Tentative Steps Toward a Development Method for Interfering Programs

C. B. JONES
Manchester University

Development methods for (sequential) programs that run in isolation have been studied elsewhere. Programs that run in parallel can interfere with each other, either via shared storage or by sending messages. Extensions to earlier development methods are proposed for the rigorous development of interfering programs. In particular, extensions to the specification method based on postconditions that are predicates of two states and the development methods of operation decomposition and data refinement are proposed.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Program Verification; D.3.2 [**Programming Languages**]: Language Classifications—*Ada*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Rely-conditions, guarantee-conditions, communicating sequential processes

1. INTRODUCTION

A brief review of the history of attempts to formalize the development of sequential (isolated) programs will set the context for the extensions we propose. The first results to appear were concerned with correctness proofs for complete programs and normally concentrated on trivial data structures such as natural numbers (cf. [7, 14, 31]). Subsequent papers showed how the proof rules could be used in a design process; in this way a proof could be used to justify the design step before development of the final code took place (cf. [5, 13, 39]). The wider application of such ideas became possible with the study of abstract data types and their refinement (cf. [12, 29]). The development method that evolved through [21], [20], and [18] mirrors this development but uses postconditions that are predicates of the initial and final states. This method is outlined in Section 2 below. The emphasis nowadays is more on a "rigorous method" that relies on the underlying mathematical ideas but in which these foundations are used mainly as a guide to less formal "correctness arguments." The approach of employing checklists of results (based on formal rules) as an integral part of the development

Author's address: Department of Computer Science, Manchester University, Manchester M13 9PL, England.

process can lead to higher productivity for the programming task because errors in design are detected before other work is based on them.

The development of microprocessors and distributed systems has given extra impetus to the study of programs that run other than in isolation. The term "tightly coupled" is applied to systems that interfere by sharing (at least partially) the same storage. Where processes communicate only via messages, systems are referred to as being "loosely coupled." Loosely coupled systems are somewhat more tractable for proof purposes (cf. [22, 28, 40]). Unfortunately, some situations force consideration of shared variables. Here, both loosely and tightly coupled systems are regarded as interfering (nonisolated) systems, although emphasis is on the latter.

As with isolated programs, the first formal material on interfering programs has been proof methods for complete programs: tightly coupled systems are addressed in the work of Owicki and Gries [33], and loosely coupled systems are covered in [3], [22], [25], and [40]. The approaches are characterized by proving correct the components of the completed programs in isolation and then proving that the proofs do not interfere. It is argued below that this is unacceptable as a program development method.

The nature of the problem of interference makes the application of ex post facto proof methods to the development process rather difficult. This paper shows how certain problems can be tackled by a development method that is a fairly natural extension of that described in [18]. The basic idea is to add to a specification a precise statement of its interference: a "rely-condition" defines assumptions that can be made in program development; a "guarantee-condition" places requirements on the interference a would-be implementation can generate. The proof rules that describe decomposition into parallel tasks define conditions for the interference specifications to match. These rules can be compared to those known for control structures like loops: once the proof at one stage of development is completed, the specification is a complete description of acceptable implementations. Another form of development shown below is the way in which data refinement proofs can give rise to tasks that are activated by communication activity. It would appear to be an important contribution to the development described in Section 3 that it uses predicates of pairs of states.

The presentation of parallel programs is given in the syntax of the Ada[1] language; this choice is based on the useful properties of the "rendezvous" concept [37]. The method described here has so far only been shown to be applicable to a narrow class of problems. Deadlock, for example, is not yet handled. Section 4 reviews some of the limitations and makes comparisons with other work.

2. OUTLINE OF A DEVELOPMENT METHOD FOR ISOLATED PROGRAMS

Isolated, or sequential, programs are those whose environment can be considered to be unchanging: if a value is assigned to a variable, that variable will yield that same value when next referenced. This is not to say that isolated programs run in a machine of their own. Rather, it is the responsibility of an operating system to ensure that the assumptions of noninterference are not violated. The program development method outlined in this section is described more fully in [18]. The

[1] Ada is a registered trademark of the U. S. Department of Defense.

distinguishing feature is the use of postconditions that are predicates of two states.

2.1 Specifications

A specification can be given in terms of a required input–output behavior. It is frequently far easier to write such a specification than it is to provide a realization. Thus, a function can be specified by giving a type clause, precondition, and a postcondition. For example, the smallest element of a set can be found using

$$\begin{aligned} &\mathit{mins}\colon \mathit{Int\text{-}set} \to \mathit{Int};\\ &\mathit{pre\text{-}mins}(s) \triangleq s \neq \{\ \};\\ &\mathit{post\text{-}mins}(s, r) \triangleq r \in s \wedge (\forall e \in s)(r \leq e). \end{aligned} \tag{2.1}$$

The meaning of such a specification is that any putative realization, say f, must satisfy

$$\begin{aligned} \forall s \in \mathit{Int\text{-}set})(&\mathit{pre\text{-}mins}(s)\\ &\Rightarrow f(s) \in \mathit{Int} \wedge \mathit{post\text{-}mins}(s, f(s))). \end{aligned} \tag{2.2}$$

Notice that the input–output relation is given by a predicate. This predicate may allow for more than one result for given inputs; for the time being this is to be interpreted as allowing one of a class of (deterministic) functions to be acceptable as implementations. Preconditions, here and below, are omitted for total functions.

The execution of programs or their parts (referred to generically as "operations") has the effect of changing the values in a state. It would thus be possible to view operations as functions from states to states. Even with sequential programs, it has been found to be advantageous to emphasize that operations cannot change the structure of the state by recording a name for the set of states separately from any auxiliary inputs and outputs. When one considers possible interferences to the state, there are additional arguments against trying to fit operations into the specification mold for functions. Thus, [18] proposes that operations be specified in terms of a state whose structure can be described in an abstract syntax notation. When one is dealing with parallelism, it is worth identifying those parts of the global state that should be "read only." Although in the sequential case this can be specified by a postcondition, the "read only" abbreviation is used in the examples later in this section.

Relatively few problems can be conveniently specified solely in terms of their inputs and outputs. For a system of operations where a result might depend in a complicated way on earlier events, it is necessary to adopt the notion of "state" as shown above. For a specification to be useful, the states must eschew implementation details. It is in the decription of the states that abstract objects like sets and mappings can be used to provide concise and precise specifications.

2.2 Program Design by Data Refinement

The set of states to be used in a specification might be given using abstract syntax and a "data type invariant." For example,

$$\mathit{Partition} = \{S \in (\mathit{El\text{-}set})\text{-}\mathit{set} \mid \mathit{invp}(S)\} \tag{2.3}$$

where

$$\mathit{invp}(p) \triangleq (\forall s1, s2 \in p)(s1 = s2 \vee \mathit{is\text{-}disj}(s1, s2)) \\ \wedge \bigcup p = El \wedge \{\ \} \notin p$$

defines a class of objects each of which is a member of the power set of *El*; each "valid" partition must also satisfy the predicate *invp*, which requires that the contained sets be pairwise disjoint and that their distributed union be the whole set. Formally,

$$\mathit{is\text{-}disj}(s1, s2) \triangleq \sim(\exists e)(e \in s1 \wedge e \in s2); \tag{2.4}$$

$$\bigcup ss \triangleq \{e \mid (\exists s \in ss)(e \in s)\}. \tag{2.5}$$

An abstract state can be represented by one that contains more structure and is closer to the data structures available for the program that is to be developed. A possible representation for elements of *Partition* might be a mapping to some arbitrary *Key* set:

$$Mtok = \{m \in (El \xrightarrow{m} Key) \mid \mathit{invm}(m)\} \tag{2.6}$$

where

$$\mathit{invm}(m) \triangleq \mathbf{dom}\ m = El.$$

The relationship of this representation to the given abstraction can be given by a function that "retrieves" the abstraction from the representation:

$$\mathit{retrp}: Mtok \to Partition;$$

$$\mathit{retrp}(m) \triangleq \{\mathit{retrgrp}(m, k) \mid k \in \mathbf{rng}\ m\}; \tag{2.7}$$

$$\mathit{retrgrp}(m, k) \triangleq \{e \in \mathbf{dom}\ m \mid m(e) = k\}.$$

The existence of many possible representations for the same abstract element is typical and is the reason for documenting the relationship between abstraction and representation by a function from the latter to the former.

For a given class of states and a proposed representation there are two tests to be applied. First, the retrieve function must be total. Second, the representation must be "adequate" in the sense that there must be at least one representation for each valid abstract state:

$$(\forall p \in Partition)((\exists m \in Mtok)(p = \mathit{retrp}(m))). \tag{2.8}$$

The remaining part of a proof by data refinement is to establish that each of the operations on the representation (say *OPM*) models the corresponding operation on the abstraction (say *OPP*). There are various ways in which this can be done. A rule that can be used to relate the postconditions is

$$(\forall m \in Mtok \mid \mathit{pre\text{-}OPM}(m))(\mathit{post\text{-}OPM}(m, m') \\ \Rightarrow \mathit{post\text{-}OPP}(\mathit{retrp}(m), \mathit{retrp}(m'))). \tag{2.9}$$

It is also necessary when employing this rule to ensure that the domain of the modeling operation is sufficient:

$$(\forall m \in Mtok \mid \mathit{pre\text{-}OPP}(\mathit{retrp}(m)))(\mathit{pre\text{-}OPM}(m)). \tag{2.10}$$

An example of a data refinement proof is given in Section 2.5.

2.3 Program Development by Operation Decomposition

A specification normally employs abstract data objects. The method outlined in the last section can be used to design data structures that match the implementation possibilities. Such a design would still be documented in terms of preconditions and postconditions. If the available software (ultimately the programming language) does not possess suitable primitives, the operations must be decomposed into more primitive ones whose eventual realizations will be combined by using language features that combine statements like **if** and **for**. Just as a step of data refinement could be proved correct using rules (2.8)–(2.10) similar requirements can be stated for the use of the main "combinators" available in programming languages.

The simplest way of combining two operations is to execute them one after the other. Suppose a specification of *OP* is given by preconditions and postconditions. Furthermore, assume that *OP* is to be realized by

$$OP1; \quad OP2 \tag{2.11}$$

and that specifications of both of the proposed operations are given in the same format and based on the same states. To prove the realization correct it must be shown that

$$\begin{aligned}
&(\forall \sigma \in \Sigma \mid \textit{pre-OP}(\sigma))(\textit{pre-OP}1(\sigma));\\
&(\forall \sigma \in \Sigma \mid \textit{pre-OP}(\sigma))(\textit{post-OP}1(\sigma, \sigma') \Rightarrow \textit{pre-OP}2(\sigma'));\\
&(\forall \sigma \in \Sigma \mid \textit{pre-OP}(\sigma))(\textit{post-OP}1(\sigma, \sigma') \wedge \textit{post-OP}2(\sigma', \sigma'')\\
&\qquad\qquad \Rightarrow \textit{post-OP}(\sigma, \sigma'')).
\end{aligned} \tag{2.12}$$

These rules look more complex than ones made possible by assuming that postconditions can be predicates of single states alone. For example, [14] uses

$$\{P\}\ OP1\ \{Q\},\ \{Q\}\ OP2\ \{R\} \vdash \{P\}\ OP1;\ OP2\ \{R\}. \tag{2.13}$$

There are a number of reasons, reviewed in [18], for preferring postconditions that can refer directly to the starting state. As larger problems are tackled, the balance would appear to shift from preferring simple rules to expression of the true input–output relation and a collection of simple checks like (2.12).

Similar rules are given in [18] for **if** and **while** statements. The latter rules use an invariant that defines the relationship between the initial state and any that can arise after some number of iterations of the loop body. (Peter Aczel [2] has shown a much more concise form of these rules.)

2.4 Find an Array Index

As an example of a proof by decomposition, a slight generalization of a problem discussed in [32] is considered. Of course, the interesting aspect of this problem is the possibility of employing parallel processes; this is considered in Section 3.3. The specification makes no mention of parallelism since this is an implementation rather than a specification issue. The problem is to find the least index of an array such that the indexed element satisfies some predicate ("*p*").

Rather than following the rigid format of [18], this specification is presented in the form of a skeletal Ada program with a **spec** block in place of the normal **begin**. Thus,

```
function FINDP return RES: Nat is
   globals X: rd array (Nat) of Val;        -- state, read only,
                                            -- view as a mapping
   function P(V: in Val) return Bool;       -- assumed
   N: constant Nat := ...;                  -- set to max in dom x
spec
   post res' = mins({i ∈ dom x | p(x(i))} ∪ {n + 1})
end                                                              (2.14)
```

The relation of the specification shown above to those of [18] should not be difficult to understand. A state might be defined with components named X and N. Here, X is shown as a **global** (cf. "glocon" in [5]), and stating in the postcondition that it cannot be changed is obviated by marking X as "read only." Similarly, the constant N cannot be the target of any assignment. The assertions have used lowercase versions of the identifiers to denote their initial values and primed versions to indicate the final values.

A program that meets this specification can now be developed by decomposing the operation *FINDP* into an initialized loop. Thus,

```
RES := N + 1; CTR := 1;      -- INIT
while CTR ≤ N loop BODY end loop                                 (2.15)
```

The states that are valid for the loop satisfy

$$1 \leq ctr \leq n + 1 \wedge res \leq n + 1. \tag{2.16}$$

The predicate that shows the relationship of the initial state to that after n loop interations is

$$res' = mins(\{i \in \{1{:}ctr\} \mid p(x(i))\} \cup \{res\}). \tag{2.17}$$

Details of this proof are not given here; the realization of "BODY" might be

```
if P(X(CTR)) then RES := CTR; CTR := N + 1
             else  CTR := CTR + 1;
end if                                                           (2.18)
```

2.5 Recording Equivalence Relations

The last section illustrates the need for decomposition proofs. This problem gives us the opportunity to show how the design of data structures works on a practical example and, at the same time, to lay much of the groundwork for the parallel solution discussed in Section 3.4.

The problem is to develop modules that will record an equivalence relation over some fixed set of elements *El*. This is a frequent subproblem of graph processing algorithms but is also useful in more homely situations such as a database of equivalent engineering parts. After initialization the system must be able to record new equivalent pairs (*EQUATE*) and to answer the question whether two things are equivalent (*TEST*). Such answers must, of course, reflect the symmetric, reflexive, and transitive properties of equivalence relations.

Mathematicians would probably find the model (2.3) the most natural basis for a specification. However, it saves some effort if the mapping to keys defined in (2.6) is chosen as the starting point. (A data refinement proof linking these two alternative bases is given in [18].) The specifications can be presented in the form of an Ada "package":

```
package body QREL is
  M: array (El) of Key;      -- view as mapping
  function TEST(E1: in El, E2: in El) return RES: Bool is
     globals M: rd Mtok;
  spec
    post $res' \Leftrightarrow (m(e1) = m(e2))$
  end;
  procedure EQUATE(E1: in El, E2: in El) is
    globals M: rw Mtok;
  spec
    post $m' = m \dagger [e \mapsto m(e2) \mid m(e) = m(e1)]$
  end;
begin
  INIT      -- initialization
     globals M: wr Mtok;
  spec
     post $\mathbf{dom}\ m' = El \wedge (\forall e1, e2 \in \mathbf{dom}\ m')(m'(e1) = m'(e2) \Rightarrow e1 = e2)$
  end
end                                                                    (2.19)
```

Consulting (2.6), it is clear that, as well as ensuring a mapping, or array, from *El* to some *Key* set, it is necessary to preserve the data type invariant *invm* (cf. (2.7)). Clearly, *INIT* establishes the invariant, and *TEST* cannot destroy it since the operation only has read access to *M*. The *EQUATE* operation is required to overwrite ("†") some elements of mapping *m* (in this case with a mapping: from all elements for which $m(e) = m(e1)$ to the value $m(e2)$)—but since the domain is unchanged, the invariant is preserved.

It is now possible to turn to the design. The well-known Fischer–Galler algorithm is built around a data structure that organizes equivalent elements into trees. Trees are represented by a mapping from *El* to *El* in which each element is mapped to one nearer the root. Roots are indicated by not being in the domain of the mapping. Thus

$$Forest = \{m \in (El \xrightarrow{m} El) \mid invf(m)\} \qquad (2.20)$$

where

$$invf(m) \triangleq is\text{-}wellfounded(m).$$

Well-foundedness can be expressed in terms of avoiding infinite descending chains. Section 2.2 requires that the first stage of a proof of data refinement be to provide a retrieve function (cf. (2.7)), here

$$retrm: Forest \to Mtok; \qquad (2.21)$$

$$retrm(f) \triangleq [e \mapsto root(e, f) \mid e \in El]$$

where

$$root: El \times Forest \to El; \qquad (2.22)$$

$$root(e, f) \triangleq \mathbf{if}\ e \notin \mathbf{dom}\ f\ \mathbf{then}\ e\ \mathbf{else}\ root(f(e), f).$$

Given the data-type invariant, it is easy to see that this retrieve function is total.

The question of adequacy becomes

$$(\forall m \in Mtok)(\exists f \in Forest)(m = retrm(f)). \qquad (2.23)$$

It can be argued that a *Forest* can always be constructed by taking one element for each key and making it a root and then making any other elements with the same key map directly to that root. Of course, many other constructions could be used, but to show adequacy only existence is necessary.

Each of the three operations can now be redefined to reflect the chosen representation. For example,

```
function TESTF(E1: in El, E2: in El) return RES: Bool is
  globals F: rd Forest
spec
  post $res' \Leftrightarrow (root(e1, f) = root(e2, f))$
end                                                          (2.24)
```

Following (2.9) and comparing with (2.19) and (2.21), it is necessary to show that

$$\begin{aligned}(root(e1, f) = root(e2, f)) &\Leftrightarrow (retrm(f)(e1) = retrm(f)(e2)) \qquad (2.25)\\ &\Leftrightarrow (root(e1, f) = root(e2, f)).\end{aligned}$$

Notice that the state-change part of the problem is absent here. There is also nothing to be shown for the preconditions (cf. (2.10)) since both of the operations are total.

Similarly,

```
INITF
  globals F: wr Forest;
spec
  post $f' = [\,]$
end                                                          (2.26)
```

establishes the forest invariant and is easily proven to be a model of *INIT*.

Finally,

```
procedure EQUATEF(E1: in El, E2: in El) is
  globals F: rw Forest;
spec
  post $f' = f \dagger [root(e1, f) \mapsto root(e2, f)]$
end                                                          (2.27)
```

The correctness of this operation is shown in [19], where it is also argued that large parts of the proofs can be factored out into a theory of the *Forest* data structure.

As is indicated above, after completion of such a step of data refinement, the operations can be decomposed and the code proved to match specifications (2.26), (2.24), and (2.27). For example, *TESTF* might be coded

```
function TEST(E1: in El, E2: in El) returns Bool;
begin
  return (ROOT(E1) = ROOT(E2))
end                                                          (2.28)
```

with

```
function ROOT(E: in El) returns El is
  T: El;
```

```
begin
  T := E;
  while F(T) ≠ Nil loop
    T := F(T)
  end loop;
  return (T);
end                                                    (2.29)
```

This can be proved correct using the method of Section 2.3.

3. EXTENSIONS TO DEVELOPMENT METHOD TO COPE WITH INTERFERING PROGRAMS

The specifications above permit access to global variables. Providing these programs are run in isolation from any others that might change the values of these variables, all is well. As soon as the possibility of other programs (processes) running in parallel is admitted, there is a danger of "interference." Of more interest are the places where it is required to permit parallel processes to cooperate by changing and referencing the same variables. It is then necessary to show that the interference assumptions of the parallel processes coexist.

Other work has been published in this area, notably [33]. The method explained in this section has a crucial advantage over earlier work. It is an essential part of the method described in Section 2 that, once one stage of development is completed, it is possible to perform the next solely in terms of the inherited specifications; it is never necessary to perform some final test whose failure might expose an erroneous assumption on which work has been built. This would appear to be an essential requirement for a method to be useful for large problems. By documenting the interference assumptions in the way described in Section 3.1, it is possible to preserve this cardinal property of a development method. The proof rules that are required for development steps in the presence of interference are discussed in Section 3.2.

Some readers will find the emphasis on shared variables lamentable. Even CSP enthusiasts (cf. [10]) will concede that some problems do naturally present themselves in terms of shared storage. Furthermore, one of the implementations in Section 3.3 results in a communication form of parallelism. But, most important, the concept of capturing the allowable interference in a specification should be thought of as an approach to parallelism in general. The similarity between [33] on the one hand and [3, 25] on the other supports the hope that the general approach taken here could be the stimulus for a new development method for communicating processes.

Sections 3.3 and 3.4 provide further implementations of the specifications considered in Section 2. Here, the developments use the tasking features of the Ada language to express parallelism.

3.1 Specification of Interfering Programs

The first observation to be made is that the specifications proposed in Section 2 do, in a sense, already cover interfering programs. The sort of nondeterminism that often comes from interference can be adequately subsumed by postconditions that do not determine a unique answer. It has, however, already been shown how recording a design gives rise to a mixture of program constructs and further specifications. It is in documenting parallel solutions to problems that there is a need to control interference.

The basic idea for how to express the specifications of programs that run in an interfering environment is to add rely- and guarantee-conditions. Thus the hidden assumptions in Section 2 are expressed by stating that the programs are permitted to rely on the fact that the global variables will not change. Similarly, other specifications include clauses that require a guarantee that any effects on global variables are constrained in a defined way.

To be more precise, a "rely-condition" is a predicate of two states. The intention of documenting such a predicate is that a program development according to such a specification can assume that, although the global state may alter, the changes will be constrained. Specifically, any state changes made by other processes can be assumed to satisfy the rely-condition. Thus, a very strict rely-condition might require that a global variable does not change, for example,

$$x' = x, \tag{3.1}$$

whereas one of the programs developed below can perform its required function with an assumption that a variable decreases monotonically, for example,

$$t' \leq t. \tag{3.2}$$

Thus, if the process being defined ceases progress for some time, the designer can assume that, when the process resumes, the earlier/current state pair satisfy the rely-condition. Obviously, a rely-condition must be reflexive and transitive.

The design of a process that has write access to global variables has constraints on the way these variables can be handled. The specification includes a "guarantee-condition." This is again a predicate of two states. The interpretation here is that any process must make its state changes in such a way that any other process observing the global variables will only see (time-ordererd) pairs of states that satisfy the guarantee-condition. One example used below constrains the way that a variable may be changed:

$$t' \neq t \Rightarrow (t' < t \wedge satp(x, t')). \tag{3.3}$$

Notice that a process that has "read-only" access has an implicit guarantee-condition that the variable does not change. A guarantee-condition must be reflexive and transitive.

It is useful to compare the rely-condition to a precondition and the guarantee-condition to a postcondition. In both of the former pair of cases, an assumption is recorded on which the developer is invited to depend; if it is violated, there is no specified constraint on the behavior of the program. In the latter pair of cases, a requirement is stated about the behavior of a developed program. This comparison leads to several useful properties. Clearly, all four conditions record behavior only in terms of externally visible entities (mainly the global variables). Furthermore, it is quite legitimate to use a program with a weaker rely-condition or a stronger guarantee-condition than those shown in the specification.

Where no rely- or guarantee-conditions are given, there must be an accepted interpretation; these are as follows:

$$rely\text{-}OP(\sigma, \sigma') = \sigma' = \sigma; \tag{3.4}$$

$$guar\text{-}OP(\sigma, \sigma') = \mathbf{TRUE}. \tag{3.5}$$

Thus, the specifications of Section 1 are interpreted as having a rule like (3.4) relating to the global variables.

It is now clear that there is another advantage to separating the "states" part of a specification. With the acceptance of interference, it is no longer possible to regard an operation as a function from states to states.

3.2 Development of Interfering Programs

If the objective of finding a true development process is to be met, the specifications of any required subcomponents must include rely- and guarantee-conditions, and their coexistence must be proved before the independent development of the processes is undertaken. Furthermore, the subprocesses inherit the interference conditions from the process that they are being used to realize.

Suppose *OP* is to be realized by executing two processes (generalization to more processes is straightforward) in parallel. It must be true that both processes rely on nothing more than *rely-OP* asserts about the globals:

$$\mathit{rely\text{-}OP}(gl, gl') \Rightarrow \mathit{rely\text{-}T_i}((gl, loc), (gl', loc)). \tag{3.6}$$

There is also, in general, a requirement to show that the guarantee-conditions of the parallel processes imply the guarantee-condition of the overall operation; this rule is not required below since the overall guarantee-conditions are all **TRUE**. In addition, the processes must be able to coexist in the sense that each one's guarantee-condition should be at least as strong as the rely-condition of the other (for $i \neq j$):

$$\mathit{guar\text{-}T_i}(\sigma, \sigma') \Rightarrow \mathit{rely\text{-}T_j}(\sigma, \sigma'). \tag{3.7}$$

An obvious aspect of the usability of the tasks is that their preconditions are suitable:

$$\mathit{pre\text{-}OP}(\sigma) \Rightarrow \mathit{pre\text{-}T_i}(\sigma). \tag{3.8}$$

In order to establish correctness it is necessary to find a dynamic invariant that relates the initial state to any that can arise:

$$\mathit{dinv}\colon \Sigma \times \Sigma \rightarrow \mathit{Bool}. \tag{3.9}$$

This is similar to the relational invariant for loops (cf. Section 2.3). The required conditions are

$$\mathit{pre\text{-}OP}(\sigma) \Rightarrow \mathit{dinv}(\sigma, \sigma); \tag{3.10}$$

$$\mathit{dinv}(\sigma\ \sigma') \wedge \mathit{guar\text{-}T_i}(\sigma', \sigma'') \Rightarrow \mathit{dinv}(\sigma, \sigma''). \tag{3.11}$$

It should also be shown that the interference expected by the environment preserves the dynamic invariant:

$$\mathit{dinv}(\sigma, \sigma') \wedge \mathit{rely\text{-}OP}(\sigma', \sigma'') \Rightarrow \mathit{dinv}(\sigma, \sigma''). \tag{3.12}$$

Finally, correctness is given by

$$\mathit{dinv}(\sigma, \sigma') \wedge \bigwedge_i \mathit{post\text{-}T_i}(\sigma, \sigma') \Rightarrow \mathit{post\text{-}OP}(\sigma, \sigma'). \tag{3.13}$$

The above set of rules shows how parallel process creation can be introduced as a stage of program design. There remains the problem of how to develop a program, using normal control constructs, to meet a specification that includes interference conditions. Basically, this requires extension of the rules of Section 2.3 to cover the possibility that the state changes between steps. These rules are discussed in the examples below, as are some new problems relating to data refinement.

3.3 Find an Array Index

A parallel solution to the problem described in Section 2.4 can be pursued if the specification of *FINDP* (2.14) is extended with

rely $x' = x$
guar TRUE (3.14)

To illustrate the use of the rules for decomposing a problem into tasks that can execute in parallel, a first development step is made in which an arbitrary number of task instances is used; each is made responsible for checking a subset of the indices of array X. This step of development can be recorded as follows:

```
function FINDP return RES: Nat;
  globals X: rd array (Nat) of Val;
  function P(V: in Val) return Bool;
  N: constant Nat := ... ;                       -- set to max in dom x
  T: Nat;                                        -- top of search area
  subtype Tinds is constant Nat range 1 .. MAXTSK;
  GRPS: constant array (Tinds) of Nat-set        -- defines for each task
      := ...                                     -- its index set
begin
  T := N + 1:                                    -- INIT
  declare
    task type SEARCH;
    task body SEARCH is
      globals X: rd array (Nat) of Val;
              T: wr Nat;
              GRPS: rd array (Tinds) of Nat-set;
      MINE: constant Els-set := GRPS (SEARCH' index);
    spec
      post $consid(x', t', mine)$
      rely $x' \upharpoonright mine = x \upharpoonright mine \wedge t' \leq t$
      guar $t' \neq t \Rightarrow (t' < t \wedge satp(x, t'))$
    end;
    SEARCHAR: array (Tinds) of SEARCH;
  begin
    end;   -- waits for all subtasks
  return (T);
end                                                                  (3.15)
```

The auxiliary functions are these:

$$satp(x, t) \triangleq t \leq n \Rightarrow p(x(t)); \tag{3.16}$$

$$consid(x, t, s) \triangleq (\forall i \in s)(p(x(i)) \Rightarrow t \leq i). \tag{3.17}$$

Speaking intuitively, each task is forced to *consid*er the set of indices allocated to its (task) index by *GRPS*. The predicate *consid* requires that the variable T, which is tracking the lowest index for which P is true, be as low as the least such (allocated) index. If array X were to change from time to time, it would be impossible to prove anything useful about the *SEARCH* tasks. Furthermore, if *T*—which is global to each task instance—were to hop about, it would not be possible to realize the postcondition. Both clauses of the rely-condition are thus necessary. It is, of course, instances of the same task type that can change the variable *T*, and it is therefore not surprising that the guarantee-condition also requires that any change in *T* be monotonically decreasing. In order to prove the correctness of the effect of the whole array of tasks, it is also necessary to show that only satisfactory values are assigned to *T*; this is the purpose of the *satp* part of the guarantee-condition.

More formally, coexistence can be proved from rules (3.6) and (3.7). The environment can only affect the variable *X* (*T* is local), and thus it is necessary that

$$x' = x \Rightarrow x' \restriction mine = x \restriction mine. \tag{3.18}$$

To check the interaction of the individual task instances, observe that *X* can only be read, so

$$\begin{aligned}(t' \neq t \Rightarrow t' < t \wedge satp(x, t')) \wedge x' = x \\ \Rightarrow x' \restriction mine = x \restriction mine \wedge t' \leq t.\end{aligned} \tag{3.19}$$

In order to establish overall correctness, a "dynamic invariant" must be found which summarizes the relationship between the initial state and that existing at an arbitrary point in the computation. (In a real development it would be preferable to follow the approach used in Section 2.3 and begin with the invariant as an aid in the design of the subcomponents. The use of the dynamic invariant in the design process is illustrated in Section 3.4.) Thus,

$$x' = x \wedge t' \leq t \wedge satp(x', t'). \tag{3.20}$$

As required by (3.10) this is reflexive under the assumption that initially

$$t = n + 1. \tag{3.21}$$

Furthermore, the dynamic invariant can be seen to hold over arbitrary steps of the instances of *SEARCH* by

$$\begin{aligned}& x' = x \wedge t' \leq t \wedge satp(x', t') \\ & \wedge x'' = x' \wedge (t'' \neq t' \Rightarrow t'' \leq t' \wedge satp(x, t'')) \\ & \quad \Rightarrow x'' = x \wedge t'' \leq t \wedge satp(x, t'').\end{aligned} \tag{3.22}$$

With the chosen dynamic invariant it is easy to establish (cf. (3.13)), that the execution of the whole array of tasks achieves

$$satp(x, t_f) \wedge consid(x, t_f, \textbf{union rng } grps). \tag{3.23}$$

The rule for the preconditions of the tasks is vacuously true.

To complete the correctness argument of this stage of development, it is necessary to use an extended form of rule (2.12). Initialization can be seen (cf. (3.15), (3.16)) to establish

$$satp(x, t). \tag{3.24}$$

The rely-condition of *FINDP* and the fact that *T* is local guarantee that this will not be destroyed. The foregoing proof, under the overall rely-condition, has shown that (3.23) holds. Thus, providing that *GRPS* provides a "cover" for the array indices, that is, that

$$\mathbf{dom}\ x = \mathbf{union\ rng}\ grps, \tag{3.25}$$

the overall effect relies on

$$\begin{aligned} &satp(x, t) \wedge consid(x, t, \mathbf{dom}\ x) \\ &\quad \Rightarrow t = mins(\{i \in \mathbf{dom}\ x \mid p(x(i))\} \cup \{n + 1\}). \end{aligned} \tag{3.26}$$

This concludes the justification of (3.15). Notice that the proof applies to the design decisions made and will not have to be reconsidered when further development takes place.

This first step of development can now be used as the starting point for a number of alternative programs. Here, both a maximum parallel and the original program of [33] are discussed since they illustrate different development problems. (It is interesting to note that the sequential program of Section 2.4 is a special case of the multitask solution. Other alternatives include programs that refer to *T* less often than the number of indices for which they are responsible and tasks that spawn further subtasks.)

The development up to (3.15) is now known to meet the overall specification (2.14). Leading on from this, one possible specialization is to have one task instance per array index. This obviates the need for the array *GRPS*. The task specification now becomes

```
task body SEARCH is
  globals X: rd array (Nat) of Val;
          T: rw Nat;
  ME: constant Nat := SEARCH'index;
spec
  post consid(x', t', {me})
  rely x'(me) = x(me) ∧ t' ≤ t
  guar t' ≠ t ⇒ (t' < t ∧ satp(x, t'))
end                                                    (3.27)
```

In addition to the performance improvement from parallelism, some instances of the *SEARCH* process can be made to execute faster by ascertaining whether the current value of *T* is already less than the index (*ME*) for which they are responsible:

```
declare
  LOC: Nat;
begin
  LOC := T;
  if ME < LOC then
    TESTP
     globals X:  rd array (Nat) of Val;
             T:  wr Nat;
             ME: rd Nat;
    spec
      post consid(x', t', {me})
      rely x'(me) = x(me) ∧ t' ≤ t
      guar t' ≠ t ⇒ (t' < t ∧ satp(x, t')))
    end
  end if
end                                                    (3.28)
```

The sequential rule for conditional statements requires that, given the result of the Boolean expression, the postconditions of the arms of the conditional each imply the postcondition of the whole construct. Here, the rule must be extended to cover the interference. Thus, for the **then** case,

$$\begin{aligned} &loc = t_1 \wedge me < loc \wedge \mathit{rely\text{-}SEARCH}(\sigma_1, \sigma_2) \\ &\qquad \wedge \mathit{post\text{-}TESTP}(\sigma_1, \sigma_2) \wedge \mathit{rely\text{-}SEARCH}(\sigma_2, \sigma_3) \\ &\Rightarrow \mathit{consid}(x, t_3, \{me\}), \end{aligned} \tag{3.29}$$

which simplifies to the following (using subscripts only for T, which is the variable for which interference is critical):

$$\begin{aligned} &loc = t_1 \wedge me < loc \wedge t_2 \le t_1 \wedge \mathit{consid}(x, t_2, \{me\}) \wedge t_3 \le t_2 \\ &\Rightarrow \mathit{consid}(x, t_3, \{me\}). \end{aligned} \tag{3.30}$$

In the **else** case

$$\begin{aligned} &loc = t_1 \wedge me \ge loc \wedge \mathit{rely\text{-}SEARCH}(t_1, t_2) \\ &\qquad \Rightarrow \mathit{consid}(x, t_2, \{me\}) \end{aligned} \tag{3.31}$$

simplifies to

$$loc = t_1 \wedge me \ge loc \wedge t_2 \le t_1 \Rightarrow \mathit{consid}(x, t_2, \{me\}). \tag{3.32}$$

This follows from the definition of *consid.*

Having shown that (3.28) is a valid realization of (3.27), we can use the same sort of reasoning to develop *TESTP* into

```
if P(X(ME)) then
  SETT
   globals T:   rw Nat;
           ME: rd Nat;
  spec
    post $t' \le me$
    rely $t' \le t$
    guar $t' \ne t \Rightarrow t' = me \wedge t' < t$
  end
end if                                                        (3.33)
```

In each of these steps, it is easy to see that the guarantee-condition is fulfilled. (Notice that the specification of *SETT* inherits the condition.)

The attentive reader might by now be wondering whether this multistage development has not lead up a blind alley: how is the change required by *SETT* to be realized in the presence of the allowed interference? The concept of a "shared variable" has served for the development so far, but now the time has come to move to more message-oriented communication. The "rendezvous" concept of the Ada language can be used to provide set and read entries to a task. In effect, the idealized behavior of the abstract variable T is being realized by representing it as a value guarded by a monitorlike construct (this coincides with Hoare's original concept in [11]).

To summarize this avenue of development, the main parts of the final Ada program would be

```
function FINDP return Nat is
  function P(V: in Val) return Bool;
  task TOP ... end;
  N: constant N := ... ;
```

```
begin
  TOP.SET (N + 1);
  declare
    task type SEARCH . . . ;
    SEARCHAR: array (X'indices) of SEARCH;
  begin
  end;      -- causes wait
  TOP.RD(R):
  return (R);
end                                                        (3.34)
task body SEARCH is
  ME: constant Nat := SEARCH'index;
  LOC: Nat;
begin
  TOP.RD(LOC);
  if ME < LOC then
    if P(X(ME)) then TOP.SET (ME); end if;
  end if;
end                                                        (3.35)
task body TOP is
  T: Nat:
  TEMP: Nat;
begin
  accept SET(V: in Nat) do T := V end;
  loop
    select
      accept SET(V: in Nat) do TEMP := V; end;
      if TEMP < T then T := TEMP; end if;
    or
      accept RD(RES: out Nat) do RES := T end;
    or
      terminate;
    end select;
  end loop;
end;                                                       (3.36)
```

(It is a comment on the verbosity of the Ada language that the whole program is expressed in about a dozen lines of CSP in [16].)

The development of the maximum parallel program abandoned the shared variable at the eleventh hour. The program given in [33] actually uses shared variables. Reverting to specification (3.15), we can illustrate another form of data refinement. The original form of the *FINDP* program split the index set into even and odd indices. Two tasks were then set working on their respective sets. The interesting point was the method used to minimize unnecessary work. Each task has a global variable in which it can record the index of an array element found to satisfy *P*; each task investigates only the area up to the minimum of the two variables. Clearly, this solution could be generalized to many tasks, and the tasks are also free to read the limit variables less frequently than shown below.

The design of this version of *FINDP* can again be presented as a skeletal program (notice that *GRPS* can again be made implicit):

```
function FINDP return Nat is
  globals X: rd array (Nat) of Val;
  function P(V: in Val) return Bool;
  N: constant Nat := . . . ;
  ET, OT: Nat;      --even and odd limits
```

```
begin
  ET := N + 1; OT := N + 1;
  declare
    task ESEARCH, OSEARCH;
    task body ESEARCH is
      globals X:  rd array (Nat) of Val,
              ET: rw Nat,
              OT: rd Nat;
    EC: Nat;      --local counter
    spec
      post $consid(x', min(et', ot'), even(n))$
      rely $x' \upharpoonright evens(n) = x \upharpoonright evens(n)$
           $\wedge\ et' = et \wedge ot' \leq ot$
      guar $et' \neq et \Rightarrow (et' < et \wedge satp(x, p, et'))$
    end;
    task body OSEARCH is ... mutatis mutandis ... end;
  begin
  end;     -- wait
  return (min(ET, OT));
end                                                        (3.37)
```

The general idea for proving that (3.37) is a valid development with respect to (3.15) is to regard

$$min(et, ot) \tag{3.38}$$

as a representation of the abstract variable *T*. The initialization performs as required by the rules of Section 2.3. Furthermore

$$guar\text{-}ESEARCH(\sigma, \sigma') \Rightarrow guar\text{-}SEARCH(retrT(\sigma), retrT(\sigma')). \tag{3.39}$$

But, unfortunately, the corresponding strengthening of the rely-condition is not true:

$$\sim(min(et', ot') \leq min(et, ot) \Rightarrow et' = et \wedge ot' \leq ot). \tag{3.40}$$

What has gone wrong? In fact, the difficulty encountered here was potentially present in the rules of Section 2.2. The aim behind a development step of data refinement is to consider each operation in isolation and then to argue that the combination of the operations must perform correctly. Consider a design using two modules one after the other (say *OP*1; *OP*2) with specifications in terms of sets. If a list refinement were then chosen, it would be possible to adopt a precondition for *OP*2 that the list must be ordered. Clearly, this would not be an automatic condition on the model of *OP*1 because order could not even be discussed in the more abstract specification where the attempt was made to separate the specifications. There is no difficulty in choosing an appropriate postcondition for the refinement of *OP*1; the only cost is that the proof that the precondition of *OP*2 will not be violated, must be repeated.

It is exactly the same sort of problem in (3.40): it is necessary to repeat the proof of coexistence because the representation relies on properties that could not be stated for the abstraction. There is no difficulty in conducting the revised proof.

A final stage of decomposition now yields

```
task body ESEARCH is
  EC: Nat;     -- local counter
begin
  EC := 2;
  while EC < min(ET, OT) loop
    if P(X(EC)) then ET := EC; end if;
    EC := EC + 2;
  end loop
end                                                        (3.41)
```

Just as above, this can be proved to satisfy the specification in (3.37) by extended forms of the rules in Section 2.3, and the guarantee-condition is again straightforward.

3.4 Recording Equivalence Relations

The problem discussed in Section 2.5 can now be treated using parallel tasks.

The algorithm presented in (2.26), (2.24), and (2.27) is the basic Fischer–Galler algorithm and is far superior in space and time requirements to algorithms designed around simpler data structures. There is, however, a problem that has lead to a number of further developments. The decision embodied in (2.27) is to graft the tree of the first element onto that of the second. But there are sequences of *EQUATE* operations that will make the trees become long, and this is a problem that remains if (2.27) is changed to reverse the order of grafting. The length of the chain from the tips of the tree to the root is obviously going to affect the time taken by *ROOT* to search. There is then an advantage to be gained by cleaning up the tall trees by squashing them down (to short bushes). This must be done, of course, in a way that preserves the same groupings into trees. The published solutions to this problem (e.g., [5]) extend the existing operations to clean up as they perform their main tasks. In an aside in [19] it was observed that running *CLEANUP* as a cooperating process might yield a useful program. At that time, the tools to handle such a development were not available. One of the pleasant surprises in working with the method outlined in Sections 3.1 and 3.2 is that it was found that there has been much less need to coordinate the steps of the parallel tasks than was originally thought to be necessary.

The development to *Forests* in section 2.5 is the basis of the parallel version. But it is not possible simply to continue from the stage of (2.26), etc., because, by default, these were justified under the assumption of no interference.

The basic idea of the parallel solution is to have the *CLEANUP* task always running and to make the *EQUATE* and *TEST* functions available as entries to another task (*OPF*), thus ensuring their mutual exclusion with respect to each other. The two main functions now have to work in an environment where they can only rely on the same basic tree groupings being preserved. *CLEANUP* must guarantee that it will only change nonroot elements and that these will only be changed to point further down the same tree. So far this is fairly easy to handle. The situation is made more complicated (and thus interesting) by the fact that *EQUATE* might well be changing the tree structure while *CLEANUP* is running. Appropriate rely- and guarantee-conditions must be found to govern this interference.

The overall program structure is

```
F: array (Nat) of Nat;
begin
  INITF;
  declare
    task CLEANUPF;
    task OPF is
      entry EQUATEF(E1: in El, E2: in El);
      entry TESTF(E1: in El, E2: in El, RES: out Bool);
    end;
    task spec CLEANUPF is . . . below . . . end;
    task body OPF is
    begin
      loop
        select
          accept EQUATEF( . . . ) spec . . . end;
        or
          accept TESTF( . . . ) spec . . . end;
        end select;
      end loop;
    end;
  begin
    . . . OPF.EQUATEF( , ) . . .
  end
end                                                             (3.42)
```

The operation *INITF* will be run in isolation, and thus it is possible to adopt the development from Section 2.5 simply by making the rely-condition explicit $(f' = f)$.

The postcondition of *EQUATE* in (2.27) requires that exactly one change be made to *F*. For the case where *EQUATE* is to be run in parallel with a *CLEANUP* operation, this is too restrictive. Clearly, the minimum requirement is that the roots change at exactly the required place. This can be expressed by a dynamic invariant (for the interaction of *EQUATE* with *CLEANUP*):

$$(\forall e \in El)(root(e, f') = root(e, f) \vee root(e, f) = root(e1, f) \wedge root(e, f') = root(e2, f)). \quad (3.43)$$

In order to check that some change does take place, the postcondition of *EQUATE* must require that

$$root(e1, f') = root(e2, f). \quad (3.44)$$

The guarantee-conditions of the two processes must be such both that the dynamic invariant is preserved and that (with respect to the rely-conditions) the two processes can be shown to coexist. A sufficient condition is to partition the spheres of change; clearly, *CLEANUP* does not change the roots, while *EQUATE* does not change the "body" of the trees:

$$rootunch(f, f') = (\forall e \in El)(root(e, f') = root(e, f)); \quad (3.45)$$

$$bodyunch(f, f') = (\forall e \in \mathbf{dom}\, f)(f'(e) = f(e)). \quad (3.46)$$

Before giving the final specifications, we must consider one more problem. This is one of the cases where the intention to document a top-down design must not inhibit thinking ahead. Clearly, any tree traversal can only be shown to terminate if the order of elements in the tree is not reversed. This is expressed by a predicate:

$$\mathit{ordpres}\colon \mathit{Forest} \times \mathit{Forest} \to \mathit{Bool}. \tag{3.47}$$

The specifications then become

```
EQUATEF(E1: inEl, E2: inEl) is
  globals F: rw Forest;
spec
  post $root(e1, f') = root(e2, f)$
  rely $rootunch(f, f') \wedge ordpres(f, f')$
  guar $(\forall e \in E1)(f'(e) = f(e)$
                  $\vee\ e = root(e1, f) \wedge e \neq root(e2, f)$
                          $\wedge\ f'(e) = root(e2, f))$                    (3.48)
end
CLEANUPF
  globals F: wr Forest(El)
spec
  post TRUE
  rely $bodyunch(f, f')$
  guar $rootunch(f, f') \wedge ordpres(f, f')$
end                                                                  (3.49)
```

(The detailed proofs of this and other steps can be found in [16]. It is interesting to note (cf. *post-CLEANUPF*) that one possible implementation is to make no change at all in *CLEANUP*. This reflects the fact that its only purpose is optimization and that our specifications must also require some comment about performance if we are to avoid misunderstanding.

As would be expected, the interaction with *TESTF* is simpler since it has only read access to *F*. The specification given in Section 2.5 (cf. (2.24)) is given a rely-condition:

$$\mathit{rootunch}(f, f') \wedge \mathit{ordpres}(f, f'). \tag{3.50}$$

For the interaction of *TESTF* and *CLEANUPF*, a dynamic invariant of **TRUE** suffices (i.e., the overall postcondition follows from *post-TESTF* without relying on some property of the interaction).

This step of development has treated together both data refinement and decomposition into tasks. Is this avoidable? It would appear not. The rely- and guarantee-conditions have no meaning on the more abstract level and thus can only be discussed after the refinement. On the other hand, the development in Section 2.5 overcommits the operations. The possibility of performing a step that did no more than copy the conditions composed with retrieve functions runs afoul of the rule of "active decomposition" proposed in [18].

Once again, having concluded a step of development, we can base a range of developments on the specifications above. One of the interesting points is that valid implementations of the sequential code for *TEST*, *ROOT*, and *EQUATE* can also be shown to satisfy the specifications for the parallel case. For example

```
TESTF(E1: El, E2: El) returns El
declare ROOT1, ROOT2: El;
ROOT1 := ROOTF(E1);    -- could be
ROOT2 := ROOTF(E2);    -- parallel
return (ROOT1 = ROOT2);                                              (3.51)
```

```
EQUATEF
declare ROOT1, ROOT2: E1;
ROOT1 := ROOTF(E1);        -- could be
ROOT2 := ROOTF(E2);        -- parallel
F(ROOT1) := ROOT2;                                          (3.52)
```

These can both be shown to be valid decompositions of their respective (postcondition and guarantee-condition) specifications under the assumption that *ROOTF* satisfies

```
ROOTF(E: El returns RT: El
  globals F: rd Forest
spec
  post $rt' = root(e, f)$
  rely $rootunch(f, f') \wedge ordpres(f, f')$
end                                                         (3.53)
```

Notice that it is necessary to inherit the rely-condition (any subprograms have to accept the interference of their user) but that the guarantee-condition can be dropped since *ROOTF* requires only read access to *F*. The final code for this function is as for the sequential case.

Of more interest is the development of task bodies that satisfy the *CLEANUPF* specification (3.49). An algorithm that makes very local changes to the representation is

```
CLEANUPF1
  declare CUR: El
          NEXT: El;
begin
  loop    -- forever
    for CUR := 1 .. N loop
      if F(CUR) $\neq$ 0 then
          NEXT := F(CUR);
          if F(NEXT) $\neq$ 0 then
            F(CUR) := F(NEXT)
          end if;
      end if;
    end loop;
  end loop;
end                                                         (3.54)
```

In [16], another algorithm is developed, and the further problem of more than one instance of *CLEANUP* is solved.

4. DISCUSSION

It is clear that two examples do not make a development method, and, even though other problems have been worked out, it is necessary to review the limitations of what has been presented in Section 3. First, the proof rules used in Section 3 are far less stable than those in Section 2. These latter, or rather a argument. Modal logic is now being widely used for such problems [1, 9, 23, 27, 35]. The extension of a modal logic to cover binary relations may yield some interesting insights. In particular, more general forms of rely- and guarantee-conditions should be definable, and perhaps some unification of proof methods found.

Some synchronization problems appear to be handled well by predicates of streams (cf. [40]); it is not yet clear how to combine the strengths of the alternative approaches.

In spite of these limitations, the general approach set out above does appear to be worthy of further study. Apart from the advantage of offering a true development method, a number of other points are encouraging. The rules used for proofs without interference are natural specializations of those for the more general case. Furthermore, the concept of merging of atomic operations is not forced into the discussion: true parallelism can be considered as satisfying the rely- and guarantee-conditions. Last, a point familiar from the most recent work on parallelism: there is very strong pressure to adopt language features that make the degree of interference controllable.

5. POSTSCRIPT

In the time between the first submission of this paper and its revision a considerable amount of additional work has been done. In particular, [16] justifies the proof rules for parallel programs by relating them to a semantic model of the language. The same monograph also indicates how [41] can be thought of as showing an "interference" approach to CSP.

Two recent publications are relevant to the material here. Lamport [23] tackles the problem of providing a development method for parallel programs. He uses $\{P\}\ S\ \{Q\}$ to mean that, if execution is begun anywhere in S with predicate P true, P will remain true and Q will be true at the end of execution. The method is difficult to apply and serves to convince this author of the need for predicates of two states.

The referee drew attention to [8], which uses "interference predicates." That paper considers cyclic programs such as operating systems. The predicates used in proofs make the notion of time explicit.

ACKNOWLEDGMENTS

The work described here was undertaken during a most rewarding sojourn in Oxford, and it is a pleasure to be able to thank Tony Hoare for these two years. Lockwood Morris went to great lengths to understand and comment on the first draft. Further valuable feedback has been provided by Ole-Johan Dahl, Ian Mearns, and Susan Owicki. The ideas have also been discussed at two meetings of IFIP WG2.3.

REFERENCES

Note. References [4, 6, 15, 17, 24, 26, 30, 34, 36, 38] are not cited in the text.

1. ABRIAL, J.-R., AND SCHUMAN, S.A. Non-deterministic system specification. In *Semantics of Concurrent Computation*, G. Kahn (Ed.). Springer-Verlag, New York, 1979, pp. 34–50.
2. ACZEL, P. A note on program verification. Private communication, Jan. 1982.

3. APT, K.R., FRANCEZ, N., AND DE ROEVER, W.P. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst. 2*, 3 (July 1980), 359–385.
4. BURSTALL, R.M. Program proving as hand simulation with a little induction. In *Proceedings, IFIP Congress 1974*. Elsevier North-Holland, New York, 1974, pp. 308–312.
5. DIJKSTRA, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
6. DIJKSTRA, E.W., LAMPORT, L., MARTIN, A.J., SCHOLTEN, C.S., AND STEFFENS, E.F.M. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966–975.
7. FLOYD, R.W. Assigning meanings to programs. In *Proceedings of 19th Symposium on Applied Mathematics*. American Mathematical Society, Providence, R.I., 1967, pp. 19–31.
8. FRANCEZ, N., AND PNUELI, A. A proof method for cyclic programs. *Acta Inf. 9*, 2 (Apr. 1978), 133–157.
9. HAILPERN, B., AND OWICKI, S. Verifying network protocols using temporal logic. Private communication, 1981.
10. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
11. HOARE, C.A.R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.
12. HOARE, C.A.R. Proof of correctness of data representations. *Acta Inf. 1*, 4 (Nov. 1972), 271–281.
13. HOARE, C.A.R. Proof of a program: FIND. *Commun. ACM 14*, 1 (Jan. 1971), 39–45.
14. HOARE, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580, 583.
15. JACKSON, M.A. *System Development*. Prentice-Hall International, Englewood Cliffs, N.J., 1982.
16. JONES, C.B. Development methods for computer programs including a notion of interference. Tech. Rep. PRG 25, Programming Research Group, Oxford Univ., Oxford, Eng., 1981.
17. JONES, C.B. Towards more formal specifications. In *Software Engineering: Entwurf und Spezifikation*, C. Floyd and H. Kopetz (Eds.). B.G. Teubner, Stuttgart, W. Germany, 1981, pp. 14–45.
18. JONES, C.B. *Software Development: A Rigorous Approach*. Prentice-Hall International, Englewood Cliffs, N.J., 1980.
19. JONES, C.B. Constructing a theory of a data structure as an aid to program development. *Acta Inf. 11*, 2 (Jan. 1979), 119–137.
20. JONES, C.B. Formal development of programs. Tech. Rep. TR 12.117, IBM Hursley Laboratory, England, June 1973.
21. JONES, C.B. Formal development of correct algorithms: An example based on Earley's recogniser. In Proceedings of an ACM Conference on Proving Assertions about Programs (Las Cruces, N.M., Jan. 6–7, 1972). Published as combined issue: *SIGPLAN Notices* (ACM) 7, 1 (Jan. 1972), and *SIGACT News* 14 (Jan. 1972), 150–169.
22. KAHN, G. A preliminary theory for parallel programs. Tech. Rep. 6, IRIA, Le Chesnay, France, 1973.
23. LAMPORT, L. The "Hoare logic" of concurrent programs. Acta Inf. *14*, 1 (June 1980), 21–37.
24. LAUER, P.E. Consistent formal theories of the semantics of programming languages. Tech. Rep TR25.121, IBM Laboratory Vienna, Nov. 1971.
25. LEVIN, G.M., AND GRIES, D. A proof technique for communicating sequential processes. *Acta Inf. 15*, 3 (June 1981), 281–302.
26. MANNA, Z. Properties of programs and the first-order predicate calculus. *J. ACM 16*, 2 (Apr. 1969), 244–255.
27. MANNA, Z., AND PNUELI, A. The modal logic of programs. In *Proceedings, Sixth International Conference on Artificial Languages and Programming* (Graz, Austria, July 1979), pp. 385–409.
28. MILNER, R. *A Calculus of Communicating Systems*. Springer-Verlag, New York, 1980.
29. MILNER, R. An algebraic definition of simulation between programs. Tech. Rep. AIM-142, Computer Science Dept., Stanford Univ., Stanford, Calif., Feb. 1971.
30. MORRIS, J.H., JR. A correctness proof using recursively defined functions. In *Formal Semantics of Programming Languages*, R. Rustin (Ed.). Prentice-Hall, Englewood Cliffs, N.J., 1972.
31. NAUR, P. Proof of algorithms by general snapshots. *BIT 6* (1966), 310–316.
32. OWICKI, S.S. Axiomatic proof techniques for parallel programs. Tech. Rep. TR 75-251, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., 1975.
33. OWICKI, S., AND GRIES, D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM 19*, 5 (May 1976), 279–285.
34. PLOTKIN, G.D. A power domain construction. *SIAM J. Comput. 5*, 3 (Sept. 1976), 452–487.

35. PNUELI, A. The temporal semantics of concurrent programs. In *Semantics of Concurrent Computation*, G. Kahn (Ed.). Springer-Verlag, New York, 1979, pp. 1–20.
36. SMYTH, M.B. Power domains. *J. Comput. Syst. Sci. 16* (1978), 23–36.
37. U.S. DEPT. OF DEFENSE. *Reference Manual for the Ada Programming Language* (Proposed standard document). July 1980.
38. VON LAMSWEERDE, A., AND SINTZOFF, M. Formal derivation of strongly correct parallel programs. Tech. Rep. R338, MBLE, Belgium, Oct. 1976.
39. WIRTH, N. Program development by stepwise refinement. *Commun. ACM 14*, 4 (Apr. 1971), pp. 221–227.
40. ZHOU, C.C., AND HOARE, C.A.R. Partial correctness of Communicating sequential processes. In *Proceedings, 2d International Conference on Distributed Computing Systems* (Apr. 1981).
41. ZHOU, C.C., AND HOARE, C.A.R. Partial correctness of communicating processes and protocols. Tech. Rep. PRG-20, Programming Research Group, Oxford Univ., Oxford, Eng., 1981.

Received November 1980; revised March and August 1982; accepted October 1982