THE ROLE OF PROOF OBLIGATIONS IN SOFTWARE DESIGN

Cliff B. Jones
Department of Computer Science
University of Manchester
Manchester, ENGLAND

Abstract

This paper presents certain "proof obligations" which can be used to establish the correctness of software design. The design of both sequential and parallel programs is considered. The position is taken that an understanding of formal results of this kind can aid practical software development.

Introduction

Other papers in this volume discuss the use of formal semantics in language design. The, so-called, "VDM" work in this area is discussed in [Bjorner 82] and a recent application is contained in [Welsh 84]. This paper is concerned with the use of "VDM" in the design of general software. It outlines a revision of the method proposed in [Jones 80] and some extensions to cope with parallelism.

The concluding section presents a position statement on the role of such formal methods in software design.

Functional Specification

The *functional specification* of a system must define the required input/output behaviour. For simple sequential operations, such specifications can be formalised by pre- and post-conditions written as logical expressions. More complex systems comprise a collection of operations. There are two contrasting ways of recording the specifications of such a collection. In the "property oriented" approach, the meaning of the operations is fixed by writing equations in terms of several operations. This approach appears to be well suited to basic data types such as lists.

In the alternative - "model oriented" - approach, each operation is defined in

terms of an underlying state. Such a state is chosen to capture the information which is essential to a system. The state is normally defined in terms of basic data types like sets and lists. There is an intuitive notion that certain states are more abstract than others; this notion can be formalized so that it is possible to prove that there is no *bias* towards certain implementations. (The paper by Horning in this volume discusses the property and model oriented approaches.)

A simple function, which locates those indices of an Array (modelled here by a map) which are mapped to elements satisfying 'p', can be specified by writing its *signature* and pre- and post-conditions:

```
findp: map N to X → N
pre-findp(m) ≙ ∃i∈domm . p(m(i))
post-findp(m,r) ≙ r = mins({i∈domm | p(m(i))})

where:

mins: set of N → N
pre-mins(s) ≙ s ≠ {}
post-mins(s,r) ≙ r∈s ∧ ∀i∈s . r≤i

N set of natural numbers
```

Such specifications clearly show the assumptions on the arguments to functions (pre-conditions). The post-conditions show the required relationship between results and arguments. One advantage of such specifications can be seen in 'post-mins'. It is frequently clearer to write a specification by recording separate (conjoined) properties. It is also possible to write a specification which does not precisely characterize a result. For example:

```
post-findp(m,r) ≙ p(m(r))
```

would permit an implemntation to return any index - rather than the minimum - with the required property. One view is that such specifications under-determine implementations. With parallel programs, the implementations themselves can be non-deterministic.

A proposed implementation of the 'findp' specification is written as a function definition. If the implementation is correct, the following logical expression can be proved to follow from the definitions:

$\forall$ m $\in$ map $\mathbb{N}$ to X . pre-findp(m) $\Rightarrow$ post-findp(m,findp(m))

(The axiomatisation used is that of the "Logic of Partial Functions" described in [Barringer 84a].)

The preceding sequent is a *proof obligation* which establishes that the function definition *satisfies* the given specification. Clearly, it would be possible to write "specifications" which are unimplementable. There is, therefore, a proof obligation on the specification itself that it be *implementable*. In this case:

m $\in$ map $\mathbb{N}$ to X, pre-findp(m) $\vdash$ $\exists r \in \mathbb{N}$. post-findp(m,r)

The ideas on functions apply directly to operations which transform states. The experience with writing large "VDM" specifications has, however, prompted a number of abbreviations which make it easier to define the dependance on, and changes to, the state. Each operation lists those parts of the state to which the operation has *external* access; read only (rd) or read/write (wr) access is marked for each component. The names of such state components are written in upper case letters; the values are referred to in the logical expressions using lower case letters; in post-conditions, the value prior to the operation is marked with a hook while the final value is undecorated.

An operation 'FINDP' can be specified.

FINDP
ext rd M: Array
    wr R: $\mathbb{N}$
pre true
post satp(m,r) $\wedge$ $\forall i \in \{1,\ldots,r-1\}$. $\neg$p(m(i))

where:

satp(m,i) $\triangleq$ $i \leq \mathbb{N} \Rightarrow$ p(m(i))

Array = map N to X
where
inv(m) ≜ dom m = {1,...,N}

p: X → B
N (∈ N) a given constant

Notice that the pre-condition needed on the function has, here, been obviated by allowing 'FINDP' to return a number greater than 'N'.

In this small example, where only one operation is being considered, the advantage of this notation may not be clear - the larger examples in [Jones 80] benefit from such notational conventions.

Sequential Design Steps

An implementation can be proved correct with respect to a formal specification. In the design of large programs, errors can be made early in the design process. Attempts to detect errors by test cases are known to be unreliable. Even if the construction of proofs were a good way of detecting errors, it would share with running test cases the defficiency that such late detection of errors can result in the need to replace work based on erroneous design decisions.

A development method would therefore provide greater benefit if it could be used to establish the correctness of early (high-level) design decisions before proceeding to more detailed design. This observation applies equally to formal and informal design methods. Techniques for "inspections" or "structured walkthroughs" aim to increase the chance of detecting any errors in early design decisions before design goes further.

If the assumptions about the more detailed design are recorded formally, a proof can be produced that the high-level design decision is correct. The consequence of the foregoing discussion is that a formal development method must satisfy the following *requirement*:

> If a design step introduces sub-problems, its correctness can be established solely in terms of their specifications.

This requirement is relatively easy to satisfy for sequential programs; the

equivalent problem for parallel programs is less well understood.

A large specification might consist of an abstract state model and a number of operations. It is normally the case that the early steps of design involve refining, in one or more stages, the abstract state into data structures which can be easily represented in the implementation. One set of proof obligations for *data refinement* is built around the idea of providing a retrieve function (homomorphism) from the representation to the abstraction:

retr: Rep → Abs

The *adequacy* proof obligation concerns the states alone and establishes that there is at least one representation for each abstract state:

$a \in Abs \vdash \exists r \in Rep \;.\; a = retr(r)$

Two proof obligations must be discharged for each operation. If the operations on the abstract state are 'OPAi' and those on the representation 'OPRi', the *domain* proof obligation is:

$r \in Rep,\ pre\text{-}OPAi(retr(r)) \vdash pre\text{-}OPRi(r)$

The *result* proof obligation is:

$\overleftarrow{r}, r \in Rep,\ pre\text{-}OPi(retr(\overleftarrow{r})),\ post\text{-}OPRi(\overleftarrow{r}, r) \vdash post\text{-}OPAi(retr(\overleftarrow{r}), retr(r))$

The requirement on development methods is satisfied since subsequent steps of design rely only on the representation and its operations.

Large examples of such data refinements are published elsewhere (e.g. [Fielding 80], [Jones 83b], [Bjorner 82], [Welsh 82]).

The process of data refinement brings the specification closer to an implementation, but post-conditions still define what has to be done rather than how to do it. *Operation decomposition* splits such implicitly specified operations into small steps. These smaller (sub-)operations may either be represented by specifications or be available operations of the implementation (hardware, language or other supporting software).

Proof obligations for operation decomposition must exist for each construct in a programming language. The original proof rules of [Hoare 69] concerned

post-conditions of the final state alone; the first attempt in [Jones 80] to handle post-conditions of two states was unnecessarily clumsy; the more tractable proof obligations given in [Jones 83b] were suggested by Peter Aczel (cf.[Aczel 82]). These rules again satisfy the requirement on a development method: sub-components can be developed solely from their specifications and can ignore the context in which they are used.

A very simple example is to show that the sequential composition of 'INIT' and 'SEARCHES' satisfies the earlier specification of 'FINDP':

```
INIT
ext wr R: ℕ
pre true
post r = N + 1

SEARCHES
ext rd M: Array
    wr R: ℕ
pre satp(m,r)
post consid(m,r,Ind) ∧ satp(m,r)

where:

Ind = {1,...,N}
consid(m,i,s) ≙ ∀j∈s . p(m(j)) ⇒ i≤j
```

The sequential composition rule in this case does little more than check that the post-condition of 'INIT' establishes the pre-condition for 'SEARCHES'. A slightly more interesting example would be the decomposition of 'SEARCHES' into a loop. Examples of the use of these rules are given in [Jones 83a], [Jones 83b].

Parallel Decomposition Steps

The decomposition of operations into sub-operations which can execute in parallel must now be considered. The difficulty is to meet the development method requirement. Here, shared variable paralleism is considered. The first observation is that the pre-/post-condition form of specification is not rich enough. It is easy to construct examples where the behaviour of two operations running in parallel is not goverened by their separate post-conditions. The

interference which the operations exert on each other influences the final result.

The research results reported in [Francez 78], [Lamport 80] and [Jones 81] each attempt to solve this problem by extending the notion of specification to capture some aspects of the interference. The last of these methods is described here. The approach proposed is to face the issue of interference throughout development; to reflect its existence in the specification; and to recognise that it must be checked at each design step.

Specifications of interfering operations are extended with assertions (rely-conditions), which express the assumptions that can be made about the interference which can be tolerated, and assertions (guarantee-conditions), which constrain the interference which may be caused. More precisley, a *rely-condition* is a predicate of two states which defines the relationship which can be assumed to exist between the external variables in states changed by other processes. Thus the implementor of an operation, although not able to assume that the implementation will run in isolation, knows some limit to the state changes which other processes can make. For example the rely-condition:

$$x = \overleftarrow{x}$$

expresses the assumption that the (value of the) external variable 'X' will not change; but:

$$t \leqslant \overleftarrow{t}$$

accepts the possibility of change but requires that 'T' never increases in value. The rely-condition:

$$x+y = \overleftarrow{x}+\overleftarrow{y}$$

requires that the sum of two values remains unchanged. (This example and the next imply some notion of indivisible operations in the implementation.) The role of a switch to control changes can be given in a rely-condition:

$$\overleftarrow{sw}=FULL \Rightarrow buf=\overleftarrow{buf}$$

Rely-conditions, like pre-conditions, are recording assumptions for the implementation; the commitments (cf. post-conditions) relating to interference are

recorded in *guarantee-conditions*. These are again predicates of two states which all state transformations must respect. The expessions above could occur in guarantee-conditions. A process which was to *coexist* with one whose rely-condition was as in the last equation, might have only read access to 'SW' and use:

$$\overleftarrow{sw}=EMPTY \Rightarrow buf=\overleftarrow{buf}$$

as a guarantee-condition.

It is possible to think of a rely-condition, for say 'OP', as a post-condition of an operation which may be executed between any two atomic steps of 'OP'. The guarantee-condition can be thought of as the post-condition for the atomic steps of 'OP'. (Although this explanation refers to "atomic steps", the level of atomicity is not fixed.)

The *Sieve of Erathosthenes* is used as a first illustration of the extended specifications. The idea of using two parallel processes was proposed in [Hoare 75]. The solution is extended here to use more processes. The overall task can be achieved by storing the set of all integers between 2 and N in a variable and then invoking SIEVE:

```
S := {2,...,N}; SIEVE

SIEVE
ext wr S: set of ℕ
rely s = s←
post s = s - ∪{mults(i) | i∈{2,...,sqrt(N)}}

mults(i) ≙ {i*m | m∈{2,...,N}}
```

Notice that interference on 'S' is excluded in the specification. Suppose 'SIEVE' is to be implemented by executing in parallel many instances of a process 'REM' - one instance for each 'I'. What should the specification of the 'REM' process be? The beginning is straightforward:

```
REM(I: ℕ)
ext wr S: set of ℕ
```

If the post-condition were:

$$s = \overleftarrow{s} - mults(i)$$

the overall post-condition would follow from the conjunction of those for each 'REM(I)'. This would be acceptable if each instance of 'REM' were run in isolation. The equality sets an upper and lower bound on changes to 'S'. This is too restrictive in the case that other parallel interfering processes are changing 'S'. ('REM(2)' might, for example, run at a time when 'REM(3)' removes the value '9' from 'S'.) The lower bound on the effect of 'REM' can be defined in the post-condition:

$$\forall j \in mults(i) \;.\; j \notin s$$

However, the conjunction of such post-conditions will not yield the overall post-condition. The proof rule for realization by parallel processes is given below (in a form suggested by Peter Aczel in [Aczel 83]). The overall post-condition can be a consequence of information about interference. In this case it is clear that for any state which can arise a value which has been removed must be a multiple of one of the process indices. Formally:

$$\forall c \in (\overleftarrow{s} - s) \;.\; \exists i \in \{2,\ldots,N\} \;.\; c \in mults(i)$$

Since the states which can arise are all created by the steps of the instances of 'REM', this must follow from the transitive closure of the guarantee-condition for 'REM'. Thus the guarantee-condition must include:

$$\forall c \in (\overleftarrow{s} - s) \;.\; c \in mults(i)$$

Another way to see the need for this guarantee-condition is to observe that the post-condition would be satisfied by a process which set 'S' to the empty set!

Furthermore, the 'REM' process can only ensure that elements will not be in the final value of 'S' if no other (interfering) process can reinsert values (e.g. 'REM(2)' will remove the value '6' once - if 'REM(3)' were to reinsert the value, the post-condition would not be satisfied). Thus a rely-condition of:

$$s \subseteq \overleftarrow{s}$$

is required. Since multiple instances of 'REM' will run in parallel, this must also be conjoined to the guarantee-condition. Thus the overall specification for 'REM' becomes:

```
REM(I: ℕ)
ext wr S: set of ℕ
pre true
rely s ⊆ s←
guar s⊆s← ∧ ∀c∈(s←-s) . c∉mults(i)
post ∀j∈mults(i) . j∉s
```

Returning to the 'FINDP' problem, the overall specification might have rely-and guarantee-conditions:

```
rely m=m← ∧ r=r←
guar true
```

The operations 'INIT' and 'SEARCHES' require similar, trivial, extensions. The extended form of the sequential proof rules is given in [Barringer 84b].

Suppose that 'SEARCHES' is now to be implemented by the parallel execution of 'T' processes 'SEARCHi' (for i∈{1,...,T}) where each such process is responsible for checking a set of indices given in a map:

```
GS: map {1,...,T} to (set of Ind)
```

The each 'SEARCHi' process is specified:

```
SEARCHi
ext rd M: Array
    wr R: ℕ
pre true
rely m | gs(i) = m← | gs(i) ∧ r ≤ r←
guar r≠r← ⇒ r<r← ∧ satp(m,r)
post consid(m,r,gs(i))
```

where the map restriction operator yields that portion of a map whose domain is in the set:

```
m | s = [d ↦ m(s) | d∈(dom m ∩ s)]
```

A specification now consists of four predicates:

P pre-condition of one state

R rely-condition of two states

G guarantee-condition of two states

Q post-condition of two states

The proof rule for the decomposition to two parallel processes is:

$$\frac{S1 \text{ sat } (P, R \vee G2, G1, Q1),\quad S2 \text{ sat } (P, R \vee G1, G2, Q2)}{(S1 \parallel S2) \text{ sat } (P, R, G1 \vee G2, Q1 \wedge Q2 \wedge (R \vee G1 \vee G2)^*)}$$

The generalisation to 'n' processes is:

$$\frac{\&_i(Si \text{ sat } (P, R \vee \bigvee_{j \neq i} Gj, Gi, Qi))}{(\parallel Si) \text{ sat } (P, R, \bigvee_i Gi, \&_i Qi \wedge (R \vee \bigvee_i Gi)^*)}$$

where $\&_i/\bigvee_i$ are generalised (finite) conjuctions and disjunctions.

To show that the parallel decomposition of 'SEARCHES' is correct, a number of proof obligations must be discharged.

$$\&_i(\text{pre-SEARCHES} \Rightarrow \text{pre-SEARCHi})$$

is vacuously true;

$$\&_i(\text{rely-SEARCHES} \vee \text{Vj guar-SEARCHj} \Rightarrow \text{rely-SEARCHi})$$

is straightforward;

$$\bigvee_i(\text{guar-SEARCHi}) \Rightarrow \text{guar-SEARCHES}$$

is vacuously true;

$$\text{pre-SEARCHES} \wedge (\&_i \text{ post-SEARCHi}) \wedge (\text{rely-SEARCHES} \vee \bigvee_i \text{guar-SEARCHi})^* \Rightarrow \text{post-SEARCHES}$$

requires:

$$satp(\overleftarrow{m},\overleftarrow{r}) \wedge \&_i\ consid(m,r,gs(i)) \wedge m=\overleftarrow{m} \wedge r=\overleftarrow{r} \Rightarrow satp(m,r) \Rightarrow consid(m,r,Ind) \wedge satp(m,r)$$

which follows provided that the distributed union of the range of 'GS' is equal to 'Ind'.

In [Owicki 76] a parallel implementation of this problem is given using one process each for the odd and even indices. In [Jones 83c] it is shown how this can be developed from the 'SEARCHi' specification. (Basically by a specialization to two processes and a data refinement of 'R' onto an expression.) The key requirement of a development method is again met in that it is not necessary to reconsider the earlier stages of design. (The same paper discusses a maximal parallel solution with one process per index.)

The method outlined above illustrates that it might be possible to find a way of developing interfering programs which meets the requirement. The expressiveness of the rely-/guarantee-conditions is, however, inadequate for many problems. Recent research (e.g. [Lamport 83], [Barringer 84], [Barringer 85] - for overview see [de Roever 85]) has moved to using Temporal Logic. This author has some hesitation in following this step! It must be realized that even the sequential rules discuss the temporal changes to states. The contribution of the Naur/Floyd/Hoare techniques is that the proof obligations themselves hide this fact. The rely/guarantee idea was an attempt to regain this situation in spite of the more complex environment. The next step should be work on a number of examples using such temporal rules (e.g. [Sa 84]); if patterns of specification and proof can be isolated, perhaps we can again confine temporal arguments to the justification of proof rules which then do not use temporal logic. (A general overview of approaches to parallelism is given in [Barringer 84b].)

Position

These advanced seminars are considering the relevance of formal methods in software development. This section contains a number of *claims* which indicate this author's position.

1. Formal specifications tend to focus on the functional apsects of systems - questions like the need for a system are not normally considered. (The separation

of performance issues should not cause surprise.)

2. Formal specifications have been written for (the functional apsects) of significant software, and hardware, systems.

3. There is a need to develop techniques to aid checking that the specifications of large systems meet the user's intentions.

4. A specific area where formal specification is, at the moment, less helpful (than simulation) is so-called "Man-Machine Interfaces". (Some work in this are is reported in this conference - [Marshall 85].)

5. A formal specification can provide a precise and (relatively) concise model of a system prior to construction. Such a specification can both deepen understanding of the intended system and provide a correctness criteria for implementation (and design).

6. The skills required to write such specifications are not yet widely available - education is likely to be the limiting factor in the use of formal methods.

7. Support tools (cf. [Madhavji 85], [Snelting 85] in this conference) are required - there is a danger that such tools could force "mathematics as a specification language" to develop into Ada-like syntactic quagmires.

8. Formal development methods focus on correctness issues and leave somewhat aside the intuitive aspects of how to choose a good design (cf. [Naur 85]).

9. Application of formal methods to the early steps of design can reduce the possibility of undetected errors - it is these errors which damage the "productivity" of the program design process.

10. The decision as to the appropriate degree of formality to be used in design verification is difficult. It is, as yet, unrealistic to use completely formal (machine checked) proofs for large systems. It would appear that the earlier stages warrant more formal treatment than the later ones. A crucial step towards greater machine support would be the development of "theories" about commonly used data structures.

11. Formal methods will certainly not solve all problems concerned with

software development. It is, however, a valid research area.

12. Experiments are needed to improve the ease of access and style of formal documents. Perhaps some of the research effort which appears to develop mathematics for its own sake could benefit from more application to actual computing problems.

13. Formal methods, even in their evolving state, have a contribution to make to current problems (see reports from industry at this conference).

References

[Aczel 82] **A Note on Program Verification**, P.Aczel, *manuscript*, January 1982.

[Aczel 83] **On an Inference Rule for Parallel Composition**, P.Aczel, *manuscript*, February 1983.

[Barringer 84] **Now You May Compose Temporal Logic Specifications**, H.Barringer, R.Kuiper and A. Pnueli, *Procs. of 16th ACM Symposium on Theory of Computing*, May 1984.

[Barringer 84a] **A Logic Covering Undefinedness in Program Proofs**, H. Barringer, J.H. Cheng and C.B. Jones, *ACTA Informatica*, Vol 21 Part 3, pp251-269, 1984.

[Barringer 84b] **A Survey of Verification Techniques for Parallel Programs**, H.Barringer, *to be published, LNCS, Springer-Verlag.*

[Barringer 85] **A Compositional Temporal Approach to a CSP-Like Language**, H.Barringer, R.Kuiper and A.Pnueli, *IFIP Working Conference on "The Role of Abstract Model in Information Processing"*, Vienna, January 30th - February 1st, 1985.

[Bjorner 82] **Formal Specification and Software Development**, D.Bjorner and C.B.Jones, *Prentice-Hall International*, 1982.

[Fielding 80] **The Specification of Abstract Mappings and their Implementation as B^+-Trees**, E. Fielding, *Oxford University, Monograph PRG-18*, 1980.

[Francez 78] **A Proof Method for Cyclic Programs**, N.Francez and A.Pnueli, *ACTA Inf.* Vol 9 No 2, pp133-157, April 1978.

[Hoare 69] **An Axiomatic Basis of Computer Programming**, C.A.R.Hoare, *CACM*

[Hoare 75] **Parallel Programming: An Axiomatic Approach**, C.A.R.Hoare, *In Computer Langs, Permagon Press*, Vol 1, pp 151-160.

[Jones 80] **Software Development: A Rigorous Approach**, C.B. Jones, *Prentice-Hall International*, 400 pages, 1980.

[Jones 81] **Development Methods for Computer Program Including a Notion of Interference**, C.B.Jones, *Oxford University, Monograph PRG 25*, June 1981.

[Jones 83a] **Specification and Design of (Parallel) Programs**, C.B.Jones, *(invited*

paper), IFIP 1983, Paris, North-Holland, pp 321 – 332, September 1983.

[Jones 83b] **Systematic Program Development**, C.B.Jones, *Symposium 'Wiskunde en Informatica', Amsterdam, to be published in the Mathematical Centre Tracts.*

[Jones 83c] **Tentative Steps Toward a Development Method for Interfering Programs**, C.B.Jones *ACM Trans. Program. Lang. Syst.*, Vol 5 No4, pp 596 – 619, October 1983.

[Lamport 80] **The "Hoare Logic" of Concurrent Programs**, L.Lamport, *Acta Inf.*, vol 14 no 1, pp21–37, June 1980.

[Lamport 83] **What Good Is Temporal Logic?**, L. Lamport, *North-Holland, Proc. of the IFIP 9th World Computer Congress, Paris*, pages 657–668, 1983.

[Marshall 85] **A Formal Specification of Line Representations on Graphics Devices**, L.S.Marshall, *TAPSOFT Joint Conference on Theory and Practice of Software Development*, Berlin, March 1985.

[Madhavji 85] **Software Construction Using Typed Fragments**, N.H.Madhavji, N.Leoutsarakos, D Voulliouris, *TAPSOFT Joint Conference on Theory and Practice of Software Development*, Berlin, March 1985.

[Naur 85] **Intuition in Software Development**, P. Naur, *TAPSOFT Joint Conference on Theory and Practice of Software Development*, Berlin, March 1985.

[Owicki 76] **Verifying Properties of Parallel Programs: An Axiomatic Approach**, S.S.Owicki and D.Gries, *Comm. ACM*, Vol 19 No 5, pp 279–285.

[de Roever 85] **The Quest for Compositionality – a Survey of Assertion-based Proof Systems for Concurrent Programs**, W.P.de Roever, *IFIP Working Conference on "The Role of Abstract Models in Information Processing"*, Vienna, January 30th – February 1st., 1985.

[Sa 84] **Temporal Logic Specifications of Communication Protocals**, J.Sa, *Manchester University*, 1984.

[Snelting 85] **Experiences with the PSG–Programming System Generator**, G.Snelting, *TAPSOFT Joint Conference on Theory and Practice of Software Development*, Berlin, March 1985.

[Welsh 82] **The Specification, Design and Implementation of NDB**, A. Welsh, *M.Sc. Thesis, Manchester University*, October 1982.

[Welsh 84] **A Database Programming Language: Definition, Implementation and Correctness Proofs**, A. Welsh, *Ph.D. thesis, Manchester University*, October 1984.