FORMAL SPECIFICATION AND DESIGN OF PROGRAMS*

C.B. Jones

Rapporteurs: Mr. D.R. Brownbridge  
Mr. G.D. Parrington

Abstract:

The example of the Fischer/Galler algorithm is used to show how a formal specification can be used as the basis for a program design. Particular emphasis is given to design by data refinement. Extensions of the method to cope with parallelism are also considered.

This paper is a cut-down version of "Tentative Steps Towards a Development Method for Interfering Programs", which is to be published in ACM Transactions on Programming Languages and Systems.

Introduction

A brief review of the history of attempts to formalise the development of sequential (isolated) programs can be given. The first results to appear were concerned with correctness proofs for complete programs and normally concentrated on trivial data structures like natural numbers (cf. [33], [8] and [11]). Subsequent papers showed how the proof rules could be used in a design process - in this way a proof could be used to justify one step of design before development of the final code took place (cf. [12], [39] and [6]). The wider application of such ideas became possible with the study of abstract data types and their refinement (cf. [30] and [13]). The development method which evolved through [17], [16] and [20] mirrors this development but uses post-conditions which are predicates of the initial and final states. This method is outlined in Section 1 below. The emphasis nowadays is more on a "rigorous method" which 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 check-lists of results (based on formal rules) as an integral part of the development 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 micro-processors and distributed systems has given extra impetus to the study of programs which run other than in isolation. The term "tightly coupled" is applied to systems which 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. [23], [31] or [40]). Unfortunately, some situations force consideration of shared variables. Here, both loosely and tightly-coupled systems will be regarded as interfering (non-isolated) 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 (1979) and loosely coupled systems are covered in (1983), (1983), (1983) and (1983). The approaches are characterised 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 post-facto proof methods to the development process rather difficult. This paper shows how certain problems can be tackled by a development method which is a fairly natural extension of that described in (1983). The basic idea is to add to a specification a precise statement of its interference: a "rely-condition" defines assumptions which can be made in program development; a "guarantee-condition" places requirements on the interference a would-be implementation can generate. The proof rules which 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: having completed the proof at one stage of development, 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 which are activated by communication activity. It would appear to be an important contribution to the development described in Section 2 that it uses predicates of pairs of states.

The presentation of parallel programs is given in the syntax of the Ada language - this choice is based on the useful properties of the "rendezvous" concept (1983). 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 3 reviews some of the limitations and comparisons with other work.

1. 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 non-interference are not violated. The program development method outlined in this section is described more fully in (1983). The distinguishing feature is the use of post-conditions which are predicates of two states.

1.1 Specifications

A specification can be given in terms of a required input-output behaviour. It is frequently far easier to write such a specification than it is to provide a realisation. Thus a function can be specified by giving a type clause, a pre-condition and a post-condition.
For example, the smallest element of a set can be found using:

\[
\begin{align*}
\text{mins: Int-set} & \rightarrow \text{Int} \\
\text{pre-mins(s)} & \triangleq s \neq \emptyset \\
\text{post-mins(s,r)} & \triangleq r \in s \land (\forall r \in s)(r \leq r)
\end{align*}
\]

The meaning of such a specification is that any putative realisation, say 'f', must satisfy:

\[
(\forall s \in \text{Int-set})(\text{pre-mins}(s) \Rightarrow f(s) \in \text{Int} \land \text{post-mins}(s,f(s)))
\]

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. Pre-conditions, here and below, will be 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 emphasise 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 considering possible interferences to the state, there are additional arguments against trying to fit operations into the specification mould for functions. Thus, [20] proposes that operations be specified in terms of a state whose structure can be described by an abstract syntax notation. When dealing with parallelism, it is worth identifying those parts of the global state which should be "read only". Although in the sequential case this can be specified by a post-condition, the "read only" abbreviation will be 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 description of the states that abstract objects like sets and mappings can be used to provide concise and precise specifications.

1.2 Program Design by Data Refinement

The set of states to be used in a specification might be given by an abstract syntax and a "data type invariant". For example:
Partition = \{S_\delta(El-set)-set \mid invp(S)\} \quad (1-3)

where

\[
invp(p) \triangleq (\forall s_1, s_2 \in p)(s_1 = s_2 \lor is-disj(s_1, s_2)) \land \\
\cup p = El \land \{\} \notin p
\]

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 are pairwise disjoint and that their distributed union is the whole set. Formally:

\[
is-disj(s_1, s_2) \triangleq (\exists e)(e \in s_1 \land e \in s_2) \land \\
\{s \Delta \{e \mid (\exists s \in s)(e \in s)\}\}
\]

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

\[
Mtok = \{m: El \rightarrow Key \mid invm(m)\} \quad (1-6)
\]

where

\[
invm(m) \triangleq dom m = El
\]

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

\[
retrp: Mtok \rightarrow Partition \quad (1-7)
\]

\[
retrp(m) \triangleq \{retrgrp(m, k) \mid kern m\}
\]

\[
retrgrp(m, k) \triangleq \{e \in dom m \mid m(e) = k\}
\]

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. Firstly, the retrieve function must be total. Secondly, the representation must be "adequate" in the sense
that there must be at least one representation for each valid
abstract state:

\[(\forall p \in \text{Partition})(\exists m \in \text{Mtok})(p = \text{retrp}(m))\]  \hspace{1cm} (1-8)

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 which can be used to
relate the post-condition is:

\[(\forall m \in \text{Mtok})(\text{pre-OPM}(m) \land (\text{post-OPM}(m,m') = \text{post-OPP}(\text{retrp}(m),\text{retrp}(m'))))\]  \hspace{1cm} (1-9)

Employing this rule it is also necessary to ensure that the domain of
the modelling operation is sufficient:

\[(\forall m \in \text{Mtok})(\text{pre-OPP}(\text{retrp}(m)))(\text{pre-OPM}(m))\]  \hspace{1cm} (1-10)

An example of a data refinement proof is given in section 1.5

1.3 Program Development by Operation Decomposition

A specification will normally employ abstract data objects. The
method outlined in the last section can be used to design data
structures which match the implementation possibilities. Such a
design would still be documented in terms of pre- and
post-conditions. If the available software (ultimately the
programming language) does not possess suitable primitives, the
operations must be decomposed into more primitive ones whose eventual
realisations will be combined by using language features which
combine statements like 'if' and 'for'. Just as a step of data
refinement could be proved correct using the rules of 1-8/1-10,
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 pre-
and post-conditions. Furthermore, assume that 'OP' is to be realised
by:

\[\text{OP1; OP2}\]  \hspace{1cm} (1-11)

and that specifications of both of the proposed operations are given
in the same format and based on the same states. To be correct it
must be shown that:
\[(\forall \sigma \in \Sigma | \text{pre-OP}(\sigma))(\text{pre-OP1}(\sigma))\]
\[(\forall \sigma \in \Sigma | \text{pre-OP}(\sigma))(\text{post-OP1}(\sigma,\sigma') \Rightarrow \text{pre-OP2}(\sigma'))\]
\[(\forall \sigma \in \Sigma | \text{pre-OP}(\sigma))(\text{post-OP1}(\sigma,\sigma') \land \text{post-OP2}(\sigma',\sigma'') \Rightarrow \text{post-OP}(\sigma,\sigma''))\]

These rules look more complex than ones made possible by assuming that post-conditions can be predicates of single states alone. For example, \[1/1\] uses:

\[\{P\} \text{ OP1 } \{Q\}, \quad \{Q\} \text{ OP2 } \{R\} \quad \{P\} \text{ OP1; OP2 } \{R\}\]

There are a number of reasons, reviewed in \[2/0\], for preferring post-conditions which can directly refer to the starting state. As larger problems are tackled, the balance would appear to shift from preferring simple rules to expressions of the true input-output relation and a collection of simple checks like 1-12.

Similar rules are given in \[2/0\] for IF and WHILE statements. The latter rules use an invariant which defines the relationship between the initial state and any that can arise after some number of iterations of the loop body. (Peter Aczel (\[1/\]) has shown a much more concise form of these rules).

1.4 Find an Array Index  
(Section omitted)

1.5 Recording Equivalence Relations

This problem gives 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 2.4.

The problem is to develop modules which will record an equivalence relation over some fixed set of elements (\('E1'\)). This is a frequent sub-problem of graph processing algorithms but is also useful in more homely situations such as a database of equivalent engineering parts. After initialisation the system must be able to record new equivalent pairs (\('EQUATE'\)) and to be able to answer the question whether two things are equivalent (\('TEST'\)). Such answers must, of course, reflect the symmetric, reflexive properties of equivalence relations.

Mathematicians would probably find the model of 1-3 the most natural basis for a specification. It will, however, save some effort if the mapping to keys defined in 1-6 is chosen as the starting
point. (A data refinement proof linking these two alternative bases is given in /20/.) The specifications can be presented in the form of an Ada "package":

```ada
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' ⇔ (m(e1) = m(e2))
        end;

    procedure EQUATE (El: in EI, E2: in EI) is
        globals M: rw Mtok;
        spec
        post m' = m ⊕ [e → m(e2) | m(e) = m(e1)]
        end;

begin

    INIT
        -- initialisation

        globals M: wr Mtok;
        spec
        post dom m' = El ∧ (∀e1, e2: dom m'(e1) = m'(e2) ⇒ e1 = e2)
        end

end
```

Consulting 1-6, 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. 1-7). 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 which organises 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:

\[
\text{Forest} = \{ m \in (\text{El} \to \text{El}) \mid \text{invf}(m) \} \tag{1-20}
\]

where

\[
\text{invf}(m) \triangleq \text{is-wellfounded}(m)
\]

Well foundedness can be expressed in terms of avoiding infinite descending chains. Section 1.2 requires that the first stage of a proof of data refinement is to provide a retrieve function (cf. 1-7), here:

\[
\text{retrm}: \text{Forest} \to \text{Mtok} \tag{1-21}
\]

\[
\text{retrm}(f) \triangleq \{ e \in \text{root}(e, f) \mid e \in \text{El} \}
\]

where

\[
\text{root}: \text{El} \times \text{Forest} \to \text{El} \tag{1-22}
\]

\[
\text{root}(e, f) \triangleq \begin{cases} 
    e & \text{if } e \notin \text{dom} f \\
    \text{else} \text{root}(f(e), f)
\end{cases}
\]

Given the data type invariant, it is easy to see that this retrieve function is total. The question of adequacy becomes:

\[
(\forall m \in \text{Mtok})(\exists f \in \text{Forest})(m = \text{retrm}(f)) \tag{1-23}
\]

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' \iff (root(e1,f) = root(e2,f))
  end

Following 1-9 and comparing with 1-19, 1-21 it is necessary to show that:

\[(\text{root}(e1,f) = \text{root}(e2,f)) \iff (\text{retrm}(f)(e1) = \text{retrm}(f)(e2))\] (1-25)

\[(\text{root}(e1,f) = \text{root}(e2,f))\]

Notice that the state change part of the problem is absent here. There is also nothing to be shown for the pre-conditions (cf. 1-10) since both of the operations are total.

Similarly:

INITF (1-26)

globals F : wr Forest;
spec
  post f' = [ ]
end

establishes the forest invariant and is easy to prove a model of 'INIT'.

Finally,

procedure EQUATEF (E1: in El, E2: in R1) is
  globals F: rw Forest;
  spec
    post f' = f \uparrow [\text{root}(e1,f) \rightarrow \text{root}(e2,f)]
end
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 the specifications 1-26, 1-24 and 1-27. For example 'TESTF' might be coded:

```plaintext
function TEST (E1: in El, E2: in El) returns Bool; (1-28)
begin
  return (ROOT(E1) = ROOT(E2))
end
with:

function ROOT (E: in El) returns El is (1-29)
  T: El;
begin
  T := E;
  while F(T) ≠ Nil loop
    T := F(T)
  end loop;
  return (T)
end
```

this can be proved correct using the method of section 1.3.

2. 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 which 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 /35/. The method explained in this section has a crucial advantage over earlier work. It is an essential part of the method described in section 1 that having completed one stage of development, 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 assumption in the way described in section 2.1, it is possible to preserve this cardinal property of a development method. The proof rules which are required for development steps in the presence of interference are discussed in section 2.2.

Some readers will find the emphasis on shared variables lamentable. Even CSP enthusiasts (cf. /15/) will concede that some problems do naturally present themselves in terms of shared storage. Furthermore, one of the implementations in section 2.3 will result in a communication form of parallelism. But, most importantly, the concept of capturing the allowable interference in a specification should be thought of as an approach to parallelism in general. The similarity between /35/ on the one hand and /4/, /27/ 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 2.3 and 2.4 provide further implementations of the specifications considered in Section 1. Here, the developments use the tasking features of the Ada language to express parallelism.

2.1 Specification of Interfering Programs

The first observation to be made is that the specifications proposed in section 1 do, in a sense, already cover interfering programs. The sort of non-determinism which often comes from interference can be adequately subsumed by post-conditions which 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 expressing the specifications of programs which run in an interfering environment is to add rely- and guarantee-conditions. Thus the hidden assumptions in section 1 will be expressed by stating that the programs are permitted to rely on the fact that the global variables will not change. Similarly, other specifications will include clauses which require a guarantee that any effects on global variables will be constrained in a defined way.

More precisely, a "rely-condition" will be 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:

\[ x' = x \]  (2-1)

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

\[ t' \leq t \]  (2-2)

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 which 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 ordered) pairs of states which satisfy the guarantee-condition. Notice that a process which 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 pre-condition and the guarantee-condition to a post-condition. In both of the former pair of cases, an assumption is recorded on which the developer is invited to depend: if violated there is no specified constraint on the behaviour of the program. In the latter pair of cases, a requirement is stated about the behaviour of a developed program. This comparison leads to several useful properties. Clearly, all four conditions record behaviour 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:

\[ \text{rely-OP}(\sigma, \sigma') = \sigma' = \sigma \]  (2-4)
\[ \text{guar-OP}(\sigma, \sigma') = \text{TRUE} \]  (2-5)

Thus, the specifications of section 1 are interpreted as having a rule like 2-4 relating to the global variables.

It is now clear that there is another advantage of 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.
2.2 Development of Interfering Programs

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

Suppose 'OP' is to be realised by executing two processes (generalisation 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:

\[
\text{rely-OP}(gl, gl') = \text{rely-}T_i((gl, loc), (gl', loc)) \tag{2-6}
\]

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 co-exist in the sense that each one's guarantee-condition should be at least as strong as the rely-condition of the other (for \(i \neq j\)):

\[
\text{guar-}T_i(\sigma, \theta) = \text{rely-}T_j(\sigma, \sigma') \tag{2-7}
\]

An obvious aspect of the usability of the tasks is that their pre-conditions are suitable:

\[
\text{pre-OP}(\sigma) = \text{pre-}T_i(\sigma) \tag{2-8}
\]

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

\[
dinv : \Sigma \times \Sigma \rightarrow \text{Bool} \tag{2-9}
\]

This is similar to the relational invariant for loops (cf. section 1.3). The required conditions are:

\[
\text{pre-OP}(\sigma) = \text{dinv}(\sigma, \sigma) \tag{2-10}
\]

\[
\text{dinv}(\sigma, \sigma') \land \text{guar-}T_i(\sigma', \sigma'') = \text{dinv}(\sigma, \sigma'') \tag{2-11}
\]

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

\[
\text{dinv}(\sigma, \sigma') \land \text{rely-OP}(\sigma', \sigma'') = \text{dinv}(\sigma, \sigma'') \tag{2-12}
\]
Finally correctness is given by:

\[ \text{dinv}(\sigma, \sigma') \wedge \text{post-}T_1(\sigma, \sigma') \Rightarrow \text{post-OP}(\sigma, \sigma') \]  

(2-13)

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 which includes interference conditions. Basically, this will require extension to the rules of section 1.3 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.

2.3 Find an Array Index  (Section omitted)

2.4 Recording Equivalence Relations

The problem discussed in section 1.5 can now be treated using parallel tasks.

The algorithm presented in 1-26, 1-24 and 1-27 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 which has led to a number of further developments. The decision embodied in 1-27 is to graft the tree of the first element onto that of the second. There are, however, sequences of 'EQUATE' operations which will make the trees become long and this is a problem which remains if 1-27 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 which preserves the same groupings into trees. The published solutions to this problem (e.g. /6/) 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 2.1 and 2.2 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 1.5 will be the basis of the parallel version. But it will not be possible to simply continue from the stage of 1-26 etc. because, by default, these were justified under the assumption of no interference.

The basic idea of the parallel solution will be 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 will 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 non-root 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:

\[
F: \text{ array (Nat) of Nat; (2-42)}
\]

begin

INITF;

declare

(task CLEANUPF;

\text{task OPF is}

entry EQUATEF(E1: in E1, E2: in E1);

entry TESTF(E1: in E1, E2: in E1, RES: out Bool);

end;

\text{task spec CLEANUPF is ... below ... end;}

\text{task body OPF is}

begin

\text{loop}

select

\text{accept EQUATEF (...) spec ... end;}

or

\text{accept TESTF (...) spec ... end;}

\text{end select;}

\text{end loop;}

\text{end;}

begin

\text{...OPF.EQUATEF( , ) ...}

\text{end}

end

163
The operation 'INITF' will be run in isolation and thus it is possible to adopt the development from section 1.5 simply by making the rely-condition explicit ($f' = f$).

The post-condition of EQUATE in 1-27 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):

$$(\forall e \in E)(\text{root}(e,f') = \text{root}(e,f) \lor \text{root}(e,f') = \text{root}(e_1,f) \land \text{root}(e,f) = \text{root}(e_2,f)) \quad (2-43)$$

In order to check that some change does take place, the post-condition of EQUATE must require:

$$\text{root}(e_1,f') = \text{root}(e_2,f) \quad (2-44)$$

The guarantee-conditions of the two processes must be such that both 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:

$$\text{rootunch}(f,f') = (\forall e \in E)(\text{root}(e,f') = \text{root}(e,f)) \quad (2-45)$$

$$\text{bodyunch}(f,f') = (\forall e \in \text{dom } f)(f'(e) = f(e)) \quad (2-46)$$

Before giving the final specifications, one more problem must be considered. 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:

$$\text{ordpres}: \text{Forest} \times \text{Forest} \rightarrow \text{Bool} \quad (2-47)$$

The specifications then become:
EQUATEF(E1: in E1, E2: in E1) is

\[ (2-48) \]

\[ \text{globals } F: \text{rw Forest; } \]

\[ \text{spec} \]

\[ \text{post } \text{root(e1,f')} = \text{root(e2,f')} \]

\[ \text{rely } \text{rootunch(f,f')} \land \text{ordpres(f,f')} \]

\[ \text{guar} \ (\forall e \in E1)(f'(e) = f(e) \lor \]

\[ e=\text{root(e1,f)} \land e \neq \text{root(e2,f)} \land \]

\[ f'(e) = \text{root(e2,f)} ) \]

\[ \text{end} \]

CLEANUPF

\[ (2-49) \]

\[ \text{globals } F: \text{wr Forest(E1) } \]

\[ \text{spec} \]

\[ \text{post } \text{TRUE} \]

\[ \text{rely } \text{bodyunch(f,f')} \]

\[ \text{guar} \ \text{rootunch(f,f')} \land \text{ordpres(f,f')} \]

\[ \text{end} \]

(The detailed proofs of this and other steps can be found in \[22\].

It is interesting to note that (cf. post-CLEANUPF) 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 only has read access to F. The specification given in section 1.5 (cf. 1-24) is given a rely-condition:

\[ \text{rootunch(f,f')} \land \text{ordpres } (f,f') \]

(2-50)

For the interaction of TESTF and CLEANUPF, a dynamic invariant of TRUE suffices (i.e. the overall post-condition follows from post-TESTF without relying on some property of the interaction).
This step of development has treated both data refinement and decomposition into tasks together. Is this avoidable? It would appear not. The rely- and guarantee-conditions have no meaning on the more abstract level and can thus only be discussed after the refinement. On the other hand, the development in section 1.5 overcommits the operations. The possibility to perform a step which did no more than copy the conditions composed with retrieve functions runs foul of the rule of "active decomposition" proposed in [20/].

Once again, having concluded a step of development, it is possible to 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:

**TEST**

\[
\text{TESTF (E1:E1, E2: E1) returns E1} \tag{2-51}
\]

\[
\text{declare ROOT1, ROOT2: E1;}
\]

\[
\text{ROOT1 := ROOTF(E1); -- could be}
\]

\[
\text{ROOT2 := ROOTF(E2); -- parallel}
\]

\[
\text{return (ROOT1 = ROOT2);}
\]

\[
\text{EQUATEF} \tag{2-52}
\]

\[
\text{declare ROOT1, ROOT2: E1;}
\]

\[
\text{ROOT1 := ROOTF(E1); -- could be}
\]

\[
\text{ROOT2 := ROOTF(E2); -- parallel}
\]

\[
\text{F(ROOT1) := ROOT2;}
\]

These can both be shown to be valid decompositions of their respective (post- and guarantee-conditions) specifications under the assumption that 'ROOTF' satisfies:

\[
\text{ROOTF (E:E1) returns RT:E1} \tag{2-53}
\]

\[
\text{globals F: rd Forest}
\]

\[
\text{post \hspace{1cm} rt' = root(e,f)}
\]

\[
\text{rely \hspace{1cm} rootunch(f,f') \land ordpres(f,f')}
\]

Notice that it is necessary to inherit the rely-condition (any sub-programs 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 which satisfy the 'CLEANUP' specification 2-49. An algorithm which makes very local changes to the representation is:

CLEANUPF1  

\[
\begin{align*}
\text{declare} & \quad \text{CUR: E1;} \\
& \quad \text{NEXT: E1;} \\
\text{begin} \\
& \quad \text{loop} \quad \quad \text{- - forever} \\
& \quad \quad \text{for} \quad \text{CUR} = 1 .. N \text{ loop} \\
& \quad \quad \quad \text{if} \quad F(\text{CUR}) \neq 0 \text{ then} \\
& \quad \quad \quad \quad \text{NEXT} := F(\text{CUR}); \\
& \quad \quad \quad \quad \quad \text{if} \quad F(\text{NEXT}) \neq 0 \text{ then} \\
& \quad \quad \quad \quad \quad \quad \text{F(\text{CUR})} := \text{F(\text{NEXT})} \\
& \quad \quad \quad \quad \quad \text{endif} \\
& \quad \quad \quad \quad \text{endif} \\
& \quad \text{end loop} \\
& \quad \text{end loop} \\
& \text{end}
\end{align*}
\]

In /22/, another algorithm is developed and the further problem of more than one instance of CLEANUP is solved.

3. Discussion

It is clear that a few 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 2. Firstly, the proof rules used in section 2 are far less stable than those in section 1. These latter, or rather a fuller set in /20/, have stood the test of use in many examples. The contribution here must be seen as showing an existence proof for the ability to specify interference. Further use of this idea will almost certainly suggest new ways of documenting interference.
Handling deadlock will require an extension to the rules contained in this paper. Those examples tried have appeared to require some form of temporal argument. Modal logic is now being widely used for such problems (/3/, /29/, /37/, /10/ or /24/). 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 synchronisation problems appear to be handled well by predicates of streams (cf. /40/), but it is not clear yet 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 specialisations 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. Lastly, a point familiar from most recent work on parallelism: there is very strong pressure to adopt language features which make the degree of interference controllable.

Postscript

In the time between the first submission of this paper and its revision, a considerable amount of further work has been done. In particular, /22/ 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: /24/ tackles the problem of providing a development method for parallel programs. Lamport 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 /9/ which uses "interference predicates". The paper considers cyclic programs like operating systems. The predicates used in proofs make the notion of time explicit.
Acknowledgements

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. Lookwood 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


170


DISCUSSION 1

Professor Randell enquired whether Professor Jones was using his design to debug his specification in this instance.

Professor Jones replied that any top-down design technique really needed some iteration and that in some sense errors in software were of minor importance - if errors were totally intolerable perhaps a computer should not be used at all. Given the accepted reliability limits, he suggested we still need usable systems without gratuitous complexity.

Dr. J. Horning wondered whether it was harder to get a specification or to use it once it had been made. The speaker replied that the Rigorous Method could not solve all problems since there was no way to design a perfect system, however, it did give a better chance of designing one.

DISCUSSION 2

Professor Dijkstra suggested that the reason for wanting unbiased specifications was to enable minimal operations to be specified and that the design process would introduce more redundancy but was undirectional; and Professor Hoare added that the specification of an invariant would restore the unbiasedness. Professor Jones agreed but said that there may be more than one unbiased specification.

Dr. Rushby stated that the requirement for verification should be considered. Professor Jones replied that to use specifications in that way rewrite rule type semantics were needed.

Professor Dahl was unhappy that in the model approach the space had been restricted by an invariant and wondered whether this was the same invariant proposed by Hoare for the stack example. Professor Jones replied that there was some virtue in not using bias and simplified invariants.

Professor Dahl added that he saw two reasons for the invariant, firstly to exclude points which were not meaningful and secondly to exclude meaningful points which could already be reached from elsewhere.

Professor Hoare commented that when functional or logic programming was used a specification may already be a runnable program. Professor Jones replied that it was a disadvantage for a specification to be runnable as it stopped you thinking about the problem. Professor Randell added that he remembered Professor Hoare describing a specification of a multi-window display and liking it as being far from an implementation, when Peter Henderson pointed out that it was actually only hovering above an implementation.
DISCUSSION 3

Referring to the example given Professor Wells wondered whether it could be proved that the situation did not deteriorate with multiple copies of the Cleanup process running. The speaker was unsure about this but stated that in some cases garbage collection could cancel each other out.

Professor Wheeler then asked if there were two different cleanup processes would progress be guaranteed, and the speaker suggested that it might make an interesting idea for further investigation.

Professor Randell wondered if it was true that the Rely conditions complemented the Guarantee conditions instead of having a single constraint on the system as a whole. Professor Jones replied that since only two components were being considered with more components the conditions became more complex.

Professor Dahl questioned the use of decorated variables in the example and wondered if they implied the Universal Quantifier, to which Professor Jones replied that in operational terms they were quantified at many points, like a loop invariant. Professor Dahl then wondered if the speaker was only addressing the problem of safety? Professor Jones agreed, saying that Temporal statements would be needed to go further; and Professor Dahl added that the decorated variables were at least needed for liveness checks.