipated that these proofs could easily be adapted to show that our original system[11] was sound and relatively complete. Essentially, it is, but the paper does have a number of informal statements open to misinterpretation, omissions and errors that can lead to inconsistency. Three examples follow.

(i) (Due to Rob Gerth.) Given the following annotated task bodies:

```
task body T1 is
begin
  {true}
< T2.E; >
end; {true}
```

```
task body T2 is
h: integer: = 0;
begin
  {h = 0}
< accept E do
    h: = 1; {h = 1}
  < T3.E(1); >
    h: = 0; {h = 0}
  end; >
end; {h = 0}
```

```
task body T3 is
y: integer;
```

```
begin
  {true}
< accept E (x: in integer) do
    {x = 0} y: = x; {y = 0}
  end; >
end; (y = 0}
```

we may take $GI \triangleq h = 0$ and 'prove' that $\{true\}$ $T1\mathbf{par}T2\mathbf{par}T3$ $\{y = 0\}$, which is obviously incorrect. It was intended that such an example would require that $GI \triangleq h = 0 \vee h = 1$, but this intention is not stated formally.

(ii) The 'communication out rule' of [11] is

$$\frac{\{Q\}\, s\, \{R\}}{\{Q\}\quad T.E(x) \parallel \textbf{accept } E\ (y\text{: }\textbf{out} \ldots)\ \textbf{do } s;\ \textbf{end};\quad \{R \wedge x = yf\}}$$

where 'yf means the final value of a formal parameter y prior to reaching the end of the accept statement'. As well as being clumsy, the rule omits a necessary condition for soundness, namely $FV(R) \cap x = \emptyset$.

This omission may be exemplified by the two task bodies:

```
task body T1 is          task body T2 is
x:integer: = 0;          begin
begin                      {true}
  {x = 0}                < accept E (y: out integer) do
< T2.E(x); >                 y: = 1;
end;                       end; >
{x = 0 ∧ x = 1}          end; {true}
```

Using the communication out rule as given, we could (erroneously) conclude that $\{true\}$ $T1\mathbf{par}T2$ $\{x = 0 \wedge x = 1\}$.

(iii) The terminate statement axiom of Ref. 11 is

$$\{P\}\ \textbf{terminate}\ \{false\}$$

which is unsound: consider a program consisting of just one task, T, where

```
task body T is
begin
  {true}
  select accept E; {false}
    or terminate; {false}
  end select;
end; {false}
```

The conclusion $\{true\}$ T $\{false\}$ is false.

Clearly, none of the above objections applies to the revised proof system. Note that a first revision of Ref. 11 sought to retain a direct treatment of nested IO commands by extending the cooperation conditions, and surrounding nested bracketed sections with 'canonical' assertions which must be preserved during all rendezvous involving the outer bracketed section. This is intuitively reasonable, and is justified by showing that, if the program is transformed to remove the nesting, then the validity of all relevant correctness formulae is implied by the extra cooperation conditions. However, this earlier revision still uses the rule of parallel composition ($R8$ in section 3.2), and we have been unable to prove $R8$ sound in the presence of nested IO commands. Hence there is a danger that the system is, in O'Donnell's terminology,[45] 'theorem sound' (every provable formula is true) but not 'inferentially sound' (every step in a proof is correct), and it has been rejected in favour of the system presented here.

REFERENCES

(Note: LNCS abbreviates *Lecture Notes in Computer Science*, Springer-Verlag.)

1. K. R. Apt, Ten years of Hoare's logic: a survey – part I. *ACM TOPLAS*, **3** (4) 431–483 (1981).
2. K. R. Apt, Formal justification of a proof system for communicating sequential processes. *JACM* **30** (1) 197–216 (1983).
3. K. R. Apt, Ten years of Hoare's logic: a survey – part II: nondeterminism. In *Foundations of Computer Science. IV. Distributed Systems*, part 2, *Semantics and Logic*, edited J. W. de Bakker and J. van Leeuwen. Amsterdam Mathematical Centre, pp. 101–132 (1983).
4. K. R. Apt, A static analysis of CSP programs. Technical report, LITP, Paris University (1983).
5. K. R. Apt, N. Francez and W. P. de Roever, A proof system for communicating sequential processes. *ACM TOPLAS* **2** (3), 359–385 (1980).
6. R. J. R. Back, Proving total correctness of nondeterministic programs in infinitary logic. *Acta Informatica* **15**, 233–249 (1981).
7. J. G. P. Barnes, *Programming in Ada*, 2nd edn, Addison-Wesley, London (1984).
8. H. Barringer, A survey of verification techniques for parallel programs. *LNCS* **191** (1985).
9. H. Barringer, R. Kuiper and A. Pnueli, Now you may compose temporal logic specifications. *Proc. 16th ACM Symposium on Theory of Computing, Washington* (1984).
10. H. Barringer, R. Kuiper and A. Pnueli, A compositional approach to a CSP-like language. *Proc. IFIP Working Conference on the Role of Abstract Models in Information Processing, Vienna* (1985).
11. H. Barringer and I. Mearns, Axioms and proof rules for Ada tasks. *IEE Proceedings* **129** (E2), 38–48 (1982).
12. R. M. Burstall, Program proving as hand simulation with a little induction. *IFIP 74*. North-Holland, Amsterdam (1974).
13. E. M. Clarke Jr, Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *JACM* **26** (1), 129–147 (1979).
14. M. Clint, On the use of history variables. *Acta Informatica* **16** (1), 15–30 (1981).
15. J. W. de Bakker, *Mathematical Theory of Program Correctness*. Prentice-Hall International, Hemel Hempstead (1980).
16. J. W. de Bakker and J. I. Zucker, Processes and the denotational semantics of concurrency. *Information and Control* **54** (1), 70–120 (1982).
17. Department of Defense (U.S.), Reference manual for the Ada programming manual, ANSI/MIL-STD 1815A (1983). (Approved 17 Feb. 1983, American National Standard Institute, Inc.)
18. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, New York (1976).
19. L. Flon and N. Suzuki, The total correctness of parallel programs. *SIAM J Computing* **10** (2), 227–246 (1981).
20. R. W. Floyd, Assigning meanings to programs, In *Mathe-*

matical Aspects of Computer Science, edited J. T. Schwartz, *Proc. Symposia in Applied Mathematics* **19**, 19–32. American Mathematical Society (1967).
21. R. Gerth, A sound and complete Hoare axiomatisation of the Ada-rendezvous (extended abstract). In *Automata, Languages and Programming*, edited M. Nielsen and E. M. Schmidt. *Proc. Aarhus* 1982, *LNCS* **140**, 252–264 (1982).
22. R. Gerth and W. P. de Roever, A proof system for concurrent Ada programs. *Science of Computer Programming* **4** (2), 159–204 (1984).
23. D. Gries, Is sometime ever better than alway? *ACM TOPLAS* **1** (2), 258–265 (1979).
24. D. Gries, *The Science of Programming*. Springer-Verlag, New York p. 165 (1981).
25. D. Harel, *First-order dynamic logic LNCS* **68** (1979).
26. C. A. R. Hoare, An axiomatic basis for computer programming. *CACM* **12** (10), 576–583 (1969).
27. C. A. R. Hoare, Communicating sequential processes. *CACM* **21** (8), 666–677 (1978).
28. C. A. R. Hoare, A calculus of total correctness for communicating processes. *Science of Computer Programming* **1** (1), 49–72 (1981).
29. *INRIA, Formal Definition of the Ada Programming Language* (1982).
30. L. Lamport, Proving the correctness of multiprocess programs. *IEEE TOSE* **SE-3** (2), 125–143 (1977).
31. L. Lamport, A new approach to proving the correctness of multiprocess programs. *ACM TOPLAS* **1** (1), 84–97 (1979).
32. L. Lamport, The 'Hoare logic' of concurrent programs. *Acta Informatica* **14**, 21–37 (1980).
33. L. Lamport, What good is temporal logic? *IFIP 83*, pp. 657–668. North-Holland, Amsterdam (1983).
34. G. M. Levin and D. Gries, A proof technique for communicating sequential processes. *Acta Informatica* **15**, 281–302 (1981).
35. W. Li, An operational semantics of tasking and exception handling in Ada. Internal report CSR-99-82, Edinburgh University (1982).
36. H. H. Lovengreen, Parallelism in Ada. In *Towards a Formal Description of Ada*, edited D. Bjoerner and O. N. Oest. LNCS **98**, 309–432 (1980).
37. Z. Manna, *Mathematical Theory of Computation*, New York: McGraw-Hill (1974).
38. Z. Manna and A. Pnueli, Verification of concurrent programs: the temporal framework. In: *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore (ed), Academic Press, London, pp. 215–273 (1982).
39. Z. Manna and A. Pnueli, Verification of concurrent programs: temporal proof principles. In *Logics of Programs, Proc. Yorktown Heights 1981*, edited D. Kozen. LNCS **131**, 200–252 (1982).
40. Z. Manna and A. Pnueli, How to cook a temporal proof system for your pet language. *Proc. 10th ACM Symposium on principles of programming languages, Austin, Texas*, pp. 101–154 (1983).
41. Z. Manna and R. Waldinger, Is 'Sometime' sometimes better than 'Always'? Intermittent assertions in proving program correctness. *CACM* **21** (2), 159–172 (1978).
42. I. Mearns, A denotational semantics for concurrent Ada programs. Ph.D. thesis, University of Manchester (1983).
43. G. Merrill, Proofs, program correctness and software engineering. *SIGPLAN Notices* **18**, (12), 96–105 (1983).
44. J. Misra and K. M. Chandy, Proofs of networks of processes. *IEEE TOSE* **SE-7** (4), 417–426 (1981).
45. M. J. O'Donnell, A critique of the foundations of Hoare-style programming logics. In *Logics of Programs, Proc. Yorktown Heights 1981*, edited D. Kozen, LNCS **131**, 349–374 (1982).
46. S. S. Owicki, Axiomatic proof techniques for parallel programs. Technical report TR75-251, Cornell University (1975).
47. S. S. Owicki and D. Gries, An axiomatic proof technique for parallel programs. I. *Acta Informatica* **6**, 319–340 (1976).
48. S. S. Owicki and L. Lamport, Proving liveness properties of concurrent programs. *ACM TOPLAS* **4** (3), 455–495 (1982).
49. A. Salwicki, Algorithmic theories of data structures. In *Automata, Languages and Programming, Proc. Aarhus 1982*, edited M. Nielsen and E. M. Schmidt. LNCS **140**, 458–472 (1982).
50. R. K. Shyamasundar, W. P. de Roever, R. Gerth, R. Koymans and S. Arun-Kumar, Compositional Semantics for Real-Time Distributed Computing. Technical Report RUU-CS-84-6, Utrecht University (1984).
51. R. N. Taylor, A general-purpose algorithm for analyzing concurrent programs. *CACM* **26** (5), 362–376 (1983).
52. Zhou Chaochen, The consistency of the calculus of total correctness for communicating processes. Internal document, Oxford University PRG (1981).
53. Zhou Chaochen, Weakest environment of communicating processes. Internal document, Oxford University PRG (1981).
54. Zhou Chaochen and C. A. R. Hoare, Partial correctness of communicating processes and protocols. Technical monograph PRG-20, Oxford University PRG (1981).