DEVELOPMENT METHODS FOR COMPUTER PROGRAMS INCLUDING A NOTION OF INTERFERENCE

by

C.B. JONES

Oxford University Computing Laboratory

Programming Research Group

DEVELOPMENT METHODS FOR COMPUTER PROGRAMS INCLUDING A NOTION OF INTERFERENCE

by

C.B. JONES

Technical Monograph PRG-25
June 1981

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
OXFORD. OX2 6PE

Abstract

A relational basis is used for the systematic development of programs. Abstract data types are defined by models and a proof method of data refinement is given. Proof rules for various sequential control constructs (e.g. if, while) are given and proved valid with respect to a denotational semantics for the non-deterministic language. Monotonicity in the semantics is linked to the needs of a stepwise development procedure. Interference from parallel tasks which can change shared variables is also covered in the development process. The appropriate proof rule is related to an operational semantics.

To Professor A. van Wijngaarden

on his retirement

CONTENTS

		Page
Acknowledgements		
Introduction		0-1
	0.1 Development Methods	0-4
	0.2 Rigorous Methods	0-7
	0.3 Sequential Programs	0-9
	0.4 Interference	0-10
	0.5 Organisation	0-11
Chapter 1.	Specification	1-1
	1.1 Relations and Operations	1-4
	1.2 Realisation of Specification	1-11
	1.3 Data Types	1-13
	1.4 Specifications via Predicates	1-28
	1.5 Specifying Order	1-32
	1.6 Alternatives	1-34
Chapter 2.	Data Refinement	2-1
	2.1 Objects and their Properties	2-5
	2.2 Data Refinement Proofs	2-19
	2.3 General Refinement	2-30
	2.4 Alternatives	2-32
Chapter 3.	Decomposition for Isolated Programs	3-1
	3.1 Sequential Programming Language	3-4
	3.2 Proof Rules	3-14
	3.3 Justification of Proof Rules	3-29
	3.4 Examples	3-35
	3.5 Alternatives	3-47

		Page
Chapter 4.	Development of Interfering Programs	4-1
	4.1 Extensions to Specification Format ..	4-4
	4.2 Realisation	4-15
	4.3 Operational Semantics for Parallelism ..	4-16
	4.4 Proof Rules	4-21
	4.5 Justification of Proof Rule	4-31
Chapter 5.	Examples	5-1
	5.1 Sorting and Searching	5-3
	5.2 Locating an Array Element	5-9
	5.3 Recording Equivalence Relations	5-26
Chapter 6.	Alternatives	6-1
	6.1 Shared Variable Parallelism	6-3
	6.2 Communication Based Parallelism	6-12
Chapter 7.	Conclusions	7-1
	7.1 Achievements	7-3
	7.2 Limitations	7-4
	7.3 Further Work	7-5
	7.4 Bringing into Practice	7-6
References	.. ..	REF-1

Introduction

The development of computer systems faces three major problems:

i) so-called "tested" systems invariably have residual design errors

ii) the cost of producing such systems is enormously high, largely because of the rework necessitated by the late detection of errors made early in the development process

iii) even when delivered and functioning "according to specificaiont" the architecture of computer systems is such that they are unusable other than by "professionals".

It is argued in Jones /80b/ that the root cause of these problems is the way in which the complexity of tasks being tackled has outgrown the methods being used in their design.

The aim of the current dissertation is to make a contribution to systematic or rigorous methods of program development. The so-called "Vienna Development Method" (VDM) has been applied to the construction of many specifications of diverse systems (see Bjørner /82a/). (The notation used has become known as "Meta-IV".) The systematic development of programs from such specifications is described in Jones /80a/.

The earlier work has largely ignored the problems of parallelism and the main incentive to pursue the research reported here was to tackle this shortcoming. The outcome is the identification and definition of a notion of "interference".

The importance of parallelism comes both from the desire to perform some algorithms at very high speed and from the economic issues giving rise to the growth of "distributed computing".

0.1 Development Methods

Michael Jackson has pointed out that the virtue of a development method (as also with a programming language) lies in what it inhibits. The interest here is in development methods which make it very unlikely that design errors are undetected. That is, a development method should provide criteria which are applied at each stage of development; the consequent error detection should be such that the next stage can be undertaken with confidence.

A specification must define WHAT a system should do without making commitments as to how the objectives are to be achieved. Just one of the reasons for this emphasis is that a specification must be reviewed by the would-be users of a system. This rôle necessitates that a specification be based on an input/output relation: this is the user's only concern. Where it is necessary to discuss an internal state, this must be done in terms of abstract data types in order to avoid over-specification.

Since the specification is to be the basis of the design, it must be written in a language which is formal enough to support an argument of correctness. For large projects the design is likely to be made in several steps and it is an essential feature of a useful development method that correctness arguments are possible in the early stages of design. The importance of this criterion is that late detection of errors made in the early stages of design decreases the productivity of the development process dramatically. Any aspect of correctness which is postponed makes all subsequent work based on the decision a hostage to fortune.

(In addition to the widely accepted observation that the cost of correcting errors increases logarithmically over project stages, it has recently been noticed that errors tend to have been made in stages which are in inverse order to the stages where they are detected.)

The sense in which the term "development method" is used here is a basis of understanding and notation for recording a design. It is not envisaged (as in Jackson /75a/) that a development method must be normative on selected designs. It is, of course, true that a systematic design notation creates a framework in which it is easier to review design possibilities.

A systematic development method should, then, contribute to the solution of all three major problems listed at the beginning of this introduction. To understand that the contribution to the architecture of systems is more than wishful thinking, one need only consider the rôle of abstract, manipulatable, models in other disciplines (cf. Zemanek /80a/).

In order for a design document to be comprehensible, it must have an explicit structure. The most widely exalted, but by no means the only possible, structure is top-down. It is frequently convenient when writing to use a form of words which implies that the top-down structure of the design documentation results from taking decisions in a top-down order. Such a design rule would be hopelessly unrealistic as is shown when a true record of a design history (cf. Naur /72a/) is studied. Whatever is said below, the intent is to achieve

structured design descriptions but not to impose some specific order of thinking.

0.2 Rigorous Methods

Logic is the study of valid deductions. One goal is to reduce the notion of proof to precise rules of symbol manipulation. Most mathematics texts are not written at this level of formality. Essentially a proof of a theorem is an outline. The steps of a proof do three things:

i) they reduce the comprehension problem

ii) they make theorems far less vulnerable to counter-example than unsupported conjectures

iii) they make it clear how a greater level of formality could be provided.

The aim of a rigorous method for developing programs is to achieve the level of rigour of mathematical texts. In particular, a formal basis for methods is sought so that it is clear where and how greater formality can be applied in cases of doubt. This is true both for specifications themselves and for justifications based thereon.

(In several places below, verbal arguments are provided as an introduction to a more formal proof. If reading these does not convince the reader of the advantages of using formulae, writing them certainly would have done so.)

The motivation for the work reported here is to provide methods which are usable by practitioners of software development. As such, a careful evaluation must be made of any notation or concepts which

will burden the software engineer. It is, of course, permissible to use rather more difficult notions in justifying methods than are actually required in their employment. But each problem put in the way of general understanding of a development method must have a clear pay off.

One rather unsatisfactory aspect of the state of software engineering is the paucity of texts which record the accumulated knowledge of the subject in an accessible style. It is one of the objectives of a systematic method to provide a notation and style in which familiar programming concepts can be defined and their alternative solutions documented.

0.3 Sequential Programs

It is useful to review the history of attempts to bring sequential programs under intellectual control. The first proofs about programs (schema) were proofs of equivalence; the next stage provided proofs of given, or extant, programs against specifications; the third stage applied the earlier ideas to the systematic development of programs. The study of data types and data refinement came later than the concern with flow of control.

The work in Jones /80a/ is concerned with the systematic development of programs. It differs from the standard literature in a number of ways. Section 3.5 below discusses the use of post-conditions of state pairs. (The current extensions to tackle the problems of parallelism add support to this choice.) Based on early work on data refinement (Jones /72a/) a highly practical approach to this problem is also presented.

In the current dissertation some of the techniques necessary to cope with parallelism have been employed on sequential programs (e.g. identification of global variables). A firmer basis has also been provided for the treatment of non-deterministic programs.

0.4 Interference

A sequential program, or its proof, can rely on the fact that a variable will have the value last assigned to it. Parallelism brings with it a notion of interference which invalidates this assumption. (Communication based parallelism is discussed in section 6.2.)

The history of the work on sequential programs is being repeated with parallelism. It is the claim of this dissertation that little published work on parallel programs satisfies the criteria of development methods given above.

The basic proposal here is to face the notion of interference and to provide a place for it in both specifications and justification of development steps. Thus it is claimed that a true development method is provided by:

i) recognising the basic rôle of interference

ii) recording interference considerations in rely- and guarantee-conditions

iii) providing proof rules for parallel programming constructs.

In some senses, this dissertation is part of a larger, on-going, project to refine and disseminate more systematic program development methods. Some of the limitations of the current state are reviewed in chapter 7.

0.5 Organisation

Chapter 1 deals with basic notation and the concept of specification; chapter 2 provides proof rules for data refinement; chapter 3 performs the same service for sequential programming constructs; chapters 4 to 6 are concerned with parallelism. Some of the examples are carried right through the text (occasionally using notation prior to its definition). Many more examples would have to be given to establish the scope of the methods: for sequential programs these could be reworked from Jones /80a/.

Sections 1.6, 2.4, 3.5 and chapter 6 contain comparisons with other methods for tackling the relevant problems. No attempt is made to provide a full exposition where published work is readily available.

The following numbering scheme is used:

n-m	page numbers
0,1, ..., 7	chapters, with sections and sub-sections given appropriate Dewey-decimal numbers (e.g. 3.1.1)
Arabic numerals	formulae and cross-references thereto within a section
n.m(p)	cross reference to a formula in another section
Roman numerals	proof steps

Where the code of a program is given, the syntax of the Ada language is used. This is in no way an essential part of the work presented here and all of the programs could equally well be written in, say, Pascal Plus (Welsh /80a/).

Chapter 1

Specification

A specification must constrain WHAT is to be done rather than HOW the specified behaviour is to be achieved. The external behaviour of a system is normally easier to define than the internal realisation details; the definition of the behaviour should also be more stable than the internal mechanism - this point is of particular importance where a number of alternative realisations of the same specification is to be produced. It is also essential to remember that correctness can only be discussed in relation to a specification.

One aspect of the freedom from implementation details of a specification which is exploited below is the use of designs which introduce parallelism. In chapters 2 and 3, however, specifications which permit a range of results can be interpreted as defining a range of valid deterministic implementations. The simple example of a square root function can be used to illustrate this point. A specification of square root would presumably allow some tolerance for the result: this can be taken to define a set of valid deterministic functions. The consideration of under-determined results has an influence on much of this work. Furthermore, the fact that partial functions and operations are to be specified also permeates all that follows. The technical definition of what is meant by satisfying such a specification is given in section 1.2.

The natural model of specifications which cater for partial, under-determined objects is relations: the necessary notation is presented in section 1.1. In spite of the enthusiasm expressed by some authors (e.g. Burge /75a/, Backus /78a/, Henderson /80a/), most

programming is done using languages in which a state is a thin disguise for the von Neumann architecture of the computer on which the programs must run. The term "operation" is used to distinguish, from functions, those things which change or use a state.

The concept of specifying by means of an input/output relation can be readily extended to cover operations. With larger problems, however, the description of the state itself becomes of interest. The view proposed in section 1.3 is that the state is an abstract data type which is characterised by the behaviour of the operations. Much recent work on abstract data types has concentrated on "property oriented" specifications (cf. section 1.6 for references); here the specifications are given via a "model".

Although the foundations of the current work are expressed in terms of relations, it is more convenient to present and reason about particular specifications using logical expressions: the connection between these styles is discussed in section 1.4. Section 1.5 links the work presented here to those parts of VDM which are concerned with combinators.

1.1 Relations and Operations

This section fixes both the basic notation required and the concept of an operation. The logic and set notation used is introduced here informally. For a more careful treatment see Abrial /82a/.

1.1.1 Notation (Logic, Sets and Relations)

The truth values are written:

1 TRUE, FALSE

The conventional propositional operators are denoted by (in decreasing order of priority):

2 $\sim$ $\wedge$ $\vee$ $\Rightarrow$ $\Leftrightarrow$

These operators are defined (only) for the given truth values. One way of reducing the problems with UNDEFINED is to use bounded quantifiers:

3 $(\forall i \in I$ st $p(i);\ q(i))$

4 $(\exists i \in I$ st $p(i);\ q(i))$

The semicolon with the universal quantifier can be pronounced "it follows" and with the existential quantifier as "for which".

Another form of logical expression which is useful in avoiding UNDEFINED values is the conditional expression. Thus the "&" of Jones /72a/ (cf. cand in Dijkstra /76a/) is defined:

5 (p & q) is the same as (if p then q else FALSE)

In spite of these very strict methods of dealing with UNDEFINED, a more relaxed usage is permitted when there is little danger of confusion.

Standard set operators are used:

6 $\cup$ $\cap$ $-$ $\subset$ $\subseteq$ $\in$ $\notin$ $\times$ $\wp$

The operator yielding the cardinality of a finite set is written:

7 card S

Distributed union and intersection (the latter only for non-empty sets of sets) are written:

8 union SS, int SS

Explicitly given sets (the empty set) are written:

9 $\{e1, e2, \ldots, en\}$ ($\{\}$)

Implicitly defined sets are denoted by:

10 $\{i \in I \text{ st } p(i)\}$

The following basic sets are used:

11 Bool = $\{$TRUE, FALSE$\}$

12 Nat = $\{1, 2, \ldots\}$

13 Nat0 = $\{0, 1, \ldots\}$

14 Int = $\{\ldots, -1, 0, +1, \ldots\}$

Relations are viewed as sets of pairs:

15 $v \in V \wedge w \in W \Rightarrow \text{pair}(v,w) \in V \times W$

The explicit constructor (pair) is omitted where there is no danger of confusion.

Projection functions exist:

16 first: $V \times W \rightarrow V$, second: $V \times W \rightarrow W$

The "pair" function is normally avoided by using:

17 $R{:}v \leftrightarrow w$ for $pair(v,w) \in R$

18 $v \leftrightarrow w$ for $pair(v,w)$

In the remainder of this sub-section relations are assumed to be over V (i.e. $R \subseteq V \times V$) and S is assumed to be a subset of V. Then:

19 $\Omega = \{\}$

20 $\mho = V \times V$

21 $E_S = \{s \leftrightarrow s \ \underline{st} \ s \in S\}$

22 $R^{-1} = \{w \leftrightarrow v \ \underline{st} \ R{:}v \leftrightarrow w\}$

23 $R1;\ R2 = \{u \leftrightarrow w \ \underline{st} \ (\exists v \in V;\ R1{:}u \leftrightarrow v \ \wedge \ R2{:}v \leftrightarrow w)\}$

Notice that, although the same symbol is used for relational composition as the programming language connective, their semantics do <u>not</u> match. In fact, the resolution of this problem with non-deterministic programs costs some time in chapter 3.

When a set appears in a context where a relation is required, the appropriate identity or diagonal relation is to be used. Thus:

24 $S;\ R = E_S;\ R$

Define:

25 $\underline{dom}\ R = \{first(pr) \ \underline{st} \ pr \in R\}$

26 $\underline{rng}\ R = \{second(pr) \ \underline{st} \ pr \in R\}$

A relation, R, is a partial order if:

27 $E_V \subseteq R, \quad R \cap R^{-1} = E_V, \quad R; \quad R \subseteq R$

A strict (irreflexive) partial order R is a relation such that:

28 $R \cap R^{-1} = \sqcap \ , R; \quad R \subseteq R$

from which it follows:

29 $E_V \cap R = \sqcap$

A relation R is a partial function provided:

30 $R^{-1}; \quad R \subseteq E_V$

In such cases, the type of R is written:

31 $R: V \nrightarrow V$

and the pairing assertion as:

32 $R: v \mapsto w$

A relation (function) is total when:

33 $\underline{\text{dom}}\ R = V$

from which it follows:

34 $E_V \subseteq R; \quad R^{-1}$

The type of a total function is written:

35 $R: V \rightarrow V$

A relation (function) is a surjection ("onto") provided:

36 $\underline{\text{rng}}\ R = V$

from which it follows:

37 $E_V \subseteq R^{-1}; R$

The type of a total relation which is "onto" is written:

38 $R: V \leftrightarrow V$

A one-one function (bijection) is shown by:

39 $R: V \rightleftarrows V$

A relation, R, is an equivalence relation providing:

40 $E_V \subseteq R, \quad R; \quad R \subseteq R^{-1}$

from which the more conventional properties follow:

41 $R = R^{-1}, \quad R; \quad R \subseteq R$

Well-founded relations play an important part in program termination arguments. Given some (partial) relation $>$, the conventional definition of well-foundedness requires that there are no infinite descending chains:

42 $\sim(\exists f \in Nat \rightarrow V; \quad (\forall i \in Nat; \quad f(i) > f(i+1)))$

It is sometimes easier to see that a set is well-founded by stating that for any non-empty subset of V, there must be some stopping value:

43 $(\forall S \subseteq V \ \underline{st} \ S \neq \{\}; \quad (\exists s1 \in S; \quad \sim(\exists s2 \in S; \quad >: s1 \mapsto s2)))$

It is an immediate consequence of 42 that " $>$ " must be irreflexive:

44 $E_V \cap > = \Omega$

and furthermore:

45 $> \cap >^{-1} = \Omega$

(Although a well-founded relation is not necessarily transitive, its non-reflexive transitive closure is well-founded.)

Section 2.1 shows some interesting examples of well-founded relations.

1.1.2 Operations

An operation is viewed here as a relation on states. Various ways of defining sets of objects, including the use of data type invariants, are introduced in section 2. In this sub-section, a set of states (St) is assumed to be known. Thus, operations are:

1 $OP \subseteq St \times St$

The specification of an operation is also viewed as a relation:

2 $SP \subseteq St \times St$

This clearly corresponds to the proposed input/output view of a specification. The intuitive meaning of such a specification is that any acceptable realisation must be defined over the entire domain of the relation (i.e. a realisation must terminate and deliver an answer over the prescribed set) and that, when restricted to the domain of the specification, all of the answers created by a realisation must be consistent with the specification. Consider the following example:

3 $MULTP = \{\langle x,y,r\rangle \leftrightarrow \langle x',y',r'\rangle \ \underline{st}\ y \geqslant 0 \wedge r' = x * y\}$

Here, the state is shown as a list of three values. The clause "$y \geqslant 0$" can be thought of as a pre-condition, and a satisfactory

implementation might remove this restriction, thus a realisation which satisfies this specification is:

4 $\{\langle x,y,r\rangle \leftrightarrow \langle x',y',r'\rangle \ \underline{st}\ r' = x * y\}$

The specification in 3 does not determine the values of all of the variables after execution, thus another satisfactory implementation would be:

5 $\{\langle x,y,r\rangle \leftrightarrow \langle x',y',r'\rangle \ \underline{st}\ (\underline{if}\ y \geqslant 0\ \underline{then}\ r' = x * y \wedge x' = x \wedge y' = y$
$\underline{else}\ \ r' = x - y \wedge x' = y \wedge y' = x)\}$

The examples above show that the notion of satisfaction is a partial order on specifications. The aim of chapter 3 is to provide the rules by which realisations can be shown to satisfy specifications. One satisfactory decomposition of 3 is:

6 R:=0; $\underline{while}$ Y ≠ 0 $\underline{loop}$ MULTB $\underline{endloop}$;

7 MULTB = $\{\langle x,y,r\rangle \leftrightarrow \langle x',y',r'\rangle \ \underline{st}\ r + x * y = r' + x' * y' \wedge 0 \leqslant y' < y\}$

Notice how MULTB, which can be thought of as a specification, is written in the program: this usage is justified in chapter 3. Realisations of 7 which give rise to linear and logarithmic performance are:

8 R:=R + X; Y:=Y - 1;

9 $\underline{if}$ EVEN(Y) $\underline{then}$ X:=X*2; Y:=Y÷2; $\underline{endif}$;
R:=R + X; Y:=Y - 1;

These examples should have indicated how the use of relations as specifications leaves useful freedom to the realisation.

In Jones /80a/, operations were extended to allow a type clause. Such "general operations" can be used to specify procedures or functions but the extensions are straightforward and are considered in section 1.4.

1.2 Realisation of Specifications

This section makes precise the notion of satisfying a specification. The property concerned with a specification being defined over some set is:

1 $R \underline{\text{defover}}\ S \Leftrightarrow S \subseteq \underline{\text{dom}}\ R$

(In this section P,Q,R will denote relations over V; S,T,V sets). Some obvious properties are:

2 $S \subseteq T \wedge R \underline{\text{defover}}\ T \Rightarrow R \underline{\text{defover}}\ S$

3 $R \underline{\text{defover}}\ S \wedge T \in V \leftrightarrow V \Rightarrow R;\ T \underline{\text{defover}}\ S$

4 $Q \underline{\text{defover}}\ (\underline{\text{dom}}\ R) \Rightarrow T;\ Q \underline{\text{defover}}\ \underline{\text{dom}}\ (T;\ R)$

The relation concerned with partial correctness is simply:

5 $Q \underline{\text{psat}}\ R \Leftrightarrow Q \subseteq R$

Some obvious properties are:

6 $P \underline{\text{psat}}\ R \Rightarrow S;\ P \underline{\text{psat}}\ R$

7 $Q \underline{\text{psat}}\ R \Rightarrow P;\ Q \underline{\text{psat}}\ P;\ R$

8 $P \underline{\text{psat}}\ S;\ R;\ T \Rightarrow P \underline{\text{psat}}\ R$

9 $S;\ P \underline{\text{psat}}\ R \wedge T;\ P \underline{\text{psat}}\ Q \Rightarrow (S \cup T);\ P \underline{\text{psat}}\ (R \cup Q)$

For a realisation, or another specification, to satisfy a given specification it must be both defined over an adequate domain and, over that domain, be consistent with the specification:

10 $R \underline{\text{sat}}\ SP \Leftrightarrow R \underline{\text{defover}}\ \underline{\text{dom}}\ SP \wedge (\underline{\text{dom}}\ SP);\ R \underline{\text{psat}}\ SP$

The converse relation is:

11 $Q \underline{\text{satby}}\ R \Leftrightarrow R \underline{\text{sat}}\ Q$

The relation sat is a partial order ($\gtrsim$) on relations (cf. 1.1.1(27)):

i R defover dom R, (dom R); R psat R

ii Q sat R $\wedge$ R sat Q $\Rightarrow$ dom R = dom Q

$\Rightarrow$ R = Q

iii P defover dom Q $\wedge$ Q defover dom R $\Rightarrow$ P defover dom R

(dom Q); P psat Q $\wedge$ (dom R); Q psat R $\Rightarrow$ (dom R); P psat R

$\therefore$ P sat Q $\wedge$ Q sat R $\Rightarrow$ P sat R

It is shown in chapter 3 that the non-deterministic language is monotone with respect to the sat order: it is this fact which justifies the use of the language constructs with imbedded specifications.

An alternative way of thinking about relations of the form of 1.1.2(2) is to regard them as functions:

12 $f \in St \rightarrow \wp(St_\perp)$

where the $\perp$ element corresponds to undefined:

13 $St_\perp = Stu\{\perp\}$

A given relation, R, corresponds to:

14 $fR = \lambda s \in St.$ if $s \in$ dom R then $\{s'$ st R:$s \mapsto s'\}$ else $\{\perp\}$

If the constructs of the programming language were defined on this basis, range elements would arise which contain both $\perp$ and elements of St. It is then possible to show that the satby order is the same as the Smyth order (Smyth /78a/):

15 P satby R $\Leftrightarrow$ fp $\sqsubseteq$ fR

16 $f \sqsubseteq g \Leftrightarrow (\forall s \in St;\ \perp \in f(s) \vee f(s) \supseteq g(s))$

1.3 Data Types

In section 1.2 it is assumed that the effect on the state is exactly the aspect of an operation which is to be specified. The term "abstract data type" has achieved wide use in spite of the fact that it is not always precisely defined. The view taken here is that an abstract data type is characterised by a set of operations; the operations are defined as above, but the state is considered to be hidden. Thus, it is only the behaviour in terms of certain given types which is constrained by the specification.

Early papers on program proofs almost invariably used factorial as an example. The "factorial" of the abstract data type papers is certainly the LIFO stack. Consider a simple stack of integers. Operations might be EMPTY to initialise an empty stack; PUSH to add an integer onto the stack; POP to yield the most recently inserted integer and to simultaneously remove it; ISEMPTY to determine whether or not the stack contains any integers. The required behaviour must be expressed in terms of the "given" data types Int and Bool. The state itself is important because of the effect that the operations have on each other. Section 1.6 discusses the "property oriented" approach in which the semantics of the operations are given by equations which relate the operations. Here, data types will be specified in terms of a model for the state. This model, however, serves only to link the operations together and its detailed structure is not part of the specification. It must be confessed that there are dangers of over-specification in such "model oriented" specifications and section 1.3.4 addresses this problem and others concerned with

choosing appropriate models. In large specifications the structure of the state plays a vital rôle in achieving a concise specification.

An abstract data type is, then, specified by a "type scheme" (cf. sub-section 1.3.1) and a "model" (cf. sub-section 1.3.2). Here again, this very formal framework is relaxed somewhat in subsequent chapters. Given two models for the same type scheme, it is possible to determine whether the behaviour of one of them satisfies the behaviour of the other with respect to the given types: the notion of "model satisfaction" is defined in sub-section 1.3.3.

It is perhaps worth observing that single operations like MULTP in 1.1.2(3) can be thought of in the abstract data type framework by including read and write operations for the variables.

1.3.1 Type Schemes

A type scheme for an abstract data type provides its name, the names of any given types and a list of types for the operations (signature). Thus:

```
1   name StackofInt
    given  Int, Bool
    signature
        EMPTY: → StackofInt
        PUSH:  StackofInt x Int → StackofInt
        POP:   StackofInt ≃→ StackofInt x Int
        ISEMPTY:  StackofInt → Bool
```

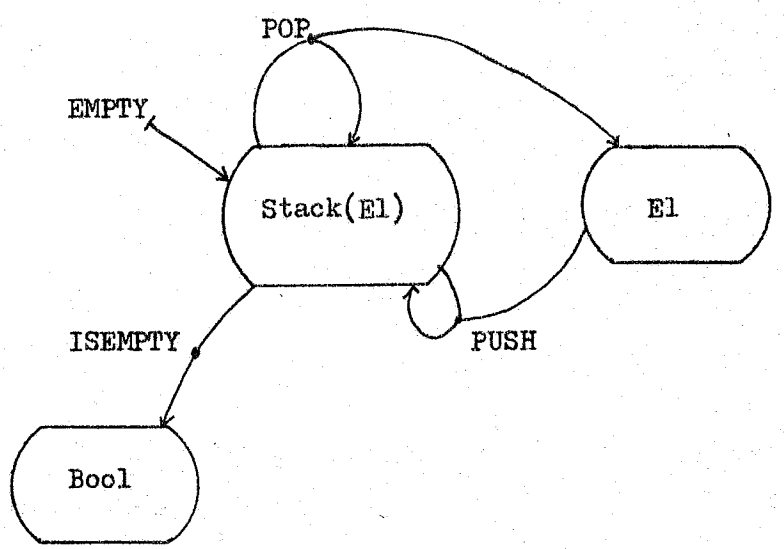
Notice that POP yields both a changed stack and a result. Furthermore, POP is marked as partial.

It is often expedient to allow for the "given" sets to be chosen differently for different instances of the abstract data type. This brings in the idea of parameterised data types and more is said about constraining arguments to such a data type in chapter 2. A more general stack might have the type scheme:

2 name Stack(El)
given Bool, El (no properties required of El)
signature

EMPTY: → Stack(El)
PUSH: Stack(El) x El → Stack(El)
POP: Stack(El) ≃→ Stack(El) x El
ISEMPTY: Stack(El) → Bool

A readable way of presenting signatures is to use "ADJ diagrams" (cf. Goguen /75a/), thus:



1.3.2 Models

A model provides a carrier and an association of appropriate relations to the operations. The carrier is a set of known objects.

The relations associated with each operation must respect the signature. The relations may in general be partial but in specific cases may be total functions. Thus (using the notation of section 2.1 for lists) a model for 1.3.1(2) is:

1 Stack(El) = El-list

2 EMPTY = $\{ <> \}$

PUSH = $\{ (s,e) \mapsto \langle e \rangle \frown s \ \underline{st}\ e \in El \wedge s \in El\text{-list} \}$

POP = $\{ s \mapsto (\underline{ft}\ s, \underline{end}\ s) \ \underline{st}\ s \in El\text{-list} \wedge s \neq <> \}$

ISEMPTY = $\{ s \mapsto (s = <>) \ \underline{st}\ s \in El\text{-list} \}$

Notice that the relations denoting the operations here are all functional and that only POP is partial.

Lockwood Morris provided a useful, but simple, example of a non-deterministic specification in proposing a symbol table (the mapping notation used here is defined in sub-section 2.1.4):

3 name Symtab
given Id, Addr
signature
EMPTY: → Symtab
LOOKUP: Id×Symtab → Addr×Symtab

4 carrier
Symtab = Id $\xrightarrow[m]{\sim}$ Addr

5 EMPTY = $\{ \{\} \}$

LOOKUP = $\{ (id,m) \leftrightarrow (r,m') \ \underline{st}\ (\underline{if}\ id \in \underline{dom}\ m\ \underline{then}\ m' = m \wedge r = m(id)$
$\underline{else}\ (\exists a \in Addr\ \underline{st}\ a \notin \underline{rng}\ m;$
$r = a \wedge m' = m \dagger \{id \mapsto a\}\ ))\}$

Different models of Symtab can be considered. On the one hand it might be possible to find a one-one function between Id and Addr. On the other hand, a realisation might build up a list of Ids and use

their index as an Addr. Either of these realisations can be shown to be models of 3-5. However, neither of them is acceptable as a specification because they preempt the non-deterministic choice.

An interesting example of the use of parameterised data types is a formulation of the "compiler dictionary" problem of Guttag /77a/. It is straightforward to define a Localdict on a carrier:

Localdict = Id $\overset{m}{\rightharpoonup}$ Attrs

A dictionary can then be defined in terms of the stack of 1.3.1(2), 1, 2 as:

Cdict = Stack (Localdict)

In such a definition there is clearly a naming problem which is not addressed in the current work. For a more language oriented view of this, the reader is referred to Burstall /80a/.

1.3.3 Satisfaction of Models

Given two models of the same type scheme, it may be true that the behaviour of one of them, with respect to the given types, satisfies the behaviour of the other. This sub-section makes the notion of model satisfaction more precise but avoids being completely formal because some technical problems would cause such a definition to be rather heavy. A basic technique for proving model satisfaction is also explained, but this is developed further in sub-section 1.3.4 and chapter 2.

For any particular signature, it is possible to define the set of "valid terms". (One of the technical difficulties alluded to above

is that operations which deliver multiple results complicate this definition.) Furthermore, it is possible to determine the type of any such expression. For example, using the signature in 1.3.1(2):

1
```
(let s1,r1 = POP(PUSH(PUSH(EMPTY(),e1),e2))
 let s2,r2 = POP(s1)
 s2,r1,r2)
        ⊆ (El x El) x (Stack(El) x El x El)
```
+

The treatment here follows, to some extent, that in Bothe /79a/. It is tempting to seek a simplification of the set of valid terms and their types by employing Bothe's technique of using the distinction between constructor and selector operations. Such a dichotomy would avoid the problem of results which are mixed states and "given values" and reduce the set of terms to be considered to arbitrary sequences of constructors to which one selector is applied. Unfortunately there are difficulties in this plan. It is clear that POP in 1.3.1(2) is both a constructor in that it creates objects of type Stack(El) and a selector in that it yields an object of the given type (El). In this case it is straightforward to split the operation into:

2
```
INSPECT: Stack(El) ≃→ El
REMOVE:  Stack(El) ≃→ Stack(El)
```

This division is, however, only acceptable because the POP operation is deterministic. Consider an alternative data type:

+ Notice that all valid terms will be built up using a zero argument constructor. Thus several objects of the defined data type can occur as "results" but not as "inputs".

3 name Set37
given Nat
signature
INIT: → Set37
ACCESS ⊆ Set37 x Set37 x Nat
model
Set37 = Nat-set
INIT = { {3,7} }
ACCESS = {s ↔ (s - {e}, e) st s ≠ {} ∧ e∈s}

The two possible results of performing two ACCESSes after INIT yield the values 3 and 7 in either order. If, however, ACCESS is split into its constructor and selector part:

4 REMOVE ⊆ Set37 x Set37
INSPECT ⊆ Set37 x Nat

two INSPECT/REMOVE pairs may result in the value 7 appearing twice. Nor does it seem to be acceptable to rely on a choice function: the specification in 3 defines a non-determinism which, in a larger example, may be resolved by things other than the content of the set.

After this digression, the reader is hopefully prepared to accept the notion of valid terms and their types as exemplified in 1.

For any model, each valid term will be associated with a relation of the appropriate type. This relation will be empty in the case that an operation is used outside its domain. The notion of model satisfaction depends only on the given (or external) types and a function which simply drops those positions in a relation corresponding to the carrier can be defined. For 1:

5 giventypes: (El x El) x (Stack(El) x El x El) → (El x El) x (El x El)

A model M2 is said to satisfy another (M1) if:

6 $(\forall t \in \text{validterms};\ \text{giventypes}(t \text{ in } M2)\ \underline{\text{sat}}\ \text{giventypes}(t \text{ in } M1))$

Thus model satisfaction inherits from the notion of satisfaction the fact that a valid implementation may be more often defined or more closely defined than a specification.

It would be unfortunate if the only way to establish model satisfaction were by reasoning about "all valid terms". A highly workable approach to data refinement proofs is presented in chapter 2; the first step towards this is to consider a <u>model satisfaction proof</u>.

Given two models M1 and M2 with carriers St1 and St2 respectively, a relation between the carriers is sought:

7 $REL \in St2 \leftrightarrow St1$

This relation is the cornerstone of a model satisfaction proof. In essence, it links the elements of one carrier with those in the other which will exhibit the same behaviour with respect to the given types. To see that this relation can indeed be many-many, consider two rather strange models for the signature in 1.3.1(2): one retains, in addition to the essential information, the first value which was ever placed in the stack; with equal perversity, the other model redundantly preserves the last value removed from the stack. If one of these models were to be shown to satisfy the other, the relation 7 would have to be many-many. Sub-section 1.3.4 discusses how this problem can be avoided.

Given a relation of the form 7, it is possible to extend it to a relation on the signatures of each operation: equality being the requirement on the given types. Thus:

8 REL^{-1}; POP2; $(REL \times E_{El})$ sat POP

A model satisfaction proof then involves showing that the appropriate extended relations establish such an equation for each operation. The fact that such a proof is adequate to ensure the earlier notion of model satisfaction is not surprising: it is only a generalisation to relations of standard algebraic ideas. Specifically, to show that the proof justifies the conclusion that the psat relation holds for "all valid terms" involves an induction on the length of valid terms. The key to the proof is to note that:

9 REL; REL^{-1}

contains the identity relation since REL is total (cf. 1.1.1(34)). A lemma on defover is similar and the overall result relies on the fact that the function giventypes drops all but the given values and it is here that the extended relation gives equality.

Examples of such proofs are given in sub-section 1.3.4 after a simplification in the proof method has been discussed.

1.3.4 Implementation Bias

Even with simple examples like Stack, model satisfaction proofs as introduced in sub-section 1.3.3 are unnecessarily difficult. In

cases where the operations themselves are defined by relations, the use of a relation on the carriers (cf 1.3.3(7)) is very cumbersome. The reason for already reviewing the material on proofs in this first chapter now becomes clear: in this sub-section, a criterion is proposed for specifications which will simplify subsequent refinement proofs. In particular, necessary conditions are given which ensure that the carrier relation becomes a "retrieve function" (homomorphism) - see chapter 2.

The first observation, then, is that a good specification should have the property that, for any chosen model, the satisfaction proof can be performed with a functional relation between the carriers. How is this to be achieved? Clearly, it is not desirable to have a test based on all models. Fortunately, a more convenient test has been suggested (Jones /77b/).

Suppose a model is such that two different elements of its carrier cannot be distinguished by any of the operations of the data type. Such a model has, in some sense, a redundancy; it will be said to be biased. That is, with such a specification some representations will be easier to prove correct than others.

A good specification is one without bias.

The first test, then, is whether all distinct elements of the carrier can be distinguished by terms of the operations. An even simpler test is offered below, but first it is worth reviewing an example. Consider the following type scheme:

1 name Set
given El, Bool
signature
INIT: → Set
ENTER: Set x El ~→ Set
ISPRESENT: Set x El → Bool

(The example is presented as a "students in classroom problem" in Jones /80a/.) An obvious carrier for this problem is El-set, but suppose that instead El-list is chosen, as follows:

2 Set = El-list
INIT = { <> }
ENTER = {(l,e) ↔ l⌢<e> st e ∉ elems l }
ISPRESENT = {(l,e)↔(e∈ elems l)}

There are then many (n factorial) representations of a set (of cardinality n). For example:

<e1,e2> and <e2,e1>

both represent the same set. More interestingly, there is no term in the operations which can distinguish these two elements of the carrier. The model in 2 is therefore said to be biased.

In many cases, the test for bias can be extended to become a positive rule. If a function can be defined, solely in terms of the operations of the signature, which decides equality of the carrier, then the model is without bias. Consider, for example, the type scheme of 1.3.1(2) and model 1.3.2(1,2) - an equality function is:

3 eq: El-list×El-list → Bool
eq(l1,l2) ≜
if ISEMPTY(l1) ∧ ISEMPTY(l2) then TRUE
else if ISEMPTY(l1) ∨ ISEMPTY(l2) then FALSE
else (let l1',e1 = POP(l1)
let l2',e2 = POP(l2)
e1 = e2 ∧ eq(l1',l2'))

If, however, a model had been based on a carrier with a list plus a counter, a danger of bias would exist. For example, if the POP operation were to reduce the counter by one but remove nothing from the list then the following two elements of the carrier could not be distinguished by the operations:

<a,b>,1 , <a,c>,1

Notice that there is nothing wrong with such a definition in terms of its external behaviour - it is identical with the original model. It is only that proving some representations correct with respect to it would be more difficult than with 1.3.2(1,2). In fact, it is exactly when trying to prove that two differently biased specifications model each other that a many-many relation between carriers is required (cf. example in sub-section 1.3.3).

It is interesting to consider the relationship between the operations and the (unbiased) models. If the POP operation is split as in 1.3.3(2), it is possible to consider dropping some of the operations. Without INSPECT the model of 1.3.2(1,2) is biased in that any two stacks of the same size cannot be distinguished. An unbiased model for a data type with the reduced set of operations is simply a counter (Nat0). If, in addition, the REMOVE operation were

deleted, even this simple counter would be biased and a single Boolean value to record whether anything had ever been put on the stack would be the carrier for the appropriate unbiased model. Reverting to the full signature (1.3.1(2)/1.3.3(2)) other cases can be considered. If the REMOVE operation is absent, only the last value on the stack can ever be accessed and the appropriate model is a single element.

In each of the cases considered so far, it is obvious how to establish the lack of bias by defining the equality test. If, however, the ISEMPTY operation were deleted, it would no longer be possible to give a decidable equality test: either partially decidable predicates or the more general argument given above about "all valid terms" must be used.

The next point to address is the sufficiency of a function (from representation to specification) in a model satisfaction proof. Although the full rules about such "retrieve" functions are discussed more fully in chapter 2, an informal argument is given here. Suppose the relation between the carriers (cf. 1.3.3(7)) related one representation element to two different abstract elements. If the functions based on the representation were correct, then there could be no way of distinguishing between the two abstract elements. Therefore the model which does distinguish them would be biased and should not have been used as a specification.

In fact a biased model can still be used as a specification for some implementations. Consider, for example, the biased stack

model which preserves elements internally after they have been POPed off the stack. A representation which preserves the complete history of the operations can easily be proven correct. The objection is that the model is biased towards implementations which preserve at least as much information.

After the discovery of the test for bias, it has been applied to many existing specifications. The fact that very few biased models were uncovered is a comment on the relative ease of using the model oriented approach to specifications. One exception is in common enough use to warrant comment. In many papers (e.g. Owicki /75a/) a buffer is modelled by a list of fixed length and two counters. There is then a problem of distinguishing the empty and full buffers. One way in which this difficulty is circumvented is by making the counters record the total numbers of inserted and removed elements. Apart from bringing gratuitous overflow problems, this is obviously biased because none of the buffer operations can distinguish between queues which only differ by their "age". Another example which arises in section 2.1 is the definition of a bag (i.e. multiset): a model based on lists is biased so the specification given is based on mappings.

Clearly, it is possible that more than one unbiased model exists for a data type. In this case the relation between the carriers becomes a one-one function (isomorphism). There may, however, be other criteria which prompt a preference for one of a family of isomorphic models. Reverting to the example of 1,2, a model could be given based on ordered lists (the order would be fixed by a data type

invariant - cf. chapter 2): such a model would be unbiased! Although surprising, this corresponds to the rôle of (unbiased) specifications in that a retrieve function can always be written to this carrier simply by arranging for a sort. The model is, however, intuitively less satisfactory than one based on sets and an additional specification rule can be derived which gives preference to carriers with simpler data type invariants. Another example of this rule is provided by the example in 2.2.1(9). A model based on two sets requires a data type invariant asserting that the two sets are to be disjoint; an isomorphic, but simpler model, can be based on a mapping to Boolean values.

The notion of (lack of) bias for abstract data types can be compared with that of "full abstraction" in denotational semantics.

1.4 Specification via Predicates

Relations are used as the underpinning of the notions of specification and satisfaction but it is easier to reason about particular programs if the specifications are given by predicates. For a simple, non-deterministic, function the specification might be given by a type clause, pre-condition and post-condition:

1
arbs: El-set $\xrightarrow{\sim}$ El
pre-arbs(S) $\triangleq$ S $\neq \{\}$
post-arbs(S,r) $\triangleq$ r$\in$S

The pre-condition is in general a predicate of all inputs and the post-condition is an input-output relation constraining the relation between all inputs and all outputs. The purpose of this section is to propose extensions to this format to cover operations without making the specification too cumbersome.

"Operations" will be specified by defining a set of states; a pre-condition which is a predicate of one state; and a post-condition which is a predicate of two states, thus:

2
St = ...
pre: St $\rightarrow$ Bool
post: St x St $\rightarrow$ Bool

The pre-condition can be interpreted as defining a subset of the universe of states over which the operation must terminate; about states which do not satisfy the pre-condition, the specification has nothing to say. The post-condition can be interpreted as defining which final states are acceptable for any valid initial state. Notice that operations are again specified to be partial and may be under-determined.

It will not be permissible to further constrain the initial states in a post-condition, thus:

3 $(\forall s \in St;\ pre(s) \Rightarrow (\exists s' \in St;\ post(s,s')))$

A specification in the form of 2 also requires that the final states are in the defined set. (The alternative to using data type invariants, as in Chapter 2, to limit sets is to put the onus on the post-condition. This is more cumbersome.) Strictly, it is always necessary to prove the existence of a result satisfying the post-condition. However, since the design of a program is a constructive proof, a formal existence proof is only justified in very rare cases.

The meaning of a specification via predicates (cf. 2) is given by fixing the relation it denotes:

4 $R = \{ s \leftrightarrow s' \in St \times St \ \underline{st}\ pre(s) \wedge post(s,s') \}$

To emphasise the point about termination, the notation of chapter 3 can be used to state that for a program P to be valid with respect to 2, it must be true that:

5 $(\forall s \in St \ \underline{st}\ pre(s);\ s \in T\ [\![P]\!]\ )$

From the definition of sat (cf. 1.2(10)) it is immediately obvious that a given specification (preg/postg) will be satisfied by pren/postn if the new pre-condition is weaker and the new post-condition is stronger:

WKNPRE/ STRPOST

$(St,pren,postn)\ \underline{sat}\ (St,preg,postg)$ if
$(\forall s \in St\ \underline{st}\ preg(s);\ pren(s)) \wedge$
$(\forall s,s' \in St\ \underline{st}\ preg(s);\ postn(s,s') \Rightarrow postg(s,s'))$

In Jones /80a/ the reference, within predicates, to components of the state was either by positional parameter or selector. An alternative is used here which results in a clearer and more compact specification. The idea is to list for each operation the global variables to which it has access, their types and whether read or write access is allowed (cf. glocon/glovar in Dijkstra /76a/). The values of variable names (upper case) are referred to by the corresponding lower case identifiers; primed identifiers denote final values. Thus the example of 1.1.2(3) might be written:

6 MULTP
globals X:rd Int, Y:rd Int, R:wr Int
pre y ⩾ 0
post r' = x * y

But notice that this prevents any overwriting of the variables X and Y. The original specification is regained if "rd" is changed to "wr" for both variables in the list of globals.

One advantage of this form of specification is that it can be interpreted in a "larger state". Given a list of globals and a state with at least its variables, the post-condition can be completed by adding clauses which require that read only or unreferenced variables are unchanged. Thus with:

7 FACT
globals N:rd Int, FN:wr Int
pre n ⩾ 0
post fn' = n!

in a state with variables N, FN and X, the completed post-condition is:

8 postFACT fn' = n! ∧ n' = n ∧ x' = x

(Notice how the names of pre- and post-conditions are generated when they are used outside the specification.) If, however, N had been a wr global then no constraint would have been generated: neither the post-condition nor the access constraints inhibit a realisation from overwriting this variable. An instance of how this specification style saves writing unnecessary "frame predicates" can be given by specifying the initialisation:

9
```
INIT
globals FN:wr Int
post fn' = 1
```

This post-condition completes, in a state with N and FN to:

10 fn' = 1 ∧ n' = n

(Notice that pre-conditions are omitted when TRUE - i.e. for total operations.)

A further advantage of the use of specifications using globals is apparent if parallelism is considered: it is then not enough to know that a value is the same in the final as in the initial state - it is often necessary to know that a process cannot possibly change the value.

The general operations of Jones /80a/ include input parameters and results. The specification style adopted here is to fit these into the syntax of a programming language. For example, in Ada:

11
```
function F (X:in T1) return RES:T2;
    globals A:wr T3, B:wr T4
spec
    pre ...
    post ...
end
```

1.5 Specifying Order

Although it is clear that a specification should state WHAT a system should do (rather than HOW it is to work), not all systems fit the input/output picture naturally. For example a database system with update and query transactions must define the order in which transactions are to be processed. One approach (used in "Z" - cf. Abrial /79a/) is to assume that the entire sequence of transactions is available as a state component. But even then it is necessary to define state transitions so that the transactions are processed in sequence.

The approach adopted in VDM is to include, when necessary, features for defining order in the meta-language. (Individual components can still be specified by post-conditions.) The capability to define order is anyway mandatory because of the wish to use the meta-language in the design of a system: a design is often defined by decomposing a task into a number of sub-operations which are to be used in some particular order.

It is necessary to choose between using the sequencing constructs of a particular programming language or defining a set of combinators for the meta-language. "Meta-IV" adopted the latter course (cf. Jones /78a/). The combinators are chosen to look familiar to programmers but their semantics is defined denotationally. Relatively little concern was given to the problems of non-determinism and the definition of arbitrary order of sub-expression evaluation in Bekič /74a/ is not entirely satisfactory.

In this dissertation, chapter 3 defines a non-deterministic programming language fragment which is used to record design decomposition. The semantics is given denotationally in terms of relations.

For further discussion of the overall structure of a system specification see Bjørner /78a/ (see also Jackson /80a/).

1.6 Alternatives

This section considers some of the alternative approaches to recording (formal) specifications. The work on abstract data types has created an extensive literature. For a fairly recent bibliography see Dungan /79a/.

1.6.1 Specification of Single Operations

The standard literature on program correctness proofs (e.g. Floyd /67a/, Hoare /69a/, Dijkstra /76a/) uses post-conditions which are predicates of single states. Since a program is normally required to realise some input/output relation, this is not in itself adequate. One way of relating the final values to the starting values is to use free variables for the latter. (Manna /69a/ does try to avoid these free variables but makes a division of the state components into input/internal/output variables.) It is clear that relations between an input and an output state provide a more natural model of a specification. The cost, however, of using post-conditions which are predicates of two states is that the proof rules associated with justifying program decomposition become more cumbersome (cf. chapter 3). This aspect of the comparison is discussed in section 3.5.

In Dijkstra /76a/ use is made of variables whose value is "held constant": an approach based on a systematic treatment of this idea is the "specification logic" of Reynolds /81a/.

1.6.2 Abstract Data Types - Model Oriented Specification

The approach taken in this chapter to the specification of abstract data types is "constructive" or "model" oriented. The "Z" approach in Abrial /79b/ is also based on specifying via a model (more is said about "Z" in connection with parallelism in chapter 6).

"Z" objects are viewed as mappings from selectors to values. Based on this denotation it is straightforward to combine operations ("&") on two different, but overlapping, states into new (atomic) operations. This is certainly an advantage over the view of constructed objects taken here (cf. sub-section 2.1.10). The wish to be able to distinguish classes with identifical components is the reason for the constructor function approach of "Meta-IV".

The "CLEAR" language (Burstall /80a/) permits both model and property oriented specifications. This is obviously a way to offer the best of both worlds. The semantics of "CLEAR", including parameterised types, have also been precisely defined (Burstall /80b/). The denotational semantics is based on Category theory (see also Lehmann /78a/).

Both "Z" and "CLEAR" attempt to provide rigid syntaxes for specification languages like those of programming languages. There are obviously advantages in doing this if mechanical processing is envisaged (see section 7.4). There are, however, many issues in the design of specifications which have yet to be resolved: a concern

with syntax would appear to violate Christopher Strachey's first law of language design (cf. Stoy /77a/):

"Decide what you want to say before you worry about how to say it".

1.6.3 Abstract Data Types - Property Oriented Specifications

Most of the literature on abstract data types makes a virtue of avoiding the model oriented approach which is discussed above (e.g. Lucas /69a/, Zilles /80a/ and references therein, Guttag /77a/). The general idea is to provide a signature of the types of the operations and to define the semantics by equations which relate the operations to each other. Since there is no model provided, the danger of bias discussed in sub-section 1.3.4 is avoided.

This overall approach is referred to in several different ways. The term "axiomatic" comes from viewing the equations as axioms for the operators. The term "algebraic" is slightly confusing since a text book like MacLane /79a/ which uses Peano's axioms to characterise the natural numbers, gives a construction for the rational numbers. Here, the term "property oriented" is used.

Several authors have taken advantage of both model and property oriented specifications to characterise data types - Burstall /77a/, Bothe /79a/, Ehrig /80a/. It is interesting to quote from Burstall /80a/:

"The abstract [property oriented] method is more elegant but more prone to mistakes".

There are a number of recognised problems with the property oriented approach. Certain fairly simple data types require either "hidden functions" or infinite collections of properties (cf. Veloso /79a/). Furthermore, as the above quote suggests, it is not easy to be sure that a property oriented specification is correct: the titles of Veloso /79a/ and Veloso /79b/ make this point quite well - a model oriented specification of the data type in question is a simple classroom exercise.

Another difficulty with property oriented specifications is the dissimilarity of specifications for basically similar data types. For example, to change the model of the LIFO stack in 1.3.1(2), 1.3.2(1,2) into one for a FIFO queue it is only necessary to change the order of the concatenation (and, presumably, rename the operators). The property oriented specifications for these two concepts are very different and a finite stack requires a "hidden function".

The beguiling simplicity of an axiom like:

1 REMOVE(PUSH(st,e)) = st

is slightly confusing. The equality must be interpreted (for some implementations) as a link between equivalence classes. Thus proving that (biased) stack implementations satisfy property 1 still requires something like retrieve functions. There are also a number of technical details like the use of total operations and the need to have deterministic results which are consequences of the normal mathematical view of algebras. (Bill Rounds is studying ways of

avoiding these restrictions and Bothe /80a/ tackles non-determinism - see also Ehrig /80a/.)

The above comments should not be interpreted as an attempt to denigrate the property oriented approach. It is argued in Jones /81a/ that distinct rôles can be found for the different approaches (in the same way that constructive and axiomatic semantics of programming languages are useful for different purposes): property oriented specifications are required for those objects which are to be used in other programs; model oriented specifications are more useful where the task is to construct the object.

It is clear that some formal connection between the approaches would be valuable. It is interesting that Guttag /80a/ moves quite a long way towards using parts of both property and model oriented specifications: the paper actually "axiomatises" what a model would make:

2 $SD = Ln \xrightarrow[m]{} Line$

The firmest theoretical basis for property oriented specifications is provided by the ADJ group (e.g. Goguen /75a/). The general idea is to define the initial algebra by using the equations (properties) to divide the free algebra of the signature. The requirement, that implementations are proved by finding a homomorphism from the initial algebra to the representation, is the reverse of the retrieve functions considered here (cf. Bothe /79a/). The problem of proving implementations correct has prompted work on "final algebras" (Kamin /80a/, Wand /77a/) and "functional specifications" (Ehrig /80a/).

Chapter 2

Data Refinement

Historically the work on data refinement (e.g. Milner /71a/) followed that on program decomposition proofs (e.g. Naur /66a/, Floyd /67a/, Hoare /69a/). This remains true even if one looks further than the standard references: Rod Burstall recalls Christopher Strachey describing data refinement ideas in the mid-1960's - but program flow proof ideas occur in Turing /49a/ and Goldstine /47a/. However, it is now widely accepted that data refinement proofs are of greater importance than those of program decomposition. Furthermore, experience in teaching the two subjects in the Oxford M.Sc. in Computation suggests that it is better to tackle data refinement first. One reason for this emphasis is that specifications of large systems (e.g. Bjørner /78a/, Bjørner /80a/, Bjørner /81a/) are built around a state. In fact, it is often possible to document most of the important aspects of a system solely by recording the state (cf. Pascal in Tennent /81a/).

Another argument for giving priority to data refinement proofs is that they are used in the earlier design phases of large projects. Since errors which are introduced early are likely to be very expensive to correct, it is more important to employ formal methods here rather than later in the design process.

This chapter presents the basic objects which are used throughout the current work. Section 2.1 does this less formally than is suggested by section 1.3. The idea of developing "theories" of data types (cf. Dahl /78a/, Reynolds /79a/, Jones /79a/, Abrial /80b/, Burstall /77a/) is emphasised. The collection of such bodies of knowledge is one of the crucial steps necessary to begin to make

program development into an engineering discipline: only in this way can the necessity to start each proof "from scratch" be avoided. The choice of appropriate concepts and lemmas can also make subsequent proofs far shorter and more intelligible. The material from section 2.1 is used freely below. In fact, not all of the material is required below, this being one of the areas where the work presented here is part of a larger plan. The reader is therefore recommended to pass over section 2.1 and consult it as necessary below.

Refinement proofs are covered in section 2.2. An attempt is made in section 2.3 to achieve the same sort of generalisation with refinement proofs which can be obtained by the use of theories of objects. That is, results about classes of refinement proofs are considered.

In defining the types of infix or prefix operators, Burstall /80a/ is followed, e.g.:

1 $\_\_ \leqslant \_\_ : \text{Int} \times \text{Int} \rightarrow \text{Bool}$

2 $\sim \_\_ : \text{Bool} \rightarrow \text{Bool}$

A number of operators are distinguished below by underlining (e.g. card). The choice of which functions enjoy the privilege of no parentheses around their arguments is mainly governed by frequency of use. When considering several data types it appears difficult to follow Reynolds /79a/ in using specific graphic symbols for predicates and operators.

Data type invariants are predicates which restrict a set of objects. For example:

3 Name ⊆ O:Obj
inv ...o...

They can play a very useful rôle in recording information about a specification. Chapter 10 of Jones /80a/ argues that such restrictions are useful both to implementors and for subsequent revision of a specification. The set of objects denoted by 3 is:

4 Name = {o ∈ Obj st ...o...}

Because of the way in which specifications of the form of 1.4(2) are interpreted, a data type invariant is an extra constraint on an operation.

The same notation is used when assumptions are to be recorded about the parameters of parameterised data types. For example:

5 Poset(El)
with $\_\_ \leq \_\_$: El x El → Bool
inv $(\forall x, y, z \in El;\ x \leq x \wedge$
$(x \leq y \wedge y \leq x \Rightarrow x = y)$
$(x \leq y \wedge y \leq z \Rightarrow x \leq z))$

2.1 Objects and their Properties

The objects, operators and their properties used in the current work are presented in this section. In each sub-section, a set of basic operators is presented first followed by others which are defined, in terms of the basic operators, directly (by recursion) or pre-/post- conditions.

2.1.1 Auxiliary Arithmetic Operators

1 min: I:Int x J:Int → R:Int
post $(r = i \vee r = j) \wedge r \leq i \wedge r \leq j$

2 max similar

3 __ mod __ : I:Int x J:Int ⇸ R:Int
pre $j \neq 0$
post $(0 \leq r < j \vee j < r \leq 0) \wedge$
$(\exists m \in Nat0;\ m * j + r = i)$

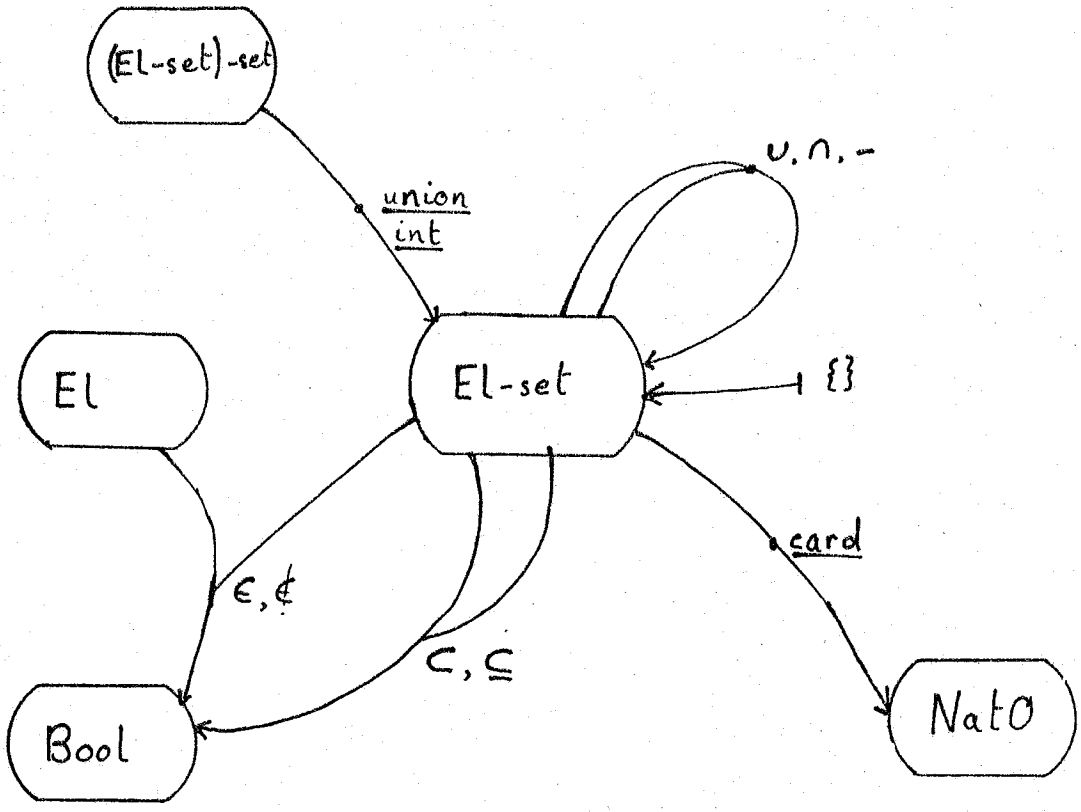
4 isdivisor: I:Int x J:Int ⇸ B:Bool
pre $j \neq 0$
isdivisor(i,j) ≙ i mod j = 0

5 iscommonfactor: I:Int x J:Int x K:Int ⇸ B:Bool
pre $k \neq 0$
iscommonfactor(i,j,k) ≙ isdivisor(i,k) ∧ isdivisor(j,k)

6 ishcf: I:Int x J:Int x K:Int ⇸ B:Bool
pre $k \neq 0$
ishcf(i,j,k) ≙ iscommonfactor(i,j,k) ∧
$\sim(\exists k' \in Int$ st $k < k';$ iscommonfactor(i,j,k'))

2.1.2 Sets

The basic operators are introduced in section 1.1. The signature can be represented by:



In writing specifications it will normally be sufficient to consider finite sets of elements. Thus El-set is a subset of the power set of El containing only the finite sets.

Properties of the set operators are used in proofs without comment (identity, associativity, commutativity, distributivity and absorption). The subset order ($\subset$) provides a well-founded order on El-set.

Further notation for sets is used:

1 $\{i\,..\,k\} \triangleq \{j \in \mathrm{Int}\ \underline{\mathrm{st}}\ i \leq j \leq k\}$

2 isdisj: El-set x El-set → Bool
isdisj (S1,S2) ≜ S1 ∩ S2 = {}

3 isdisjs: (El-set)-set → Bool
isdisjs (SS) ≜ (∀S1,S2 ∈ SS st S1 ≠ S2; isdisj (S1,S2))

4 maxs: S:Poset(El) ⇸ E:El
pre S ≠ {}
post e ∈ S ∧ (∀d∈S; d ≤ e)

5 mins similar

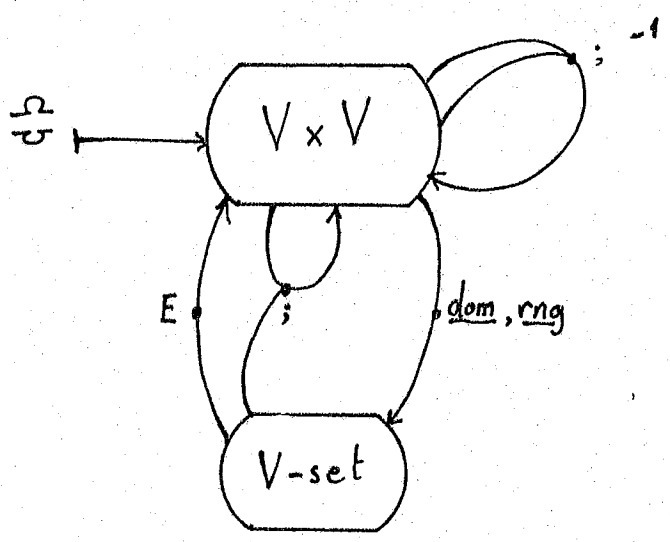
6 applys: El-set x (El → Elp) → Elp-set
applys (S,f) ≜ {f(e) st e ∈ S}

Notice that, strictly, this should be defined as:

7 $\{r \in \mathrm{Elp}\ \underline{\mathrm{st}}\ (\exists e \in S\ ;\ r = f(e))\}$

2.1.3 Relations

The operators are as introduced in section 1.1. Thus:



2.1.4 Mappings

Finite functions are often sufficient to write a specification. Such mappings will be defined by:

1 $D \overset{\sim}{\underset{m}{\Rightarrow}} R$

2 $D \overset{}{\underset{m}{\Rightarrow}} R$

The domain operator is used freely on mappings. Other operators are:

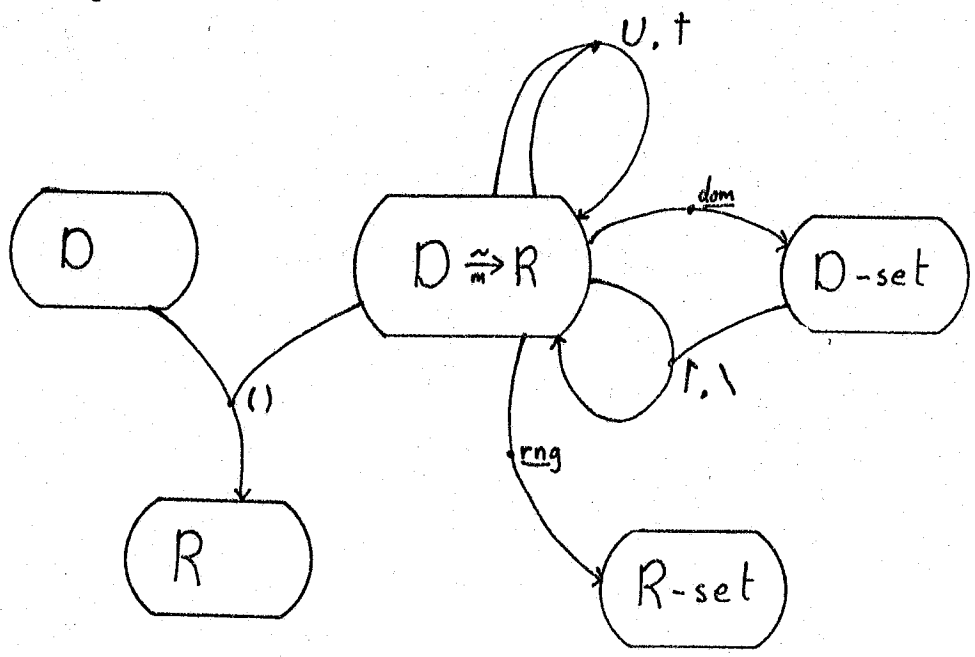
3 $\_\_ \dagger \_\_ : (D \overset{\sim}{\underset{m}{\Rightarrow}} R) \times (D \overset{\sim}{\underset{m}{\Rightarrow}} R) \to (D \overset{\sim}{\underset{m}{\Rightarrow}} R)$
$M1 \dagger M2 = \{d \mapsto (\underline{\text{if}}\ d \in \underline{\text{dom}}\ M2\ \underline{\text{then}}\ M2(d)\ \underline{\text{else}}\ M1(d))\ \underline{\text{st}}\ d \in (\underline{\text{dom}}\ M1 \cup \underline{\text{dom}}\ M2)\}$

4 $\_\_ \cup \_\_ : M1\colon (D \overset{\sim}{\underset{m}{\Rightarrow}} R) \times M2\colon (D \overset{\sim}{\underset{m}{\Rightarrow}} R) \overset{\sim}{\to} (D \overset{\sim}{\underset{m}{\Rightarrow}} R)$
$\underline{\text{pre}}$ isdisj ($\underline{\text{dom}}$ M1, $\underline{\text{dom}}$ M2)
$M1 \cup M2 = M1 \dagger M2$

5 $\_\_ \restriction \_\_ : (D \overset{\sim}{\underset{m}{\Rightarrow}} R) \times D\text{-set} \to (D \overset{\sim}{\underset{m}{\Rightarrow}} R)$
$M \restriction S = \{d \mapsto M(d)\ \underline{\text{st}}\ d \in (\underline{\text{dom}}\ M \cap S)\}$

6 $\_\_ \setminus \_\_ : (D \overset{\sim}{\underset{m}{\Rightarrow}} R) \times D\text{-set} \to (D \overset{\sim}{\underset{m}{\Rightarrow}} R)$
$M \setminus S = \{d \mapsto M(d)\ \underline{\text{st}}\ d \in (\underline{\text{dom}}\ M - S)\}$

Notice map union is commutative - this is, in fact, the only reason for introducing another symbol. The ADJ diagram for the mapping operators is:

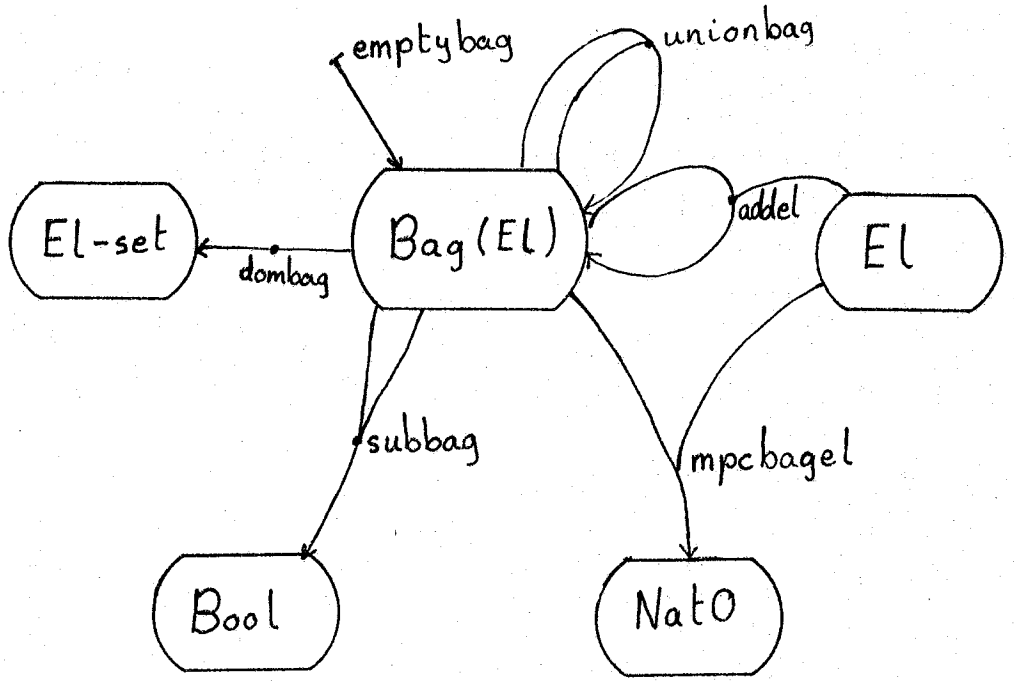


2.1.5 Bags (Multisets)

Multisets are a convenient halfway house between sets and lists in that the multiplicity of an element is significant while order is not. As such, they are useful in many specifications. Once again, finiteness will be assumed. Thus:

Bag(El) = El $\xrightarrow[m]{\sim}$ Nat

The operators on bags are:



1 emptybag( ) = $\{\}$

2 mpcbagel(b,e) $\triangleq$ if e $\in$ dom b then b(e) else 0

3 dombag(b) $\triangleq$ dom b

4 unionbag: B1: Bag(El) x B2: Bag(El) $\rightarrow$ R:Bag (El)
post dombag(r) = dombag(b1) $\cup$ dombag (b2) $\wedge$
($\forall$ e $\in$ dombag(r); mpcbagel(r,e) =
mpcbagel(b1,e) + mpcbagel(b2,e))

Notice that identity, associativity and commutativity properties do hold for bag union whereas absorption does not.

5 subbag(b1,b2) $\triangleq$ dombag(b1) $\subseteq$ dombag(b2) $\wedge$
($\forall$ e $\in$ dombag(b1);
mpcbagel(b1,e) $\leq$ mpcbagel(b2,e))

6 The irreflexive version of subbag is a well-founded relation on (finite) bags. This result follows from the fact that the sum of the multiplicities of elements must be finite.

A weaker ordering is given in Dershowitz /79a/.

7 addelbag: B:Bag(El) x E:El $\rightarrow$ R:Bag(El)
post dombag(r) = dombag(b) $\cup$ $\{e\}$ $\wedge$
mpcbagel(r,e) = mpcbagel(b,e) + 1 $\wedge$
($\forall$ f $\in$ dombag(r) st f $\neq$ e;
mpcbagel(r,f) = mpcbagel(b,f))

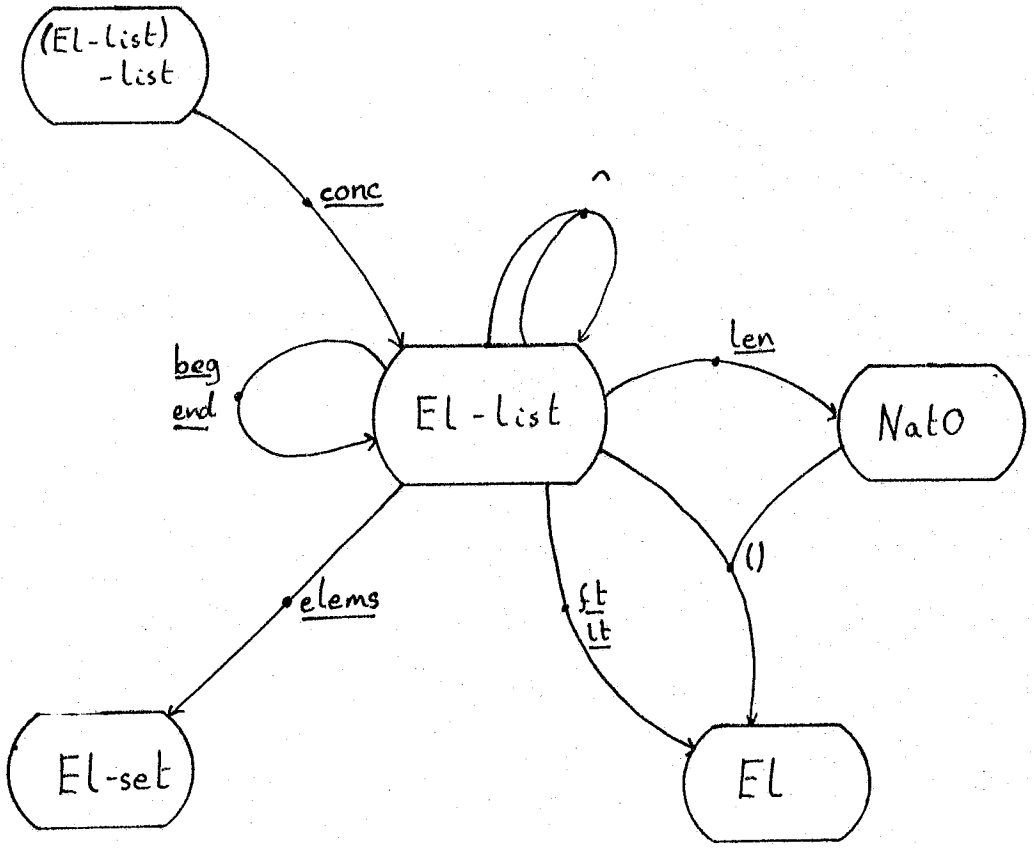
2.1.6 Lists

Lists (otherwise called sequences or tuples) are again restricted to being finite. A carrier on which the basic operators are defined is:

1 El-list $\subseteq$ M:Nat $\overset{m}{\rightarrow}$ El
inv ($\exists$ n $\in$ Nat0; dom m = $\{1 .. n\}$)

2 El-list1 = El-list - $\{<>\}$

The operators and their types are shown by:



Only two basic operators need be defined directly in terms of the model:

3 $\underline{\text{len}}\ l = \underline{\text{card}}\ \underline{\text{dom}}\ l$

4 for $1 \leqslant i \leqslant \underline{\text{len}}\ l$,
$l(i) = l(i)$

Note the left-hand expression denotes list indexing while the right-hand expression denotes application of a mapping.

Explicit lists will be written by listing the (not necessarily distinct) elements:

5 $\langle e1, e2, \ldots, en\rangle = \{1 \mapsto e1, 2 \mapsto e2, \ldots, n \mapsto en\}$

Other operators are defined:

6 inds l = {1..len l}

7 elems l = { l(i) st i∈ inds l}

8 ___ ^ ___ : L1:El-list x L2:El-list → RL:El-list
post len rl = len l1 + len l2 ∧
(∀ i ∈ inds l1; rl(i) = l1(i)) ∧
(∀ i∈ inds l2; rl(i + len l1) = l2(i))

Notice that the empty list is an identity for concatenation and that the operator is associative but is neither commutative nor absorptive.

(The idea to use the names "first"/"last" for elements and "beginning"/"ending"for lists was suggested by Ib Sorensen. The shorter abbreviations yield the elements.)

9 ft: El-list1 → El
ft l ≙ l(1)

10 lt: El-list1 → El
lt l ≙ l(len l)

11 beg: L:El-list1 → RL:El-list
post rl ^ <lt l> = l

12 end: L:El-list1 → RL:El-list
post <ft l> ^ rl = l

13 conc: (El-list)-list → El-list
conc ll ≙ if ll = <> then <> else (ft ll)^conc end ll

14 Given a well-founded order on El (<), the lexical order of (finite) lists is well-founded:

15 lexord: L1:El-list1 x L2:El-list1 $\xrightarrow{\sim}$ Bool

lexord (l1,l2) $\triangleq$
($\exists$ m $\in$ Nat; ($\forall$ n $\in$ Nat st n < m; l1(n) = l2(n)) $\wedge$
(m = len l1+1 $\vee$ l1(m) < l2(m)))

List operators have been defined and used in many papers (e.g. Reynolds /79a/, Dahl /78a/, Jones /80a/, Abrial /80b/). A distillation of the operators found in the literature is given.

16 subl: L:El-list x M:Nat0 x N:Nat0 $\xrightarrow{\sim}$ RL:El-list
pre m $\leq$ n + 1 $\leq$ len l + 1
post len rl = n-m + 1 $\wedge$
($\forall$ i $\in$ inds rl; rl(i) = l(i + m - 1))

17 isprefix: El-list x El-list $\rightarrow$ Bool
isprefix (l1,l2) $\triangleq$ len l1 $\leq$ len l2 $\wedge$
($\exists$ n $\in$ inds l2; l1 = subl(l2,1,n))

18 issuffix similar

19 issubstring: El-list x El-list $\rightarrow$ Bool
issubstring (l1,l2) $\triangleq$ ($\exists$ m,n $\in$ inds l2 st m $\leq$ n+ 1;
l1 = subl(l2,m,n))

20 del: L:El-list1 x N:Nat $\xrightarrow{\sim}$ El-list
pre n $\leq$ len l
del (l,n) $\triangleq$ subl(l,1,n-1) $\frown$ subl(l,n+1,len l)

21 modl: L:El-list1 x N:Nat x E:El $\xrightarrow{\sim}$ El-list1
pre n $\leq$ len l
modl(l,n,e) $\triangleq$ subl(l,1,n-1) $\frown$ <e> $\frown$ subl(l,n+1,len l)

22 indexmod: Nat x Nat $\rightarrow$ Nat
indexmod(m,n) $\triangleq$ ((m-1)mod n) + 1

23 rotate: L:El-list x N:Nat $\rightarrow$ RL:El-list
post len rl = len l $\wedge$
($\forall$ m $\in$ inds l; rl(m) = l(indexmod(m+n - 1, len l)))

24 rev: L:El-list $\rightarrow$ RL: El-list
post len rl = len l $\wedge$
($\forall$ m $\in$ inds l; rl(m) = l(len l+ 1 - m))

25 maskl: L:El-list x BL:Bool-list ~→ El-list
pre len bl = len l
maskl (l,bl) ≜ if l = <> then <>
else if ftbl then <ft l> ^ maskl (endl, endbl)
else maskl (endl, endbl)

26 issubseq: El-list x El-list → Bool
issubseq (l1,l2) = (∃ bl ∈ Bool-list st len bl = len l2;
l1 = maskl (l2,bl))

Implicit list definition was not allowed in Jones /80a/ because of the danger of giving a "definition" which failed to determine the order. Here, implicit lists of the form:

27 <e(i) st i ∈ {1..n}>

are considered to define:

28 {i ↦ e(i) st i ∈ {1..n}}

Thus:

29 <f(l(i)) st i∈ inds l>

30 < f(i) st i ∈ {1:n}>

31 < l(il(i)) st i∈ inds il >

with obvious constraints, are all allowed.

For sorting problems the following functions are useful:

32 isascending: Poset(El)-list → Bool
isascending (l) ≜ (∀ n ∈{1..len l-1}; l(i) ≤ l(i+1))

33 isuniquel(l) ≜ (∀ m,n ∈ inds l st m ≠ n; l(m) ≠ l(n))

34 bagol: L:El-list → B:Bag(El)
post bagdom(b) = elems l ∧
(∀ e∈ elems l;
mpcbagel(b,e) = card {n∈ inds l st l(n) = e})

35 ispermutation(l1,l2) $\triangleq$ bagol(l1) = bagol(l2)

Clearly, ispermutation is an equivalence relation on lists.

2.1.7 Covers

As a preliminary to the more specialised concept of a partition, a cover is defined as:

1 Cover(El) $\subseteq$ ss:(El-set)-set
inv union ss = El

Properties are:

2 $\{\{e\}$ st e $\in$ El$\}$ $\in$ Cover(El)

3 C $\in$ Cover(El) $\Rightarrow$
$(\{S \in C$ st p(S)$\}$ $\cup$ $\{$union $\{S \in C$ st $\sim p(S)\}\}) \in$ Cover(El)

4 C $\in$ Cover(El) $\wedge$ S $\in$ C $\wedge$ SP $\in$ Cover(S) $\Rightarrow$
$(\{S' \in C$ st $S' \neq S\}$ $\cup$ SP) $\in$ Cover(El)

2.1.8 Partitions

1 Partition(El) $\subseteq$ C:Cover(El)
inv $(\forall S1,S2 \in c$ st $S1 \neq S2;$ isdisj(S1,S2)) $\wedge$ $\{\} \notin c$

2 $\{\{e\}$ st e $\in$ El$\}$ $\in$ Partition(El)

3 P $\in$ Partition(El) $\Rightarrow$
$(\{S \in P$ st q (S)$\}$ $\cup$ $\{$union $\{S \in P$ st $\sim q$ (S)$\}\}) \in$ Partition(El)

2.1.9 Forests

The only "theory" of real interest presented here is that of forests (see Rings in Jones /80a/). The impetus for recording these

results came from a dissatisfaction with various proofs of the Fischer/Galler algorithm (cf. section 5.3): the proofs seemed to become clouded by results about the data structure which had nothing to do with the algorithm per se.

The idea of a "forest" is that a collection of trees over some set of objects (D) can be represented by a mapping in which "no loops" are allowed:

1 Forest(D) $\subseteq$ M: D $\xrightarrow[m]{\sim}$ D
inv iswellfounded(m)

(Notice that this representation, and the choice of results which follow, differ from those of Jones /80a/.)

Some useful functions are:

2 root: Forest(D) x D $\to$ D
root(f,d) $\triangleq$ if d $\notin$ dom f then d else root (f,f(d))

3 reach: Forest(D) x D $\to$ D-set
reach(f,d) $\triangleq$ if d $\notin$ dom f then {d} else {d} $\cup$ reach(f,f(d))

The root and reach functions are total because of the constraint that the mapping is well-founded.

4 coll: Forest(D) x D $\to$ D-set
coll (f,d) $\triangleq$ {e $\in$ D st d $\in$ reach(f,e)}

Useful properties of forests and their functions can be stated (use f,f' $\in$ Forest(D), c,d,e $\in$ D):

5 reach(f,d) $\subseteq$ coll(f,root(f,d))
follows from:
i coll(f,root(f,d)) = {e $\in$ dom f st root(f,d) $\in$ reach(f,e)}
ii root(f,d) $\in$ reach(f,d)

6 for r1,r2 $\in$ (rng f - dom f):
r1 = r2 $\vee$ isdisj(coll(f,r1),coll(f,r2))

7 $d \notin coll(f,e) \Rightarrow (f \dagger \{e \mapsto d\}) \in Forest(D)$

8 $f' \lceil reach(f,e) = f \lceil reach(f,e) \Rightarrow root(f',e) = root(f,e)$

9 $(\forall e \in coll(f,d);\ root(f \dagger \{d \mapsto c\},e) = root(f,c))$

2.1.10 Constructed Objects

In addition to collections like lists and sets, there is a need for inhomogeneous aggregates. In programming languages this need is met with records or structures; here a means of defining "constructed objects" is introduced. Such sets of objects are defined by an abstract syntax rule with a "::" as definition symbol. In terms of sets, this can be interpreted as implying a constructor function:

1 A::B C

2 $A = \{mk\text{-}A(b,c)$ st $b \in B \wedge c \in C\}$

3 mk-A:B x C $\rightarrow$ A

The only properties required of constructor functions is that they are unique:

4 $mk\text{-}A(x) = mk\text{-}B(y) \Leftrightarrow A=B \wedge x = y$

The refusal to give a more concrete interpretation to constructor functions makes them rather like pointers in a capability architecture (Needham /72a/): objects can only be created by the appropriate constructor. Such constructors will also be used in an obvious way as the clauses of a case construct (e.g. 2.2.4(9,10,12)).

Also, analogous to records in a programming language, selectors can be introduced as a way of accessing components of a constructed object: thus for:

5 A::S1:B S2:C

6 S1(mk-A(b,c)) = b

7 S2(mk-A(b,c)) = c

Notice that the selectors are considered to be functions, for example:

8 S1: A → B

The approach taken in "Z" (Abrial /80a/) is to view the objects themselves as mappings from the selector names to the values of their components. Although this view has some advantages in combining objects, such a denotation is not sufficiently unique for some purposes. There have, in fact, been many attempts to resolve the problem of making classes of objects distinct in spite of similar content. In the "ULD" (using what became known as "VDL" - Walk /69a/) explicit flag components had to be inserted in the "abstract" syntax; ECMA /76a/ abstract syntax trees had the names of all rules present in the tree - this made some tree operations very clumsy; the normal technique in denotational semantics (cf. Stoy /77a/) is to make the union operator responsible for introducing disjointness.

Chapter 14 of Jones /80a/ defines an obvious abstract syntax notation for building up sets of objects. Although recursive rules are permitted, any object satisfying such a definition is assumed to be finite. It is, therefore, permissible to use structural induction (cf. Burstall /69a/) on such objects.

2.2 Data Refinement Proofs

As well as introducing and justifying the proof rules to be used for data refinement (sub-section 2.2.1), a number of practical points are discussed in the remaining sub-sections.

2.2.1 Proof Rules

Much of the material on proofs of data refinement is covered in chapter 1 in developing the notion of "bias" which is used as a criterion for a "specification".

It is shown in sub-section 1.3.3 that, for total REL_{OP}, model satisfaction can be justified by showing that
REL^{-1} ; OP2; REL sat OP1 holds for each pair of representation and abstract operations. Furthermore, section 1.3.4 shows the advantage of using a representation relation:

1 retr: Rep $\rightarrow$ Abs

which is a total surjection.[+] Such functions are referred to, following Jones /80a/, as "retrieve functions". (Hoare /72b/ might have called the λ functions "abstraction functions" but in fact used the term "representation functions"; Reynolds /81a/ uses the term "invariants"; Milner /71a/, rather drily, uses "homomorphism".)

The relational form of the proof rule used below is:

2 $retr^{-1}$; R; retr sat A

if the following conditions hold:

[+] cf. sub-section 2.2.3.

(DOMAIN) R defover dom (retr; A)

(RESULT) dom (retr; A); R psat retr; A; $retr^{-1}$

To justify this proof rule:

i	R defover dom (retr; A)	DOMAIN
ii	$retr^{-1}$; R defover dom ($retr^{-1}$; retr; A)	i,$retr^{-1}$ total, 1.2(4)
iii	$retr^{-1}$; retr = E	1
iv	$retr^{-1}$; retr; A defover dom A	iii
v	$retr^{-1}$; R defover dom A	iv,ii,defover
vi	$retr^{-1}$; R; retr defover dom A	v,1,1.2(3)
vii	(dom retr);A ; R psat retr; A; $retr^{-1}$	RESULT
viii	(dom A); $retr^{-1}$; R; retr psat $retr^{-1}$; retr; A; $retr^{-1}$; retr	vii,1.2(7)
ix	$retr^{-1}$; retr = E	iii,retr is onto
x	(dom A); $retr^{-1}$; R; retr psat A	viii,ix,psat
	$\therefore$ $retr^{-1}$; R; retr sat A	vi,x

In actual proofs, it is more convenient to use a predicate version of 2. This rule is named for future reference, as are its conditions. The condition that the retrieve function be a surjection is stated explicitly (ADEQUACY) because pragmatically it is a very useful check to apply to a representation and even becomes a guide to design.

REFINE retr: Rep $\rightarrow$ Abs

(ADEQUACY) $(\forall a \in Abs;\ (\exists r \in Rep;\ a = retr(r))$

(DOMAIN) $(\forall r \in Rep;$ pre A(retr(r)) $\Rightarrow$ pre R(r))

(RESULT) $(\forall r,r' \in R\ ep;$ preA(retr(r)) $\wedge$ postR(r,r') $\Rightarrow$ postA(retr(r),retr(r)))

Notice that the inverse of the retrieve function no longer appears in the conditions. This makes the rule significantly easier to use. This rule can be seen as a consequence of 1.4 (WKNPRE/STRPOST).

As an example of a refinement proof, consider the problem of keeping track of two disjoint sets (this is motivated in Jones /80a/ as the "students who do exercises" problem).

4 $St = Id \xrightarrow[m]{\sim} Bool$

5 INIT
globals S: wr St
post s' = {}

6 ENTER (N:Id)
globals S: wr St
pre $n \notin \underline{dom}\ s$
post $s' = s \cup \{n \mapsto \underline{FALSE}\}$

7 MOVE (N:Id)
globals S: wr St
pre $s{:}n \mapsto \underline{FALSE}$
post $s' = s \dagger \{n \mapsto \underline{TRUE}\}$

8 QUERY ( ) return RES:Id-set
globals S: rd St
post $res' = \{n \in \underline{dom}\ s\ \underline{st}\ s(n)\}$

A representation might be chosen which uses two separate lists to record the Ids:

9 St2 :: NL: Id-list YL: Id-list
inv isdisj (elems nl, elems yl)

The retrieve function would link these two representations:

10 retr: St2 → St
$retr(mk\text{-}St2\ (nl,yl)) \triangleq \{id \mapsto \underline{FALSE}\ \underline{st}\ id \in \underline{elems}\ nl\} \cup \{id \mapsto \underline{TRUE}\ \underline{st}\ id \in \underline{elems}\ yl\}$

Notice that 10 is only total because of the invariant on 9, otherwise the use of map union (cf. 2.1.4(4)) might be undefined. The ADEQUACY of the representation is obvious and can be proved formally by induction on the domain of dom s. The adequacy property establishes that there is at least one representation for any abstract state. It is clear that, since order is not constrained, there are many representations for each element of St. However, the retrieve function can be seen as inducing an equivalence relation on the representation, and the purpose of the RESULT rule is to show that operations on the representation respect these equivalences. The operations on the representation (9) are:

11 INIT2
globals NL: wr Id-list YL: wr Id-list
post nl' = yl' = <>

12 ENTER2 (N: Id)
globals NL: wr Id-list YL: rd Id-list
pre $n \notin (\underline{elems}\ nl \cup \underline{elems}\ yl)$
post $nl' = nl \frown \langle n \rangle$

13 MOVE2 (N: Id)
globals NL: wr Id-list YL: wr Id-list
pre $n \in \underline{elems}\ nl$
post $(\exists i \in \underline{inds}\ nl;\ nl(i) = n \wedge nl' = del(nl,i) \wedge yl' = yl \frown \langle n \rangle)$

14 QUERY2( ) return RES: Id-set
globals YL: rd Id-list
post res' = elems yl

The most interesting of these operations to prove correct is MOVE2. The DOMAIN condition of REFINE becomes:

15 $(\forall st2 \in St2;\ preMOVE(retr(st2),n) \Rightarrow pre\text{-}MOVE2(st2,n))$

which simplifies to:

i isdisj(nl,yl) ∧ n ∈ elems nl ⇒ n ∈ elems nl

The RESULT rule for this instance is:

16 (∀ st2,st2' ∈ St2; preMOVE(retr(st2),n) ∧ postMOVE2(st2,n,st2') ⇒
post-MOVE (retr(st2),n,st2'))

which simplifies to:

i n ∈ elems nl ∧ (∃ i ∈ inds nl; nl(i) = n ∧ nl' = del(nl,i) ∧ yl' = yl^<n> ⇒
retr(mk-St2(nl',yl')) =
retr(mk-St2(nl,yl)) † { n ↦ TRUE }

2.2.2 Naming

The first of the practical points concerning data refinement proofs is the naming of stages of development. The usage of the previous sub-section is the basis of a general scheme which covers the possibility of alternative designs for a single specification.

Given a specification (SPEC), its alternative refinements are named SPEC.1, SPEC.2, etc. In the design of larger systems, it will be necessary to use more than one level of data refinement. This will give rise, in an obvious way, to stages named SPEC.2.1.2 etc. The name of the state will be suffixed in a similar way.

Notice that when an operation is decomposed (cf. chapter 3), the sub-operations will have new names which do not have to carry the entire numbering scheme with them.

2.2.3 On Adequacy

The condition concerning adequacy in 2.2.1(REFINE) has been found to be very useful in design. In particular, it is a check

which can be applied (once) to the representation rather than a property of each operation. It has, however, been pointed out by Lockwood Morris that the condition is not necessary. Consider an implementation of the example in sub-section 2.2.1 based on:

1 St1:: NS: Id-set YS: Id-set

and assume that no data type invariant is recorded. Two problems would arise. Firstly the retrieve function:

2 retr: St1 → St

would not be total. Secondly, if a subsequent design stage were to use a representation:

3 St11:: M: Id $\xrightarrow[m]{\sim}$ Nat BL: Bool-list

(or, indeed, if St were used as a representation!) it could not be shown to be adequate. These two problems are similar and discussion is focussed on the second. The "difficulty" is that the objects in 1 in which the sets are not disjoint cannot be represented by 3 (or 2.2.1(4)): such objects would, however, never arise if the operations were defined in the obvious way. The disadvantage of this form of reasoning is that it is forced to rely on the whole set of operations. Where practical, it appears to be worthwhile to restrict the class of objects with a data type invariant and to tackle adequacy as a separate issue.

2.2.4 Inheriting Invariants

A related point to that in the previous sub-section is the observation that an invariant on one level is likely to reappear in a

more detailed representation. Thus, if 2.2.3(1) is completed with the invariant:

1 St1:: NS: Id-set YS: Id-set
inv isdisj(ns,ys)

and 2.2.1(9) is presented as a representation of 1:

2 St11:: NL: Id-list YL: Id-list
inv isdisj (elems nl, elems yl) ∧
isuniquel(nl) ∧ isuniquel(yl)

then the disjointness part of the invariant has been stated at both levels. It might be desirable to state the invariant of 2 as:

3 invSt1(retr(st11)) ∧ ...

The preservation of the inherited part of the invariant then follows from the 2.2.1(REFINE(RESULT)) conditions.

The incentive to factor an invariant into inherited and new parts is much greater on an example like that of Fielding /80a/. Modifying the representations slightly, the development of B-trees (of order m) can be explained in terms of:

4 Btree = Key $\tilde{m}\rightarrow$ Data

5 Btree1 ⊆ N:Node
inv rti(n)

6 Node = Inode | Tnode

7 Inode:: NL: Node-list
inv keysetsord(nl) ∧ balanced(nl) ∧ len nl ≤ 2 * m + 1

8 Tnode::DM: Key $\tilde{m}\rightarrow$ Data
inv card domdm ≤ 2 * m

9 rti (n) $\triangleq$ cases n:
mk-Inode(nl) → 2 $\leq$ len nl $\wedge$
($\forall$ sn $\in$ elems nl; szi(sn))
mk-Tnode(mn) → TRUE

10 szi(n) $\triangleq$ cases n:
mk-Inode(nl) → m + 1 $\leq$ len nl $\wedge$
($\forall$ sn $\in$ elems nl; szi(sn))
mk-Tnode(nm) → m $\leq$ card dom nm

11 keysetsord (nl) $\triangleq$
($\forall$ i $\in$ {1..len nl-1}; collkeys(nl(i)) $\ll$ collkeys(nl(i + 1)))

12 collkeys(n) $\triangleq$ cases n:
mk-Inode(nl) → union {collkeys(nl(i)) st i $\in$ inds nl}
mk-Tnode(nm) → dom nm

13 ks $\ll$ ks' $\Leftrightarrow$ ($\forall$ k $\in$ ks; ($\forall$ k' $\in$ ks'; k < k'))

14 balanced(nl) $\triangleq$ card {depth(nl(i)) st i $\in$ inds nl} = 1

15 depth(n) $\triangleq$ cases n:
mk-Inode(nl) → union {depth(nl(i)) st i $\in$ inds nl} ++ 1
mk-Tnode(nm) → {1}

16 retr: Btree1 → Btree
retr(n) $\triangleq$ cases n
mk-Inode(nl) → munion {retr(nl(i)) st i $\in$ inds nl}
mk-Tnode(nm) → nm

17 munion: (Key $\xrightarrow[m]{\sim}$ Data)-set $\xrightarrow[m]{}$ (Key $\xrightarrow[m]{\sim}$ Data)

17A ++: Int-set x Int → Int-set

As is shown in the referenced monograph, many of the design issues of insertion and deletion can be addressed with the representation of 5. Eventually, however, the keys which steer the search at an intermediate node must be brought in. Thus

18 Btree2 $\subseteq$ Node 2

19 Node 2 = Inode 2 | Tnode

20 Inode2:: KEYL: Key-list TREEL: Node 2-list

inv len keyl = len treel-1 ∧ isascending (keyl) ∧
(∀ i ∈ inds keyl; collkeys(treel(i)) ≪ {keyl(i)} ≪
collkeys(treel(i + 1)))

This part of the invariant can only be stated on this representation because it only has meaning once the KEYL is present. The remaining part of the invariant is, however, most easily stated by:

21 invBtree1(retr 1(t2))

where:

22 retr1: Btree2 → Btree1

is an obvious function which discards the key list.

In practice, the aims of a development method may only be met by expanding the invariant at each stage of refinement: the designer of level n + 1 has enough to cope with in considering the specification at level n without being asked to search back through earlier levels.

2.2.5 On Design

Sub-section 1.3.4 offers some criteria for judging specifications. What can usefully be said about design steps? It must be appreciated that the current work offers a notation for recording and justifying a chosen design but does not claim to be in any way normative on design decisions (as, for example, does Jackson /75a/ and Jackson /80a/).

The B-tree example of the last sub-section shows how it is precisely in design that bias and complex data type invariants are introduced. This is quite acceptable. The abstract (external) behaviour is being met on a more intricate representation. Bias is introduced where it is too expensive to ensure that a set of alternative representations are always reduced to a canonical element. Data type invariants arise precisely because of the move away from the "ideal abstraction" to a (too) general data structure.

There must be a balance in choosing steps of design between their number and complexity. Normally, it pays to only bring in one design decision in a step. Thus, in the B-tree example, the concept of trees is brought in before the problem of how to choose a path is resolved. This leads to an odd description, in the first step, of the choice of path: collecting all of the keys in each sub-tree is not a good implementation but is quite acceptable as a specification. Not only does this remove some of the technical details from what is already a major design step, but it also opens up a range of choices for choosing paths (lengths of key, key compression etc.).

Another problem which has been encountered in using data refinement on large examples is how to handle the introduction of a redundant representation. There may be good reasons in a design for having two entirely equivalent representations of the same information. This may be for efficiency or recovery purposes. It appears to be clearer if this redundancy is brought in at separate steps of the design.

2.2.6 A (Potential) Problem

The picture created in this section is of data refinement proofs being conducted on a component basis. Sub-section 1.3.3 has shown that if modelling works on a component basis, then any composition will also model the composition on the abstract level. Unfortunately, this attractive picture could fail!

Suppose OPA and OPB are specified in terms of sets and their sequential composition has been shown to satisfy some overall specification. If a representation is chosen in terms of lists it is possible for OPB1 to have a pre-condition requiring that some list be ordered; providing the post-condition of OPA1 ensures this order, the composition of OPA1 and OPB1 is valid. The problem is that, assuming order is not an invariant of the representation, the correctness no longer follows entirely from a local argument on the components. To reprove the validity of the composition is not too arduous and, in fact, this problem hardly ever occurs with sequential proofs. An exactly analogous problem is, however, encountered in a design involving parallelism in sub-section 5.2.8.

2.3 General Refinement

One of the reasons for developing theories of data types is to ensure that a body of (engineering) documentation is developed as a result of the use of rigorous methods on individual projects. It would also be highly desirable to collect results about steps of data refinement in a way which made them usable by other projects. No fully worked out proposal is offered here but a number of criteria are identified.

A very simple example of a "general refinement" could be based on a specification with two lists:

1 St:: L1: El-list L2: El-list

This can be represented in an obvious way by a single list structure with pointers:

2 St1:: CL: Pr-list P1: [Nat] P2: [Nat]

3 Pr:: CONT: El PTR: [Nat]

with a data type invariant about loops etc. and an obvious retrieve function. A parameter concept like that in "Z" should be able to document such a refinement in a way which makes it reusable. A similar problem is the refinement of stacks (of elements of dissimilar size) onto linear storage. The situation, however, already becomes notationally more difficult if an arbitrary number of structures (e.g. lists) are to be represented.

A good test example for the ability to rely only on the essential parts of a representation is the problem of refinement for general trees. In Fielding /80a/, both very simple binary trees (with data in the intermediate nodes) and full B-trees (with all data in terminal nodes - cf. 2.2.4(4-22)) have to be represented in linear storage. The same basic problems must be addressed in both refinements but the differences in node structure make this general refinement difficult to separate out.

An extreme case of the general problem of splitting up a development can be found in the work on compilers. The objective of handling compiler proofs by tackling one "language concept" at a time was stated clearly by the Vienna group in the late 1960's. Such division has, however, never been fully formalised. Some work in this direction is described in Mosses /77a/, Morris /73a/ and an inspiring early recognition of similar problems is given in Cooper /66a/.

See also Ehrig /81a/.

2.4 Alternatives

The issues surrounding the specification of abstract data types are covered in section 1.6. In order to reify model oriented specifications, the concept of using, what is called here, a "retrieve function" is generally accepted (Milner /71a/, Hoare /72b/, Jones /72a/). If a "biased" representation has been given as a model then one possible proof method is to use the rule outlined in sub-section 1.3.3 where the retrieve function becomes a relation. Reynolds /81a/ brings the specification and representation together with an "invariant". Another approach to this situation is to use "ghost variables". The idea (cf. Lucas /68a/) is to reconstruct the (biased) specification objects in the representation and thus to make it possible to define a retrieve function. After a normal refinement proof has been given, the "ghost variables" can be discarded subject to certain obvious rules.

In some sense, the need to use ghost variables is a symptom of a poorly chosen specification. Looking ahead to the problems of parallelism, this prompts the question of whether the use of "ghost variables" cannot also be avoided there by finding an appropriate abstraction.

One other point of comparison with alternative approaches is worth making. When a specification of an operation is given by post-condition, it is often non-constructive and is in no way intended to be translatable into an implementation. In spite of the fact that they are still dubbed "specifications", there is a tendency to regard

abstract data types as providing implementations in some very high level language (e.g. Schonberg /81a/). This would appear to be a very dangerous interpretation of model oriented specifications: no clever selection of set representations would make the (set based) specification of sub-section 5.3.1 an efficient implementation.

Chapter 3

Decomposition for Isolated Programs

The view of specifications as relations is introduced in sub-section 1.1.2. This provides a natural way of treating under-determined, partial operations. The concept of state is taken to be central to that of operation because of the ubiquity of the assign-ment statement. In this chapter the combinators which are used to construct useful programs from simpler statements are studied.

Parts of this chapter follow Jones /80a/ very closely. The proof rules of sub-section 3.2.2 include, with some revisions, those in the earlier book. There are, in addition, some new rules (e.g. IWHILEFIX, FORARB) which have been prompted by the work on parallelism. Because of the basic similarity, only small examples of the use of these rules are given here (section 3.4). It is, of course, only on more taxing problems that the decision to use post-conditions of two states can really be justified (see discussion in section 3.5).

A much larger departure from Jones /80a/ has been made in the semantic definition (section 3.1) used for justification of the proof rules. A denotational semantics based on relations (cf. Hitchcock /72a/, Hitchcock /74a/, Park /80a/) is used to cope with the problem of non-determinacy. The relational form of the proof rules (cf. sub-section 3.2.1) is proved valid with respect to denotational semantics in section 3.3 (cf. Lauer /71a/ and Donahue /75a/).

Section 3.2 presents the rules about programming language constructs as proof rules; in section 3.4 and in larger applications the same rules are used as an integral part of a development method.

In this mode the combinators appear with specifications as their sub-components. This usage is justified in theorem 3.1.2(12) by showing that the combinators are monotone with respect to the sat ordering. Decomposition proofs are used later than data refinement in the development cycle. For this reason the proof rules tend to be used as check-lists rather than for formal symbol manipulation.

3.1 Sequential Programming Language

The language whose semantics is given below does not include any notion of parallelism but does permit non-determinism.

3.1.1 Relational Semantics

Relations have been used because of the desire to incompletely specify operations. Similarly relations are chosen here as the denotations for non-deterministic programs. The basic non-determinism is inherited from the specifications but a "guarded if" (cf. Dijkstra /75a/) is included to illustrate how non-determinism in the language itself can be handled. The guarded conditional is defined because it can be used as a basis for the more conventional conditional statements; these latter are also provided with proof rules in section 3.2. Although there is no difficulty in defining the semantics of a guarded while statement, this is not done because the incentive is weaker.

As observed in Jones /73a/, some care is required in using relations as denotations. Suppose two non-deterministic operations are denoted by:

1 $R1 = \{a \leftrightarrow m,\ b \leftrightarrow m,\ b \leftrightarrow n\}$
$R2 = \{m \leftrightarrow z\}$

Then using the definition of relational composition (1.1(23)):

2 $R1; R2 = \{a \leftrightarrow z,\ b \leftrightarrow z\}$

If this were to be defined as the meaning of sequential operation composition, the language would be impossible to implement. It is

therefore necessary to avoid the need for look-ahead and to define the semantics such that the denotation for the composition of the operations in 1 is:

3 $\{a \leftrightarrow z\}$

Another facet of the same problem is the need to distinguish between:

4 $\{a \leftrightarrow z,\ a \leftrightarrow \perp\}$ and $\{a \leftrightarrow z\}$

Here, Park /80a/ is followed in giving two denotation functions - one which yields the relation (M) and one which yields the set over which termination is assured (T).

In other places (Bekić /74a/, Bjørner /78a/), it has been proposed that a denotational semantics should be defined over the abstract syntax of a language. Because the language to be defined here is so small, the definition will use the concrete syntax.

5 Stmt ::= Composition | Guardedif | While | Basic
Composition ::= Stmt; Stmt
Guardedif ::= if Expr → Stmt [] ... []Expr → Stmt fi
While ::= while Expr loop Stmt endloop
Basic, Expr not further defined.

The semantic domains to be considered are:

6 $Tr \subseteq \wp(St \times St)$

7 $Trv : St \xrightarrow{\sim} Val$

The types of the semantic functions are:

8 $M[\![\ ]\!] : Stmt \rightarrow Tr$
$M[\![\ ]\!] : Expr \rightarrow Trv$

9 $T[\![\ ]\!] : Stmt \to \wp(St)$
$T[\![\ ]\!] : Expr \to \wp(St)$

Notice that the semantic functions can be applied to either statements or expressions. Expressions may be partial but their evaluation is assumed to produce a deterministic result. The possibility of an expression being undefined is very important in the presence of arrays etc. (It is likely that indexing outside the range of an array occurs more often in the condition than in the body of a while statement.)

The sets of states for which expressions evaluate to true are defined as:

10 $\hat{e} = \{s \in T[\![e]\!] \ \underline{st} \ M[\![e]\!] : s \mapsto \underline{TRUE}\}$

11 $\hat{\sim e} = T[\![e]\!] - \hat{e}$

Clearly:

12 $isdisj(\hat{e}, \hat{\sim e})$

The denotations of the various statements in 5 are now defined. For simple composition:

13 $M[\![P1; P2]\!] = M[\![P1]\!]; M[\![P2]\!]$

14 $T[\![P1; P2]\!] = \{s \in T[\![P1]\!] \ \underline{st} \ (\forall s' \ \underline{st} \ M[\![P1]\!] : s \leftrightarrow s'; s' \in T[\![P2]\!])\}$

The problem displayed in 1 to 3 being solved by the use of T. Notice that ";" is associative.

For the guarded conditional:

15 $IF = \underline{if}\ e1 \to P1 \ [] \ldots [] \ en \to Pn \ \underline{fi}$

the semantics are:

16 $M[\![IF]\!] = \underline{union}\ \{\hat{ei};\ M[\![Pi]\!]\ \underline{st}\ 1 \leqslant i \leqslant n\}$

17 $T[\![IF]\!] = \{s \in te\ \underline{st}\ (\forall i\ \underline{st}\ s \in \hat{ei};\ s \in T[\![Pi]\!])\}$

where $te = \underline{int}\{T[\![ei]\!]\ \underline{st}\ 1 \leqslant i \leqslant n\} \cap \underline{union}\{\hat{ei}\ \underline{st}\ 1 \leqslant i \leqslant n\}$

Non-determinism is introduced whenever the $\hat{ei}$ are non-disjoint. Termination can only be guaranteed for states in which there is no ei whose evaluation can abort and for which some ei evaluates to TRUE.

For the iterative construct:

18 WH = while e loop B endloop

the semantics are:

19 $M[\![WH]\!] = fix\ (\lambda R.\ E_{\hat{\sim e}} \cup \hat{e};\ M[\![B]\!];\ R)$

20 $T[\![WH]\!] = fix\ (\lambda S.\ \hat{\sim e} \cup \{s \in (\hat{e} \cap T[\![B]\!])\ \underline{st}\ (\forall s'\ \underline{st}\ M[\![B]\!] : s \leftrightarrow s';\ s' \in S)\})$

The fixed point definition of M is straightforward enough. The set over which iteration is guaranteed to terminate must, however, ensure that states are admitted only in the case that all of their successor states (under $M[\![B]\!]$) will terminate. This universal quantification makes 20 non-ω continuous. For a simple (non-deterministic) case, it is easy to see that:

21 $T[\![\ \underline{while}\ X \neq 0\ \underline{loop}\ X :\in \{0\ ..\ X-1\}\ \underline{endloop}\ ]\!] = Nat0$

But, so far, nothing has been done to prohibit "unbounded non-determinism" (cf. Dijkstra /76a/). With:

22 WH = while X ≠ 0 loop IF endloop

23 IF = if X < 0 → X:∈ Nat0 [] X>0 → X:=X-1 fi

Then:

i $M[\![IF]\!] = \{x \leftrightarrow x' \ \underline{st}\ x < 0 \wedge x' \geqslant 0 \vee x > 0 \wedge x' = x-1\}$

ii $T[\![IF]\!] = Int - \{0\}$

iii $T[\![WH]\!] = fix\ (\lambda S.\ \mathcal{F}(S))$

iv $\mathcal{F}(S) = \{0\} \cup \{x \ \underline{st}\ x \neq 0 \wedge (\forall x' \ \underline{st}\ M[\![IF]\!] : x \leftrightarrow x';\ x' \in S)\}$

While one fixed point is:

v $\mathcal{F}(Int) = Int$

the limit is:

vi $\bigcup_{n=0}^{\infty} \mathcal{F}^n(\{\}) = Nat0$

If, however, Park /80a/ is followed (into the transfinite):

vii $\mathcal{F}^{\omega}(\{\}) = Nat0$

viii $\mathcal{F}^{\omega+1}(\{\}) = \mathcal{F}(\mathcal{F}^{\omega}(\{\})) = Nat0 \cup \{i \in Int \ \underline{st}\ i < 0\}$
$= Int$

(Park /80a/ also shows the connection with the problem of "fairness".)

There are then two possibilities. Either non-determinism should be confined to the bounded case (cf. discussion in sub-section 3.3.4) or the limit of 20 should be computed by the transfinite union. For the current purposes this issue can be left open. The proof rules of section 3.2 were designed under the former assumption. David Park has, however, pointed out that the well-founded relation VAR could be defined over the transfinite ordinals, in which case unbounded non-determinism could be handled.

For the basic statements it is necessary (cf. 3.1.2(5)) to assume:

24 $(\forall P \in \text{Basic};\ T[\![P]\!] \subseteq \underline{\text{dom}}\ M[\![P]\!])$

The non-deterministic assignment in 21 being an example of a basic statement.

3.1.2 Properties

The definition of sat in 1.2(10) can now be restated:

1 $$[\![PR]\!]\ \underline{\text{sat}}\ [\![P]\!] \Leftrightarrow (M[\![PR]\!]\ \underline{\text{defover}}\ T[\![P]\!] \wedge T[\![P]\!];\ M[\![PR]\!]\ \underline{\text{psat}}\ M[\![P]\!])$$

The main result of this sub-section is to show that the program combinators in 13 to 24 are monotone with respect to sat. Since program design will use sat as the touchstone of correctness, this result justifies the placing of specifications in combinators. This appears to be a highly intuitive justification for the use of an abstract notion like monotonicity.

As a first step, lemmas are given which show that the termination set is always a subset of the domain of the meaning relation.

2 $$M[\![P1]\!]\ \underline{\text{defover}}\ T[\![P1]\!] \wedge M[\![P2]\!]\ \underline{\text{defover}}\ T[\![P2]\!] \Rightarrow M[\![P1;\ P2]\!]\ \underline{\text{defover}}\ T[\![P1;\ P2]\!]$$

This lemma can be shown to be true by:

i $M[\![P1;\ P2]\!] = M[\![P1]\!];\ M[\![P2]\!]$ 3.1.1(13)

ii $\underline{\text{dom}}\ M[\![P1;\ P2]\!] =$
$\{s \in \underline{\text{dom}}\ M[\![P1]\!]\ \underline{\text{st}}\ (\exists s'\ \underline{\text{st}}\ M[\![P1]\!] : s \leftrightarrow s';\ s' \in \underline{\text{dom}}\ M[\![P2]\!])\}$ i

iii $T[\![P1; P2]\!] =$
$\{s \in T[\![P1]\!]$ st $(\forall s'$ st $M[\![P1]\!] : s \leftrightarrow s';\ s' \in T[\![P2]\!])$ 3.1.1(14)

iv $s \in T[\![P1]\!] \Rightarrow s \in \underline{dom}\ M[\![P1]\!]$ hyp.

v $s \in T[\![P1]\!] \wedge (\forall s'$ st $M[\![P1]\!] : s \leftrightarrow s';\ s' \in T[\![P2]\!]) \Rightarrow$
$(\exists s'$ st $M[\![P1]\!] : s \leftrightarrow s';\ s' \in T[\![P2]\!])$ iv

vi $T[\![P1; P2]\!] \subseteq \{s \in T[\![P1]\!]$ st $(\exists s'$ st $M[\![P1]\!] : s \leftrightarrow s';\ s' \in T[\![P2]\!])\}$
iii,v

vii $\subseteq \{s \in \underline{dom}\ M[\![P1]\!]$ st $(\exists s'$ st $M[\![P1]\!] : s \leftrightarrow s';\ s' \in \underline{dom}\ M[\![P2]\!])\}$
hyps.

viii $\subseteq$ $\underline{dom}\ M[\![P1; P2]\!]$ ii

The result for conditionals is similar:

3 $\bigwedge_i (M[\![Pi]\!]$ defover $T[\![Pi]\!]) \Rightarrow M[\![IF]\!]$ defover $T[\![IF]\!]$

The corresponding lemma for the iterative construct requires an inductive proof.

4 $M[\![B]\!]$ defover $T[\![B]\!] \Rightarrow M[\![WH]\!]$ defover $T[\![WH]\!]$

let:

i $\mathcal{F}(S) = \hat{\sim e} \cup \{s \in \hat{e}$ st $(\exists s';\ M[\![B]\!] :\ s \leftrightarrow s' \wedge s' \in S)\}$

ii $\mathcal{F}(\{\}) = \hat{\sim e}$ i

iii $\mathcal{F}^{n+1}(\{\}) = \{s \in \hat{e}$ st $(\exists s';\ M[\![B]\!] : s \leftrightarrow s' \wedge s' \in \mathcal{F}^n(\{\}))\}$ i

let:

iv $\mathcal{G}(R) = E_{\hat{\sim e}} \cup \hat{e};\ M[\![B]\!];\ R$

v $\mathcal{G}(\Omega) = E_{\hat{\sim e}}$ iv

vi $\mathcal{G}^{n+1}(\Omega) = \hat{e};\ M[\![B]\!];\ \mathcal{G}^n(\Omega)$ iv

by induction on n, dom is monotone:

vii $\bigcup_{n=0}^{\infty} \mathcal{F}^n(\{\}) \subseteq \underline{dom} \bigcup_{n=0}^{\infty} \mathcal{G}^n(\Omega)$ i-vi

Now:

viii $T[\![WH]\!] =$
$\quad fix(\lambda S.\ \hat{\sim e} \cup \{s \in (\hat{e} \cap T[\![B]\!])\ \underline{st}$
$\quad\quad (\forall s'\ \underline{st}\ M[\![B]\!]: s \leftrightarrow s';\ s' \in S)\})$ 3.1.1(20)

ix $s \in T[\![B]\!] \Rightarrow s \in \underline{dom}\ M[\![B]\!]$ hyp.

x $s \in T[\![B]\!] \wedge (\forall s'\ \underline{st}\ M[\![B]\!]: s \leftrightarrow s';\ s' \in S) \Rightarrow$
$\quad (\exists s'\ \underline{st}\ M[\![B]\!]: s \leftrightarrow s';\ s' \in S)$ ix

xi $T[\![WH]\!] \subseteq fix(\lambda S.\ \hat{\sim e} \cup \{s \in (\hat{e} \cap T[\![B]\!])\ \underline{st}$
$\quad\quad (\exists s';\ M[\![B]\!]: s \leftrightarrow s' \wedge s' \in S)\})$ viii,x

xii $\subseteq fix(\lambda S.\ \hat{\sim e} \cup \{s \in \hat{e}\ \underline{st}$
$\quad\quad (\exists s';\ M[\![B]\!]: s \leftrightarrow s' \wedge s' \in S)\})$ ix,xi

Further, since $\mathcal{F}$ is monotone and ω continuous:

xiii $T[\![WH]\!] \subseteq \bigcup_{n=0}^{\infty} \mathcal{F}^n(\{\})$ xii,i

xiv $\subseteq \underline{dom} \bigcup_{n=0}^{\infty} g^n(\Omega)$ vii

xv $\subseteq \underline{dom}\ (fix\ \lambda R.\ E_{\hat{\sim e}} \cup \hat{e};\ M[\![B]\!];\ R)$ iv

The theorem which collects the results of lemmas 2, 3, 4 and 3.1.1(24) can now be proved by straightforward structural induction over 3.1.1(5).

5 for $P \in Stmt$, $M[\![P]\!]$ <u>defover</u> $T[\![P]\!]$

The next task is to show that the language combinators are monotonic with respect to T. In the remainder of this section, the use of "B", "BR", "WH", "WHR" etc. will be assumed to be obvious substitutions in 3.1.1(18) etc. The proofs of the first two lemmas are straightforward:

6 $\bigwedge_i [\![PiR]\!]$ <u>sat</u> $[\![Pi]\!] \Rightarrow T[\![P1;\ P2]\!] \subseteq T[\![P1R;\ P2R]\!]$

7 $(\bigwedge_i [\![PiR]\!]$ <u>sat</u> $[\![Pi]\!]) \Rightarrow T[\![IF]\!] \subseteq T[\![IFR]\!]$

For the iterative construct:

8 $[\![BR]\!]$ sat $[\![B]\!] \Rightarrow T[\![WH]\!] \subseteq T[\![WHR]\!]$

i $T[\![WH]\!] = fix(\lambda S.\ \hat{\sim e} \cup \{s \in (\hat{e} \cap T[\![B]\!])$ st $(\forall s'$ st $M[\![B]\!] : s \leftrightarrow s';\ s' \in S)\})$ 3.1.1(20)

ii $\subseteq fix(\lambda S.\ \hat{\sim e} \cup \{s \in (\hat{e} \cap T[\![BR]\!])$ st $(\forall s'$ st $M[\![BR]\!] : s \leftrightarrow s';\ s' \in S)\})$ hyp.

iii $\subseteq T[\![WHR]\!]$ 3.1.1(20)

To show that the language is monotonic with respect to M:

9 $\bigwedge_i (T[\![Pi]\!];\ M[\![PiR]\!]$ psat $M[\![Pi]\!]) \Rightarrow$
$T[\![P1; P2]\!];\ M[\![P1R; P2R]\!]$ psat $M[\![P1; P2]\!]$

observe:

i rng $(T[\![P1; P2]\!];\ M[\![P1]\!]) \subseteq T[\![P2]\!]$ 3.1.1(14)

ii $T[\![P1; P2]\!];\ M[\![P1]\!];\ M[\![P2R]\!]$ psat $M[\![P1]\!];\ M[\![P2]\!]$ i,hyp

iii $T[\![P1; P2]\!] \subseteq T[\![P1]\!]$ 3.1.1(14)

iv $T[\![P1; P2]\!];\ M[\![P1R]\!]$ psat $M[\![P1]\!]$ hyp,iii

v $T[\![P1; P2]\!];\ M[\![P1R]\!];\ M[\![P2R]\!]$ psat $M[\![P1]\!];\ M[\![P2]\!]$ ii,iv

vi $T[\![P1; P2]\!];\ M[\![P1R; P2R]\!]$ psat $M[\![P1; P2]\!]$ vi,3.1.1(13)

Similarly:

10 $\bigwedge_i (T[\![Pi]\!];\ M[\![PiR]\!]$ psat $M[\![Pi]\!]) \Rightarrow$
$T[\![IF]\!];\ M[\![IFR]\!]$ psat $M[\![IF]\!]$

The corresponding lemma for the iterative construct is:

11 $T[\![B]\!];\ M[\![BR]\!]$ psat $M[\![B]\!] \Rightarrow$
$T[\![WH]\!];\ M[\![WHR]\!]$ psat $M[\![WH]\!]$

i $M[\![WH]\!] = fix(\lambda R.\ E_{\hat{\sim e}} \cup \hat{e};\ M[\![B]\!];\ R)$ 3.1.1(19)

ii $M[\![WHR]\!] = fix(\lambda R.\ E_{\hat{\sim e}} \cup \hat{e};\ M[\![BR]\!];\ R)$ 3.1.1(19)

iii $(\forall s \in T[\![WH]\!];\ s \in \hat{\sim}e \vee s \in (\hat{e} \cap T[\![B]\!]))$ 3.1.1(20)

iv $T[\![WH]\!];\ (E_{\hat{\sim}e} \cup \hat{e};\ M[\![BR]\!];\ R) \subseteq$
$T[\![WH]\!];\ (E_{\hat{\sim}e} \cup \hat{e};\ M[\![B]\!];\ R)$ hyp,iii

by monotonicity:

v $T[\![WH]\!];\ M[\![WHR]\!]$ psat $T[\![WH]\!];\ M[\![WH]\!]$ iv,i,ii

vi $T[\![WH]\!];\ M[\![WHR]\!]$ psat (dom $M[\![WH]\!]$); $M[\![WH]\!]$ v,4

vii $T[\![WH]\!];\ M[\![WHR]\!]$ psat $M[\![WH]\!]$ vi

After this preparation it is straightforward to see that for any combinator $\mathcal{C}$:

12 $\bigwedge_i(PiR)$ sat $Pi \Rightarrow \mathcal{C}(PiR)$ sat $\mathcal{C}(Pi)$

follows from lemmas 6 to 11.

Monotonicity with respect to the expressions is not shown here since it is not used in the proof rules of the next section.

3.2 Proof Rules

As indicated above, the intention is to design programs by showing that some combinator (applied to sub-specifications) satisfies a given specification. This is obviously a stepwise procedure. It would, of course, be possible to prove any such step of design correct with respect to the language definition given in sub-section 3.1.1. Since Hoare /69a/ it has, however, been obvious that it is far more convenient to conduct such proofs on the basis of specially constructed sets of proof rules. Because of the decision to use post-conditions of initial and final states, the rules here appear more cumbersome. The division into separate conditions does, however, greatly facilitate their use as "check-lists" for "rigorous" justifications. (Further comparison with established methods is given in section 3.5.) In contrast to Tony Hoare's original position, the rules given here are not regarded as a semantics for the language. Instead, the proof rules are proven correct (in section 3.3) with respect to the denotational semantics of section 3.1. One effect of this viewpoint is that it opens the possibility of providing alternative rules for different uses of the same language construct.

Experience has shown that specification and proof via predicates is more convenient than directly in terms of relations. The relational form is, however, presented first and is used in the consistency proof; one program proof in this style is given in section 3.4 for comparison.

3.2.1 Relational Form

Specifications are viewed here (cf. 1.1.2(2)) as relations.

It is straightforward to define:

1 $M[\![SPEC]\!] = SPEC$

2 $T[\![SPEC]\!] = \underline{dom}\ SPEC$

From this it is obvious that, in analogy to 3.1.2(5):

3 $M[\![SPEC]\!]\ \underline{defover}\ T[\![SPEC]\!]$

Also, 1 and 2 show that 3.1.2(1) and 1.2(10) are equivalent.

Furthermore, from 3.1.2(12) it is clear that:

4 $[\![\underline{while}\ e\ \underline{loop}\ SPECB\ \underline{endloop}]\!]\ \underline{sat}\ [\![SPEC]\!]\ \wedge$
$[\![BODY]\!]\ \underline{sat}\ [\![SPECB]\!]\ \Rightarrow$
$[\![\underline{while}\ e\ \underline{loop}\ BODY\ \underline{endloop}]\!]\ \underline{sat}\ [\![SPEC]\!]$

The simplest of the proof rules is for the sequential composition of statements.

SEQ $[\![P1; P2]\!]\ \underline{sat}\ [\![SPEC]\!]$

if the following three conditions hold. Firstly P1 must terminate over a sufficiently large set:

DOM1 $T[\![SPEC]\!] \subseteq T[\![P1]\!]$

Secondly, P2 must also be used only within its termination set:

DOM2 $\underline{rng}\ (T[\![SPEC]\!];\ M[\![P1]\!]) \subseteq T[\![P2]\!]$

Lastly, the result must not contradict the specification over the required set:

RESULT $T[\![SPEC]\!];\ M[\![P1]\!];\ M[\![P2]\!]\ \underline{psat}\ M[\![SPEC]\!]$

As planned, special cases of the guarded conditional 3.1.1(16,17) are provided with proof rules (cf. sub-section 3.3.2). The rule for the conventional if/then/else is:

IFTE $[\![\underline{\text{if}}\ e\ \underline{\text{then}}\ TH\ \underline{\text{else}}\ EL\ \underline{\text{endif}}]\!]\ \underline{\text{sat}}\ [\![SPEC]\!]$

providing the following five conditions hold. Firstly, the evaluation of the expression must be defined:

DOMX $T[\![SPEC]\!] \subseteq T[\![e]\!]$

Next, both embedded statements must terminate over the set of states which they may encounter:

DOMTH $(T[\![SPEC]\!] \cap \hat{e}) \subseteq T[\![TH]\!]$

DOMEL $(T[\![SPEC]\!] \cap \hat{\sim e}) \subseteq T[\![EL]\!]$

Lastly, neither of the potential execution paths should contradict the specification:

RESULTH $(T[\![SPEC]\!] \cap \hat{e});\ M[\![TH]\!]\ \underline{\text{psat}}\ M[\![SPEC]\!]$

RESULTEL $(T[\![SPEC]\!] \cap \hat{\sim e});\ M[\![EL]\!]\ \underline{\text{psat}}\ M[\![SPEC]\!]$

The obvious rule for the case where the else clause is absent is:

IFT $[\![\underline{\text{if}}\ e\ \underline{\text{then}}\ TH\ \underline{\text{endif}}]\!]\ \underline{\text{sat}}\ [\![SPEC]\!]$

if:

DOMX $T[\![SPEC]\!] \subseteq T[\![e]\!]$

DOMTH $(T[\![SPEC]\!] \cap \hat{e}) \subseteq T[\![TH]\!]$

RESULTH $(T[\![SPEC]\!] \cap \hat{e});\ M[\![TH]\!]\ \underline{\text{psat}}\ M[\![SPEC]\!]$

RESULTEL $(T[\![SPEC]\!] \cap \hat{\sim e});\ E_{St}\ \underline{\text{psat}}\ M[\![SPEC]\!]$

For the guarded conditional itself:

IFGD $[\![\underline{\text{if}}\ e1 \rightarrow P1 \,[\!]\, \ldots \,[\!]\, en \rightarrow Pn]\!]\ \underline{\text{sat}}\ [\![SPEC]\!]$

providing 3n + 1 conditions hold; none of the n expressions must fail for any state required by the specification:

$DOMX_i$ $\quad T[\![SPEC]\!] \subseteq T[\![e_i]\!]$

At least one expression must evaluate to true for any expected state:

DOMC $\quad T[\![SPEC]\!] \subseteq \underline{\text{union}}\ \{ \hat{e}i\ \underline{\text{st}}\ 1 \leq i \leq n \}$

Each of the n embedded statements must be adequately defined:

DOM_i $\quad (T[\![SPEC]\!] \cap \hat{e}i) \subseteq T[\![Pi]\!]$

Finally, none of the n potential paths should contradict the specification:

$RESULT_i$ $\quad (T[\![SPEC]\!] \cap \hat{e}i);\ M[\![Pi]\!]\ \underline{\text{psat}}\ M[\![SPEC]\!]$

Since the main interest here is on a development method, rules for the basic statements are of less interest than in the standard axiomatic/predicate transformer systems. It is, perhaps, worth sketching a rule for assignment:

ASSN $\quad [\![x:=e]\!]\ \underline{\text{sat}}\ [\![SPEC]\!]$

if:

DOM $\quad M[\![e]\!]\ \underline{\text{defover}}\ T[\![SPEC]\!]$ (and syntactic types match)

RESULT $\quad \{ s \leftrightarrow (s \dagger \{ x \mapsto M[\![e]\!]\ s \})\ \underline{\text{st}}\ s \in T[\![SPEC]\!] \} \subseteq M[\![SPEC]\!]$

It might be worthwhile to define, also, multiple assignment because its use can avoid over-specification of order.

A variety of methods are in use for proving results about iterative statements (e.g. invariants, weakest pre-conditions, tail induction). They are distinguished not so much by fundamental power as by convenience for various applications. Here, a range of proof rules is offered (all within the relation scheme) which are shown in section 3.4 to have different domains of usefulness. The reader who is familiar with the invariant rule of Hoare /69a/ will certainly be dismayed by the number of conditions in these rules. In forming a judgement on the length it is well to bear in mind that each condition does correspond to a meaningful result which must be established: the separation of these conditions is an aid to their use as a mental check-list.

The "down" rules are shown in section 3.4 to be useful where the initial values of variables are destroyed in a computation:

WHILEDN $[\![\underline{\text{while}}\ e\ \underline{\text{loop}}\ B\ \underline{\text{endloop}}]\!]\ \underline{\text{sat}}\ [\![\text{SPEC}]\!]$

if a set ($Sl \subseteq St$) of states can be found for which the following conditions hold; B is defined on Sl:

BODY $(Sl \cap \hat{e});\ M[\![B]\!] \subseteq Sl \times Sl$

The required initial states are contained in Sl:

DOMLOOP $T[\![\text{SPEC}]\!] \subseteq Sl$

The evaluation of the loop test must be defined in any state of Sl:

DOMX $Sl \subseteq T[\![e]\!]$

The loop body must be defined whenever it can be executed:

DOMBODY $\quad Sl \cap \hat{e} \subseteq T[\![B]\!]$

"Falling through" the loop test must not contradict the specification (basis of induction):

FALL $\quad \hat{\sim e};\ E_{Sl}$ psat $M[\![SPEC]\!]$

The body of the loop composed with the specification must be contained in the specification (inductive step):

COMP $\quad (\hat{e} \cap Sl);\ M[\![B]\!];\ M[\![SPEC]\!]$ psat $M[\![SPEC]\!]$

Termination proofs should be an integral part of a development process. Here, a well-founded relation VAR over Sl ($>$) must be provided for which the loop terminates:

STOP $\quad Sl\text{-}\underline{dom}\ VAR \subseteq \hat{\sim e}$

and the body always causes a decrease in the ordering:

DECR $\quad (\hat{e} \cap Sl);\ M[\![B]\!] \subseteq VAR$

There are several interesting points about the general form of iterative rules given here which can be discussed in terms of WHILEDN. The rôle of the set Sl will be taken over in the predicate form of these rules by a data type invariant. It must be understood, however, that such "invariants" may be false within the body of the loop: the BODY condition is only a constraint on the input/output behaviour. The use of an arbitrary well-founded relation VAR is more convenient than the more conventional function from states to the natural numbers in three ways:

i) The ability to have many stopping values can avoid the necessity to define a conditional expression which, somewhat artificially, maps them all to zero (cf. integer division example 3.4(14 to 22)).

ii) Some odd formulations like the "number of trailing zeros in the binary representation of ...." (cf. p.110 of Jones /80a/) can be replaced by natural relations.

iii) More general orderings like bag containment for the indices of out-of-order pairs of elements in a sort, or lexical order, can be used.

One other interesting observation about WHILEDN is that it can be derived from viewing the iterative construct as:

11 WH = <u>while</u> e <u>loop</u> B <u>endloop</u>

12 IF = <u>if</u> e <u>then</u> B; SPEC <u>endif</u>

with:

13 ⟦IF⟧ <u>sat</u> ⟦SPEC⟧

being a necessary condition for:

14 ⟦WH⟧ <u>sat</u> ⟦SPEC⟧

as in Linger /79a/.

Returning now to the presentation of proof rules for iterative constructs, it is much more useful - especially in design - to have rules available which include the initialisation. With the "down" version of these rules, the first step in design is usually the

discovery of an invariant relation (REST) which summarises the relation computed by the remaining iterations of the loop. Thus:

IWHILEDN $[\![\text{INIT; } \underline{\text{while}}\ e\ \underline{\text{loop}}\ B\ \underline{\text{endloop}}]\!]\ \underline{\text{sat}}\ [\![\text{SPEC}]\!]$

if:

$(S1 \cap \hat{e});\ M[\![B]\!] \subseteq S1 \times S1$

$REST \subseteq S1 \times S1$

DOMINIT $T[\![\text{SPEC}]\!] \subseteq T[\![\text{INIT}]\!]$

DOMLOOP $\underline{\text{rng}}\ (\underline{\text{dom}}\ \text{SPEC};\ \text{INIT}) \subseteq S1$

DOMX $S1 \subseteq T[\![e]\!]$

DOMBODY $S1 \cap \hat{e} \subseteq T[\![B]\!]$

FALL $\hat{\sim e};\ E_{S1}\ \underline{\text{psat}}\ REST$

COMP $(\hat{e} \cap S1);\ M[\![B]\!];\ REST\ \underline{\text{psat}}\ REST$

RESULT $(\underline{\text{dom}}\ \text{SPEC});\ M[\![\text{INIT}]\!];\ REST\ \underline{\text{psat}}\ M[\![\text{SPEC}]\!]$

The STOP and DECR conditions are as for WHILEDN.

The "down" rules are most convenient where the weakest pre-condition type logic is weakest (i.e. where input values are modified by the body of a loop). But in this same area it is sometimes easier to design by choosing some "constant expressions" (this is illustrated in 3.4 (14 to 22)). Thus:

WHILEFIX $[\![\underline{\text{while}}\ e\ \underline{\text{loop}}\ B\ \underline{\text{endloop}}]\!]\ \underline{\text{sat}}\ [\![\text{SPEC}]\!]$

if:

$(S1 \cap \hat{e});\ [\![B]\!] \subseteq S1 \times S1$

$EQ = \{ s \leftrightarrow s' \in S1 \times S1\ \underline{\text{st}}\ [\![ex]\!] s = [\![ex]\!] s' \}$

The DOMLOOP, DOMX and DOMBODY conditions are as for WHILEDN.

Then:

CONSTANCY $(Sl \cap \hat{e})$; $M[\![B]\!] \subseteq EQ$

RESULT Sl; EQ; $\hat{\sim e}$ psat $M[\![SPEC]\!]$

The STOP and DECR conditions are also as for WHILEDN.

In those loops in which temporary variables are counted up to the value of some (unchanged) initial value, it is normally simpler to express the effect of the initialisation and early iterations of a loop in a relation (FRONT). The "up" rule is:

IWHILEUP $[\![INIT;\ \underline{while}\ e\ \underline{loop}\ B\ \underline{endloop}]\!]$ sat $[\![SPEC]\!]$

if:

$(Sl \cap \hat{e})$; $M[\![B]\!] \subseteq Sl \times Sl$

$FRONT \subseteq St \times Sl$

The DOMINIT, DOMLOOP, DOMX and DOMBODY conditions are as for IWHILEDN. Then:

BASIS $T[\![SPEC]\!]$; $M[\![INIT]\!]$ psat $FRONT$

COMP $FRONT$; $\hat{e}$; $M[\![B]\!]$ psat $FRONT$

RESULT $FRONT$; $\hat{\sim e}$ psat $M[\![SPEC]\!]$

The STOP and DECR conditions are as for WHILEDN.

3.2.2 Predicate Form

The rules presented in this sub-section are (except FORTO and FORARB) straightforward translations of those in the preceding sub-section. Section 1.4 proposes that specifications be given by:

1 St, pre: $St \to Bool$, $post$: $St \times St \to Bool$

with the proviso:

2 $(\forall s \in St;\ pre(s) \Rightarrow (\exists s' \in St;\ post(s,s')))$

Thus:

3 $M[\![St, pre, post]\!] = \{s \leftrightarrow s' \in St \times St \ \underline{st}\ pre(s) \wedge post(s,s')\}$

4 $T[\![St, pre, post]\!] = \{s \in St \ \underline{st}\ pre(s)\}$

Because of 2, it follows that:

5 $T[\![St, pre, post]\!] \subseteq \underline{dom}\ M[\![St, pre, post]\!]$

To fit the style of the rules given below a function will be used:

6 $eval(e,s) = [\![e]\!]\, s$

The rules are given without the verbal description of the preceding sub-section: the more compact form makes reference easier when they are used.

SEQ $[\![(St,pre1,post1);\ (St,pre2,post2)]\!]$ $\underline{sat}$ $[\![(St,pre,post)]\!]$

if:

DOM1 $(\forall s \in St\ \underline{st}\ pre(s);\ pre1(s))$

DOM2 $(\forall s,s' \in St\ \underline{st}\ pre(s);\ post1(s,s') \Rightarrow pre2(s'))$

RESULT $(\forall s,s',s'' \in St\ \underline{st}\ pre(s);\ post1(s,s') \wedge post2(s',s'') \Rightarrow post(s,s''))$

IFTE $[\![\underline{if}\ e\ \underline{then}\ (St,preTH,postTH)\ \underline{else}\ (St,preEL,postEL)]\!]$ $\underline{sat}$ $[\![(St,pre,post)]\!]$

if:

DOMX $(\forall s \in St\ \underline{st}\ pre(s);\ (\exists b \in Bool;\ b = eval(e,s)))$

DOMTH $(\forall s \in St\ \underline{st}\ pre(s);\ eval(e,s) \Rightarrow preTH(s))$

DOMEL $(\forall s \in St \ \underline{st}\ pre(s);\ \sim eval(e,s) \Rightarrow preEL(s))$

RESULTH $(\forall s,s' \in St \ \underline{st}\ pre(s);\ eval(e,s) \wedge postTH(s,s') \Rightarrow post(s,s'))$

RESULTEL $(\forall s,s' \in St \ \underline{st}\ pre(s);\ \sim eval(e,s) \wedge postEL(s,s') \Rightarrow post(s,s'))$

IFT $[\![\underline{if}\ e\ \underline{then}\ (St,preTH,postTH)\ \underline{endif}]\!]\ \underline{sat}\ [\![(St,pre,post)]\!]$

if:

DOMX $(\forall s \in St \ \underline{st}\ pre(s);\ (\exists b \in Bool;\ b = eval(e,s)))$

DOMTH $(\forall s \in St \ \underline{st}\ pre(s);\ eval(e,s) \Rightarrow preTH(s))$

RESULTH $(\forall s,s' \in St \ \underline{st}\ pre(s);\ eval(e,s) \wedge postTH(s,s') \Rightarrow post(s,s'))$

RESULTEL $(\forall s \in St \ \underline{st}\ pre(s);\ \sim eval(e,s) \Rightarrow post(s,s))$

IFGD $[\![\underline{if}\ e1 \rightarrow (St,preP1,postP1) [\!] \ldots [\!] en \rightarrow (St,prePn,postPn)]\!]\ \underline{sat}\ [\![(St,pre,post)]\!]$

if:

$DOMX_i$ $(\forall s \in St \ \underline{st}\ pre(s);\ (\exists b \in Bool;\ b = eval(ei,s)))$

DOMC $(\forall s \in St \ \underline{st}\ pre(s);\ \bigvee_i eval(ei,s))$

DOM_i $(\forall s \in St \ \underline{st}\ pre(s);\ eval(ei,s) \Rightarrow prePi(s))$

$RESULT_i$ $(\forall s,s' \in St \ \underline{st}\ pre(s);\ eval(ei,s) \wedge postPi(s,s') \Rightarrow post(s,s'))$

WHILEDN $[\![\underline{while}\ e\ \underline{loop}\ BODY\ \underline{endloop}]\!]\ \underline{sat}\ [\![(St,pre,post)]\!]$

if:

invl: $St \rightarrow Bool$

$Sl = \{s \in St \ \underline{st}\ invl(s)\}$

$[\![BODY]\!]\ \underline{sat}\ [\![(Sl,preBODY,postBODY)]\!]$

DOMLOOP $(\forall s \in St \ \underline{st}\ pre(s);\ invl(s))$

DOMX $(\forall s \in Sl;\ (\exists b \in Bool;\ b = eval(e,s)))$

DOMBODY $(\forall s \in Sl;\ eval(e,s) \Rightarrow preBODY(s))$

FALL $(\forall s \in Sl;\ \sim eval(e,s) \Rightarrow post(s,s))$

COMP $(\forall s,s',s'' \in Sl;\ eval(e,s) \wedge postBODY(s,s') \wedge post(s',s'') \Rightarrow post(s,s''))$

var: $Sl \to Wfs$ with $<$, $Min \subseteq Wfs$

STOP $(\forall s \in Sl;\ var(s) \in Min \Rightarrow \sim eval(e,s))$

DECR $(\forall s,s' \in Sl;\ eval(e,s) \wedge postBODY(s,s') \Rightarrow var(s') < var(s))$

IWHILEDN $[\![INIT;\ \underline{while}\ e\ \underline{loop}\ BODY\ \underline{endloop}]\!]\ \underline{sat}\ [\![(St,pre,post)]\!]$

if:

invl: $St \to Bool$

$Sl = \{s \in St\ \underline{st}\ invl(s)\}$

rest: $Sl \times Sl \to Bool$

$[\![INIT]\!]\ \underline{sat}\ [\![(St,preINIT,postINT)]\!]$

$[\![BODY]\!]\ \underline{sat}\ [\![(Sl,preBODY,postBODY)]\!]$

DOMINIT $(\forall s \in St\ \underline{st}\ pre(s);\ preINIT(s))$

DOMLOOP $(\forall s,s' \in St\ \underline{st}\ pre(s);\ postINIT(s,s') \Rightarrow invl(s'))$

DOMX $(\forall s \in Sl;\ (\exists b \in Bool;\ b = eval(e,s)))$

DOMBODY $(\forall s \in Sl;\ eval(e,s) \Rightarrow preBODY(s))$

FALL $(\forall s \in Sl;\ \sim eval(e,s) \Rightarrow rest(s,s))$

COMP $(\forall s,s',s'' \in Sl;\ eval(e,s) \wedge postBODY(s,s') \wedge rest(s',s'') \Rightarrow rest(s,s''))$

RESULT $(\forall s \in St,s',s'' \in Sl\ \underline{st}\ pre(s);$
$postINIT(s,s') \wedge rest(s',s'') \Rightarrow post(s,s''))$

var: $Sl \to Wfs$ with $<$, $Min \subseteq Wfs$

STOP $(\forall s \in Sl;\ var(s) \in Min \Rightarrow \sim eval(e,s))$

DECR $(\forall s,s' \in Sl;\ eval(e,s) \wedge postBODY(s,s') \Rightarrow var(s') < var(s))$

IWHILEFIX $[\![INIT:\ \underline{while}\ e\ \underline{loop}\ BODY\ \underline{endloop}]\!]\ \underline{sat}\ [\![(St,pre,post)]\!]$

if:

invl: St → Bool

Sl = $\{s \in St \ \underline{st}\ invl(s)\}$

$[\![INIT]\!]$ sat $[\![(St, preINIT, postINIT)]\!]$

$[\![BODY]\!]$ such that

cons: Sl → Val-list

is constant

DOMINIT $(\forall s \in St \ \underline{st}\ pre(s);\ preINIT(s))$

DOMLOOP $(\forall s,s' \in St \ \underline{st}\ pre(s);\ postINIT(s,s') \Rightarrow invl(s'))$

DOMX $(\forall s \in Sl;\ (\exists b \in Bool;\ b = eval(e,s)))$

DOMBODY $(\forall s \in Sl;\ eval(e,s) \Rightarrow preBODY(s))$

CONSTANCY $(\forall s,s' \in Sl;\ eval(e,s) \wedge postBODY(s,s') \Rightarrow cons(s) = cons(s'))$

RESULT $(\forall s \in St,\ s',\ s'' \in Sl \ \underline{st}\ pre(s);$
$postINIT(s,s') \wedge cons(s') = cons(s'') \wedge \sim eval(e,s'') \Rightarrow post(s,s''))$

var: Sl → Wfs with $<$, Min $\subseteq$ Wfs

STOP $(\forall s \in Sl;\ var(s) \in Min \Rightarrow \sim eval(e,s))$

DECR $(\forall s,s' \in Sl;\ eval(e,s) \wedge postBODY(s,s') \Rightarrow var(s') < var(s))$

IWHILEUP $[\![INIT;\ \underline{while}\ e\ \underline{loop}\ BODY\ \underline{endloop}]\!]$ sat $[\![(St, pre, post)]\!]$

if:

invl: St → Bool

Sl = $\{s \in St \ \underline{st}\ invl(s)\}$

front: St x Sl → Bool

$[\![INIT]\!]$ sat $[\![(St, preINIT, postINIT)]\!]$

$[\![BODY]\!]$ sat $[\![(Sl, preBODY, postBODY)]\!]$

DOMINIT $(\forall s \in St \ \underline{st}\ pre(s);\ preINIT(s))$

DOMLOOP $(\forall s,s' \in St \ \underline{st}\ pre(s);\ postINIT(s,s') \Rightarrow invl(s'))$

DOMX $(\forall s \in Sl;\ (\exists b \in Bool;\ b = eval(e,s)))$

DOMBODY $(\forall s \in Sl;\ eval(e,s) \Rightarrow preBODY(s))$

BASIS $(\forall s \in St,\ s' \in Sl \ \underline{st}\ pre(s);\ postINIT(s,s') \Rightarrow front(s,s'))$

COMP $(\forall s,s',s'' \in Sl;\ front(s,s') \wedge eval(e,s') \wedge postBODY(s',s'') \Rightarrow front(s,s''))$

RESULT $(\forall s \in St, s' \in Sl;\ front(s,s') \wedge \sim eval(e,s') \Rightarrow post(s,s'))$

var: $Sl \rightarrow Wfs$ with $<$, $Min \subseteq Wfs$

STOP $(\forall s \in Sl;\ var(s) \in Min \Rightarrow \sim eval(e,s))$

DECR $(\forall s,s' \in Sl;\ eval(e,s) \wedge postBODY(s,s') \Rightarrow var(s') < var(s))$

It would be possible to prove results about for/to style loops via the equivalent form:

i:=v1; <u>while</u> $i \leq$ v2 <u>loop</u> B(i); i:=i+1; <u>endloop</u>

It is, however, more in the spirit of the rigorous method to tailor a special rule to this task. (The control variable is treated as a local constant in Ada. This is reflected by making B a general operation with an input.)

<u>FORTO</u> $[\![$ <u>for</u> i <u>in</u> (<u>integer</u> <u>range</u> v1..v2) <u>loop</u> B(i) <u>endloop</u> $]\!]$ <u>sat</u> $[\![(St,pre,post)]\!]$

invl: St x Int $\rightarrow$ Bool

$Sl = \{s \in St \ \underline{st}\ invl(s,i) \wedge i \in \{v1..v2\}\}$

front: St x Int x Sl $\rightarrow$ Bool

$[\![B(i)]\!]$ <u>sat</u> (Sl,preB,postB) for $i \in \{v1..v2\}$

DOMLOOP $(\forall s \in St \ \underline{st}\ pre(s);\ invl(s,v1))$

DOMBODY $(\forall s \in Sl;\ i \leq v2 \Rightarrow preB(s,i))$

BASIS $(\forall s \in St \ \underline{st}\ pre(s);\ front(s,v1,s))$

COMP $(\forall s \in St,s',s'' \in Sl;\ front(s,i,s') \wedge i \leq v2 \wedge postBODY(s',i,s'') \Rightarrow front(s,i+1,s''))$

RESULT $(\forall s \in St,s' \in Sl;\ front(s,i,s') \wedge i > v2 \Rightarrow post(s,s'))$

Notice that termination follows from the form of the construct.

It is possible to go beyond FORTO to a proof rule which covers the case where the order in which elements of the index set are used is immaterial. It can be very important to record such freedom of order in a program design because of the difficulty in detecting potential reordering (or parallelism) in programs where the commitment to order has been forced by the language rather than the problem. Thus:

$\underline{\text{while}}\ \mathcal{S} \neq \{\}\ \underline{\text{loop}}\ \underline{\text{let}}\ e \in \mathcal{S}\ ;\ B(e);\ \mathcal{S} := \mathcal{S} - \{e\}\ ;\ \underline{\text{endloop}}$

can be handled by:

FORARB $[\![\underline{\text{for}}\ e \in \mathcal{S}\ \underline{\text{loop}}\ B(e)\ \underline{\text{endloop}}]\!]\ \underline{\text{sat}}\ [\![\text{SPEC}]\!]$

if:

invl: St x El-set → Bool

$S1 = \{s \in St\ \underline{\text{st}}\ invl(s,e) \wedge e \in S\}$

$[\![B(e)]\!]\ \underline{\text{sat}}\ (S1, preB, postB)$ for $e \in S$

DOMLOOP $(\forall s \in St\ \underline{\text{st}}\ pre(s);\ invl(s,\mathcal{S}))$

DOMBODY $(\forall s \in St;\ invl(s,\mathcal{S}) \wedge e \in \mathcal{S} \Rightarrow preB(s,e))$

FALL $(\forall s \in S1;\ post(s,\{\},s))$

COMP $(\forall s,s',s'' \in S1;\ e \in \mathcal{S} \wedge postB(s,e,s') \wedge post(s',\mathcal{S} - \{e\},s'') \Rightarrow post(s,\mathcal{S},s''))$

3.3 Justification of Proof Rules

A selection of the proof rules of the preceding section are proved valid with respect to the denotational semantics of sub-section 3.1.1. It is more convenient to undertake these proofs for the relational versions of the rules. Thus any references to the named rules in this section should be taken to refer to sub-section 3.2.1.

3.3.1 Sequential Rule

It is shown that under the conditions of 3.2.1(SEQ) the sequential composition must satisfy the specification.

1	SEQ establishes that $[\![P1;\ P2]\!]$ sat $[\![SPEC]\!]$	
i	$T[\![SPEC]\!] \subseteq T[\![P1]\!]$	DOM1
ii	$\underline{rng}\ (T[\![SPEC]\!];\ M[\![P1]\!]) \subseteq T[\![P2]\!]$	DOM2
iii	$T[\![SPEC]\!] \subseteq \{s \in T[\![P1]\!]$ st $(\forall s'$ st $M[\![P1]\!] : s \leftrightarrow s';\ s' \in T[\![P2]\!])\}$	i,ii
iv	$T[\![SPEC]\!] \subseteq T[\![P1;\ P2]\!]$	iii,3.1.1(14)
v	$M[\![P1;\ P2]\!] = M[\![P1]\!]\ ;\ M[\![P2]\!]$	3.1.1(13)
vi	$T[\![SPEC]\!];\ M[\![P1;\ P2]\!]$ psat $M[\![SPEC]\!]$	v,RESULT
	$\therefore [\![P1;\ P2]\!]$ sat $[\![SPEC]\!]$	iv,vi,3.1.2(1)

3.3.2 Conditional Rules

In order to justify the proof rule for the standard if/then/else construct 3.2.1(IFTE), it is first necessary to derive the semantics as a special case of 3.1.1 (15 to 17):

1 IF = if e then TH else EL endif

2 $M[\![IF]\!] = \hat{e};\ M[\![TH]\!] \cup \hat{\sim e};\ M[\![EL]\!]$

3 $T[\![IF]\!] = (\hat{e} \cap T[\![TH]\!]) \cup (\hat{\sim e} \cap T[\![EL]\!])$

4 isdisj $(\hat{e},\ \hat{\sim e})$

Thus the required theorem is:

5 IFTE establishes that $[\![IF]\!]$ sat $[\![SPEC]\!]$

i $T[\![SPEC]\!] \subseteq T[\![e]\!]$ DOMX

ii $T[\![SPEC]\!] \subseteq (\hat{e} \cup \hat{\sim e})$ i,3.1.1(10,11)

iii $(T[\![SPEC]\!] \cap \hat{e}) \subseteq T[\![TH]\!]$ DOMTH

iv $(T[\![SPEC]\!] \cap \hat{e}) \subseteq \hat{e} \cap T[\![TH]\!]$ iii

similarly:

v $(T[\![SPEC]\!] \cap \hat{\sim e}) \subseteq \hat{\sim e} \cap T[\![EL]\!]$ DOMEL

vi $T[\![SPEC]\!] \subseteq (\hat{e} \cap T[\![TH]\!]) \cup (\hat{\sim e} \cap T[\![EL]\!])$ iv,v,ii

vii $T[\![SPEC]\!] \subseteq T[\![IF]\!]$ vi,3

viii $(T[\![SPEC]\!] \cap \hat{e});\ M[\![TH]\!] \subseteq M[\![SPEC]\!]$ RESULTH

ix $(T[\![SPEC]\!] \cap \hat{\sim e});\ M[\![EL]\!] \subseteq M[\![SPEC]\!]$ RESULTEL

x $((T[\![SPEC]\!] \cap \hat{e});\ M[\![TH]\!]) \cup (T[\![SPEC]\!] \cap \hat{\sim e});\ M[\![EL]\!]) \subseteq M[\![SPEC]\!]$ viii,ix

xi $T[\![SPEC]\!];\ ((\hat{e};\ M[\![TH]\!]) \cup (\hat{\sim e};\ M[\![EL]\!])) \subseteq M[\![SPEC]\!]$ ii,x

xii $T[\![SPEC]\!];\ M[\![IF]\!]$ psat $M[\![SPEC]\!]$ xi,2

$\therefore [\![IF]\!]$ sat $[\![SPEC]\!]$ vii,xii,3.1.2(1)

3.3.3 Iterative Rules

The proof of 3.2.1(WHILEDN) is slightly complicated by the need for induction and is therefore split into parts. The first lemma is:

1 WHILEDN establishes that Sl; M ⟦WH⟧ psat M ⟦SPEC⟧

Use Scott induction (M in 3.1.1(19) is continuous), basis:

i $(Sl; \Omega) = \Omega \subseteq M[\![SPEC]\!]$

inductive step:

ii	$Sl \subseteq T[\![e]\!]$	DOMX
iii	$Sl \subseteq (\hat{e} \cup \hat{\sim e})$	ii,3.1.1(10,11)
iv	$\hat{\sim e}; E_{Sl} \subseteq M[\![SPEC]\!]$	FALL
v	$\underline{rng}\ ((Sl \cap \hat{e}); M[\![B]\!]) \subseteq Sl$	BODY
vi	$Sl; R \subseteq M[\![SPEC]\!]$	I.H.
vii	$(Sl \cap \hat{e}); M[\![B]\!]; M[\![SPEC]\!] \subseteq M[\![SPEC]\!]$	COMP
viii	$(Sl \cap \hat{e}); M[\![B]\!]; R \subseteq M[\![SPEC]\!]$	v,vi,vii
ix	$Sl; (E_{\hat{\sim e}} \cup \hat{e}; M[\![B]\!]; R) \subseteq M[\![SPEC]\!]$	iv,viii
	$\therefore$ Sl; M ⟦WH⟧ psat M ⟦SPEC⟧	ix,3.1.1(19)

Then:

2 WHILEDN establishes that $Sl \subseteq T[\![WH]\!]$

i	$Sl \subseteq T[\![e]\!]$	DOMX
ii	$T[\![e]\!] = \hat{e} \cup \hat{\sim e}$	3.1.1(10,11)

By (complete) transfinite induction on Sl using the well-founded order given by VAR, basis:

iii	$s \in (Sl\ \underline{-dom}\ VAR)$	
iv	$s \in \hat{\sim e}$	iii,STOP
v	$\therefore s \in T[\![WH]\!]$	iv,3.1.1(20)

otherwise:

vi $s \in Sl \cap \underline{dom}\ VAR$

for $s \in \hat{\sim e}$ argument is as in iv,v; so:

vii	$s \in \hat{e}$	ii
viii	$s \in T[\![B]\!]$	vii, DOMBODY
ix	$(\forall s'\ \underline{st}\ M[\![B]\!] : s \leftrightarrow s';\ s' \notin Sl)$	vi,vii,BODY
x	$(\forall s'\ \underline{st}\ M[\![B]\!] : s \leftrightarrow s';\ VAR: s \leftrightarrow s')$	vi,vii,ix,DECR
xi	$\therefore\ (\forall s'\ \underline{st}\ M[\![B]\!] : s \leftrightarrow s';\ s' \in T[\![WH]\!])$	x,IH
xii	$s \in T[\![WH]\!]$	vii,xi,3.1.1(20)
	$\therefore\ Sl \subseteq T[\![WH]\!]$	v,xii

Then finally:

3 WHILEDN establishes that $[\![WH]\!]$ $\underline{sat}$ $[\![SPEC]\!]$

follows from 1,2,DOMLOOP.

3.3.4 On "Healthiness"

Dijkstra /76a/ requires that any putative proof rules should satisfy certain "healthiness" criteria. This is necessary because no model is given of the underlying language. Although the use of a denotational semantic basis for the language here removes the need to address this question, it is interesting to consider the notion. The analogous concept to "weakest pre-condition" might be the largest set over which a program matches a specification. Thus:

1 ls: Prog x (St x St) $\rightarrow$ $\wp$(St)

ls(P,R) is largest S such that

$S \subseteq T[\![P]\!] \wedge S;\ M[\![P]\!]$ $\underline{psat}$ R

The "law of the excluded miracle" becomes:

2 $ls(P, \Omega) = \{\}$

which follows from defover.

The monotonicity requirement becomes:

3 $Q \subseteq R \Rightarrow ls(P,Q) \subseteq ls(P,R)$

which follows from defover and psat.

The third condition becomes:

4 $ls(P,Q) \cap ls(P,R) = ls(P, Q \cap R)$

The argument being similar to that used by E.W. Dijkstra.

The fourth condition becomes:

5 $ls(P,Q) \cup ls(P,R) \subseteq ls(P, Q \cup R)$

because, with:

i $q = ls(P,Q)$

ii $r = ls(P,R)$

iii $[\![P]\!]$ defover $q \wedge [\![P]\!]$ defover r

iv $\therefore [\![P]\!]$ defover $(q \cup r)$

v $q;\ M[\![P]\!] \subseteq Q \wedge r;\ M[\![P]\!] \subseteq R$

vi $(q \cup r);\ M[\![P]\!] \subseteq Q \cup R$

$\therefore\ q \cup r \subseteq ls(P, Q \cup R)$

To see that 5 is not an equality in the non-deterministic case, consider Dijkstra's example:

$P = [\![x :\in \{1,2\}]\!]$

$Q = \{x \leftrightarrow x' \text{ st } x' = 1\}$

$R = \{x \leftrightarrow x' \text{ st } x' = 2\}$

In connection with the discussion in sub-section 3.1.1, it should be mentioned that the rule for introducing existential quantifiers would have to rely on an assumption of bounded non-determinism.

3.4 Examples

The chosen examples illustrate points of contrast between the various proof rules rather than providing realistic applications - some more interesting uses are contained in chapter 5. With the exception of the very first example (1 to 5), the predicate form of the rules is used in each case. Unqualified references to the names of proof rules should be taken to refer to sub-section 3.2.2.

Consider a specification for factorial:

1 St::N:Int FN:Int

2 SPEC = $\{ \text{mk-St}(n,fn) \leftrightarrow \text{mk-St}(n',fn') \ \underline{\text{st}} \ n \geqslant 0 \wedge fn' = n! * fn \}$

Notice that negative values for N are deliberately allowed in 1 to show the rôle of a "pre-condition".

Since no constraint is put on the final value of N, a design which destroys N is permitted:

3 FACT = $[\![\ \underline{\text{while}} \ N \neq 0 \ \underline{\text{loop}} \ \text{BODY} \ \underline{\text{endloop}} \ ]\!]$

4 $[\![\text{BODY}]\!] = \{ \text{mk-St}(n,fn) \leftrightarrow \text{mk-St}(n',fn') \ \underline{\text{st}} \ fn' = fn * n \wedge n' = n-1 \}$

The overwriting of the initial values must be handled by a proof rule (not a state extension) which keeps track of the states involved:

3.2.1 (WHILEDN) is designed specifically for such cases. Thus to show:

5 FACT sat SPEC

i Sl $\subseteq$ St

inv $n \geqslant 0$

Then BODY becomes:

ii $(Sl \cap \widehat{n \neq 0})$; $[\![BODY]\!] \subseteq Sl \times Sl$

DOMLOOP:

iii $\underline{dom}$ SPEC $\subseteq$ Sl

DOMX:

iv $\underline{dom}$ $[\![N \neq 0]\!] \supseteq St \supseteq Sl$

DOMBODY:

v $Sl \cap \widehat{n \neq 0} \subseteq \underline{dom}\ [\![BODY]\!]$

FALL:

vi $\widehat{\sim n \neq 0}$; E_{Sl} $\underline{psat}$ SPEC

from:

vii $\{mk\text{-}St(n,fn) \leftrightarrow mk\text{-}St(n,fn)\ \underline{st}\ n = 0\} \subseteq$ SPEC

COMP:

viii $(\widehat{n \neq 0} \cap Sl)$; $M[\![BODY]\!]$; $M[\![SPEC]\!]$ $\underline{psat}$ $M[\![SPEC]\!]$

from:

ix $(\exists n', fn' \in Int;\ n > 0 \wedge fn' = fn * n \wedge n' = n-1 \wedge fn'' = n'! * fn') \Rightarrow$
$\{mk\text{-}St(n,fn) \leftrightarrow mk\text{-}St(n'',fn'')\} \subseteq$ SPEC

and with a well-founded relation:

x VAR = $\{mk\text{-}St(n,fn) \leftrightarrow mk\text{-}St(n',fn')\ \underline{st}\ 0 \leq n' < n\}$

STOP:

xi Sl-$\underline{dom}$ VAR $\subseteq \widehat{\sim n \neq 0}$

DECR:

xii BODY $\subseteq$ VAR

As an illustration of how the separate conditions serve as a check-list, the same example can now be tackled using the predicate form of the proof rule:

6 SPEC
globals N:wr Int, FN:wr Int
pre $n \geqslant 0$
post $fn' = n! * fn$

FACT is as in 3; BODY is:

7 BODY
globals N:wr Int, FN:wr Int
post $fn' = fn * n \wedge n' = n-1$

To show:

8 ⟦FACT⟧ sat ⟦SPEC⟧

use WHILEDN with:

i $invl(mk\text{-}St(n,fn)) \triangleq n \geqslant 0$

DOMLOOP becomes:

ii $n \geqslant 0 \Rightarrow n \geqslant 0$

DOMX:

iii Test defined on a superset of St

DOMBODY:

iv $n \geqslant 0 \wedge n \neq 0 \Rightarrow$ TRUE

FALL:

v $n = 0 \Rightarrow fn = n! * fn$

COMP:

vi $n \neq 0 \wedge fn' = fn * n \wedge n' = n-1 \wedge fn'' = n'! * fn' \Rightarrow fn'' = n! * fn$

Then with:

var: Sl → NatO

vii var(mk-St(n,fn)) ≙ n

STOP

viii n = 0 ⇒ ~(n ≠ 0)

DECR:

ix n' = n-1 ⇒ n' < n

As is pointed out in sub-section 3.2.1 the rule which deals directly with initialised iterations is frequently of more use. Although the while loop of 6-8, 3 could be considered to be one component in a sequential decomposition, such a design step would be far harder to understand than one which recognises the natural affinity of initialisation and loop body. Thus with:

9 SPEC
globals N: wr Int FN: wr Int
pre n ⩾ 0
post fn' = n

a design can be sought with the aid of IWHILEDN. An obvious aim for the "rest" of the iterations is to compute:

10 fn' = n! * fn

Which suggests a program:

11 FACT = [[FN:=1; while N ≠ 0 loop BODY endloop]]

Thus RESULT becomes:

i n' = n ∧ fn' = 1 ∧ fn" = n'! * fn' ⇒ fn" = n!

and FALL:

ii $n = 0 \Rightarrow fn = n! * fn$

A pre-condition can be sought for BODY by considering the domain rules. Thus:

iii preBODY as $n > 0$

iv invl as $n \geq 0$

and DOMINIT, DOMLOOP, DOMX and DOMBODY are all immediate.

A definition for postBODY which composes with rest is:

v $fn' = fn * n \wedge n' = n-1$

Thus COMP becomes:

vi $fn' = fn * n \wedge n' = n-1 \wedge fn'' = n'! * fn' \Rightarrow fn'' = n! * fn$

The termination argument is identical with vii-ix of 8. Finally, it is trivial to show that:

12 ⟦FN:=FN * N; N:=N-1;⟧ sat (S1,preBODY,postBODY)

It is a long-established tradition to overuse the factorial example and it will have to serve as a first introduction to the IWHILEFIX rule. Taking 9 as a specification, and 11 as a tentative design, an obvious expression to hold constant is:

13 i $fn * n!$

The requirement (CONSTANCY) to leave the value of this expression unchanged can be viewed as a different form of specification for BODY. The RESULT condition becomes:

ii $fn' = 1 \wedge n' = n \wedge fn'' * n''! = fn' * n'! \wedge n'' = 0 \Rightarrow fn'' = n!$

The domain arguments are as in 11; as is the STOP condition. The DECR condition provides the other part of the "specification" for BODY. It is obvious that the code in 12 meets the two given requirements.

This example does less than justice to the IWHILEFIX rule. A far more interesting challenge is to provide a perspicuous development of the division algorithm used by mechanical calculators. (This example was used by Tony Hoare in the Oxford M.Sc. course to illustrate weakest pre-condition proofs, Hoare /80a/.) The specification is:

14 IDIV
globals A:wr Nat0 B:wr Nat Q:wr Nat0
post $b * q' + a' = a \wedge a' < b$

The first part of the algorithm shifts B until it is larger than A, the number of decimal shifts is recorded in N; in an iteration, controlled by N, successive subtractions are then done. Thus it is required to show:

15 $[\![$ FIRST; while N $\neq$ 0 loop BODY endloop $]\!]$ sat $[\![$IDIV$]\!]$

With:

16 FIRST
globals B:wr Nat Q:wr Nat0 N:wr Nat0
post $b' = b * 10 ** n' \wedge a' < b' \wedge q' = 0$

17 BODY
globals A:wr Nat0 B:wr Nat Q:wr Nat0 N:wr Nat0
pre $n \neq 0$
cons $b * q + a,\ b/10 ** n$
var n

(Here the idea of recording the specification of BODY in a revised form has been fully implemented.) With invl as:

i $a < b \wedge$ isdivisor $(b, 10 ** n)$

the domain conditions of IWHILEFIX are immediate. The RESULT condition becomes:

ii $a' = a \wedge q' = 0 \wedge b' = b * 10 ** n' \wedge$
$b'' * q'' + a'' = b' * q' + a' \wedge b''/10 ** n'' = b'/10 ** n' \wedge$
$n'' = 0 \wedge a'' < b'' \Rightarrow$
$b'' = b \wedge b'' * q'' + a'' = a \wedge a'' < b''$

from which:

iii $b * q'' + a'' = a \wedge a'' < b$

is immediate. Termination is also trivial.

The problem of implementing code for the initial shifting gives rise to:

18 [[N:=0; while B ≤ A loop B:=B * 10; N:=N + 1; endloop; Q:=0]]

IWHILEFIX can be used to prove this step correct with:

i consBODY as $b/10 ** n$

The RESULT rule becomes:

ii $n' = 0 \wedge b' = b \wedge$
$b''/10 ** n'' = b'/10 ** n' \wedge a < b'' \Rightarrow$
$b'' = b * 10 ** n'' \wedge a < b''$

Of more interest is the one remaining task: the realisation of BODY. The basic idea is to shift B and subtract it from A keeping

track of the consequent changes in Q and N. Thus to show:

19 ⟦ N:=N-1; B:=B/10; Q:=Q * 10;
while B ≤ A loop BBODY endloop ⟧ sat ⟦BODY⟧

Use:

20 BBODY
globals A:wr Nat0 B:rd Nat Q:wr Nat0
pre b ≤ a
cons b * q + a

The only non-trivial domain rule is DOMINIT which becomes:

i $n \geq 0 \wedge n \neq 0 \wedge$ isdivisor (b, 10 ** n) ⇒
$n > 0 \wedge$ isdivisor (b,10)

The preservation by 19,20 of 17 is obvious.

Finally noting:

21 ⟦ Q:=Q+1; A:=A-B ⟧ sat ⟦BBODY⟧

permits collection of the whole algorithm:

```
22  N:=0;                    - - FIRST
    while  B ≤ A loop
        B:=B * 10;
        N:=N+1;
    endloop;
    Q:=0;                    - - FIRST

    while  N ≠ 0 loop
        N:=N-1;              - - BODY
        B:=B/10;
        Q:=Q * 10;
        while  B ≤ A loop
            Q:=Q+1;          - - BBODY
            A:=A-B;          - - BBODY
        endloop;             - - BODY
    endloop;
```

It is interesting to compare the expressions which arise in proofs using the various methods. The addition by successor example in Jones /80a/ was proved using a rule similar to IWHILEDN with a "rest" predicate:

23 $r' = x + y$

It is probably easier to spot the constant expression:

24 $y + r$

The inductive assertion method would require the use of a free variable since the initial value of Y is overwritten:

25 $r = x + (y_0 - y)$

Furthermore, the "multiplication in logarithmic time" example requires, with IWHILEDN, two different "rest" conditions.

26 $r' = r + x * y$ outer

27 $r' + x' * y' = r + x * y$ inner

Whereas, using IWHILEFIX the constant expression:

28 $r + x * y$

suffices for the whole proof.

Examples of program proofs in the literature rarely overwrite their initial values. It is claimed in section 3.5 that this is avoided largely because of a deficiency in the inductive assertion method. However, for programs in which temporary variables are used as counters (in order to leave initial values intact), the IWHILEUP rule can also be used. Reverting again to the specification in 9, the use of a temporary variable might suggest that:

29 $fn = c!$

could be preserved. In Jones /80a/ this was used as a clause of the "front" condition. Here, the importance of recognising the data type invariant of the loop is shown by absorbing 29. This also makes an interesting comparison with the inductive assertion method. Thus:

30 $[\![$FN:=1; C:=0; while N ≠ C loop BODY endloop$]\!]$ sat SPEC

with:

31 BODY
globals FN:wr Int C:wr Int
post $c' = c+1 \wedge fn' = fn * c$

The domain conditions are all immediate. Then with:

i invl as $0 \leq c \leq n \wedge fn = c!$

ii front as $n' = n$

BASIS becomes;

iii $n' = n \Rightarrow n' = n$

COMP:

iv $n' = n \wedge n'' = n' \Rightarrow n'' = n$

RESULT:

v $fn' = c'! \wedge n' = n \wedge c' = n' \Rightarrow fn' = n!$

The termination proof is done with:

vi var as $n-c \in$ Nat0

Notice that 30 could be reformulated so as to be proved by FORTO.

If an inappropriate choice is made between "up" and "down" rules, the corresponding invariant relation is likely to be more cumbersome than need be. For example, if 30 is tackled with the IWHILEDN rule, a rest predicate (cf 31 ii) results:

32 $fn' = fn * n!/c! \wedge n' = n$

Similarly if 11 is to be proved correct via IWHILEUP the simple invariant in 10 must be replaced by:

33 $fn' = n!/n'!$

Since it is one of the new rules, it is worth illustrating the use of FORARB although it is also used in section 5.3. Consider the following specification:

SETDIF (S: El-set)
globals T:rd El-set R:wr El-set
post $r' = r \cup (s-t)$

Clearly each element of S must be considered and the order in which this is to be done is arbitrary:

for $e \in S$ loop
    if $e \notin T$ then $R := R \cup \{e\}$; endif
endloop

can be proved with:

i preB as $e \in s$

ii postB as $r' = r \cup (\{e\} - t)$

Two closing points: there is a discussion in Jones /80a/ of a design guideline called "active decomposition". This carries over to the new rules and has been quietly adhered to here. Although not illustrated in this section, it is a natural consequence of the definition of sat that one piece of code can satisfy many specifications. This point will recur with parallel solutions in section 5.3.

3.5 Alternatives

This section resumes the comparison begun in sub-section 1.6.1.

3.5.1 On Post-Conditions

The general idea of using pre- and post-conditions for the specification and justification of sequential decomposition is widely accepted. The detailed decisions give rise to some divergence of opinion. Hoare /69a/ and Dijkstra /76a/ (and, thus, nearly all other published work in this area) use post-conditions of one state. (They also tend to view the proof rules as a definition of the language, but this point is less important.) The alternative adopted here of using a post-condition of both initial and final states means that the proof rules of sub-section 3.2.2 are much longer than Hoare's axioms or the relevant predicate transformers. The brevity of the latter rules results from punning in two ways. Because pre- and post-conditions have the same type, they can be used interchangeably. In addition, since only one set of values is of concern to a predicate, the names of the variables can be used in the assertions to denote the "current value". It is a major achievement that this whole system works. The result is a set of very brief proof rules which are easy to remember. (Tony Hoare pointed out that one of the advantages of the relational form of the proof rules given in sub-section 3.2.1 is that they are easier to memorise.)

How can the decision to use predicates of two states and the consequentially longer proof rules be justified? Three arguments are given:

i) They yield more natural specifications

ii) They are more useful for large problems

iii) The separate conditions provide a useful check-list.

These claims are presented in relation to the predicate transformer approach.

Consider the specification:

1 FACT
globals N: rd Nat0 FN: wr Nat
post fn' = n!

This might be realised by:

2 FACT = [FN:=1; C:=0; FACT B]

3 FACT B = [while N ≠ C loop FN:=FN * C; C:=C+1; endloop]

An obvious post-condition of the final state alone is:

4 fn = n!

But this fails completely to prohibit:

5 FACT = [FN:=6; N:=3]

The alternative to 4 is to use either free variables:

6 wp(FACT, fn = n_o!) ⇔ n = n_o

or free predicates:

7 wp(FACT, p(fn)) ⇔ p(n!)

The use of the free variables is error prone where many predicate transformers are to be manipulated and pinpoints the obvious requirement for a proper way of referring to the values in the initial state. The use of a free predicate (in 7) to fix equality appears to be unnecessarily high-level and also to put the key information (factorial) in the wrong place (i.e. in the pre-condition).

Nor is the above argument the only place where predicate transformers cause an unnatural expression of a specification. The concept of a weakest pre-condition prompts the question: to which post-condition should this relate? Which of the following is the "right" specification for FACT B?

8 $wp(\text{FACT B}, fn = 1 \wedge c = 0 \wedge n = n_o) \Leftrightarrow fn = n_o!$

9 $wp(\text{FACT B}, fn = fn_o \wedge c = 0 \wedge n = n_o) \Leftrightarrow fn = n_o! * fn_o$

10 $wp(\text{FACT B}, fn = 1 \wedge c = c_o \wedge n = n_o) \Leftrightarrow fn = n_o! / c_o!$

11 $wp(\text{FACT B}, fn = fn_o \wedge c = c_o \wedge n = n_o) \Leftrightarrow fn = n_o! / c_o! * fn_o$

In larger problems it is necessary to use several stages of development. The style normally adopted with predicate transformers is that of 4. There is also some mention made of the fact that N should not be overwritten (cf. glocon of Dijkstra /76a/ or 1). Programs like those in 3.4 (3,4) are avoided precisely because they do overwrite their input values. This is acceptable on factorial; it leads in Hoare /80a/ to a confusing description of the mechanical calculator rule for division because of the impression that more registers must be available (cf. 3.4 (4-22)); for examples where

larger data structures are involved such a constraint is unacceptable.

The third argument in favour of the use of post-conditions of two states is the use of the conditions of the proof rules as a check-list. This is illustrated in section 3.4 and chapter 5. It would be interesting to construct an interactive program development system around these check-lists.

What overall conclusion can be drawn from this comparison? The Hoare axioms originated with the task of proving extant programs. For this problem they work very well: the "punning" leads to a natural style of annotating a program (cf. King /76a/). The weaknesses of the post-conditions of state pairs (e.g. the need to use primes and their clumsiness as annotations) do not matter so much on larger multi-stage developments and the advantages claimed above then outweight the disadvantages. If the standard predicate transformer scheme is used, then the temptation to save initial values in the state solely for the purposes of the proof should be resisted. (This point occurs again in connection with parallelism.)

It is interesting to note that the construction of a range of systems which handle post-conditions have often adopted the pair approach (e.g. Wulf /76a/, Randell /78a/, Hantler /75a/).

Another interesting approach to the definition of which variables are held constant is the specification logic of Reynolds /81a/. The assertion:

12 Stmt # Expr

is taken to mean that the given statement does not alter any variable in the given expression. (This is not quite the same as the concept of a constant expression in 3.2.2 (IWHILEFIX).) The difficulties of using assertions of the form of 12 as annotations are similar to those of using post-conditions of pairs of states.

3.5.2 Technical Points

The denotational semantics in section 3.1 is based on relations over St. In de Bakker /73a/ the domain of states is extended by adding a bottom element ($\perp$) and all of the combinators are then made strict. Although the semantics here could be redefined on this basis, the given view is preferred because of the simpler ordering (cf. discussion of Smyth ordering in section 1.2).

3.5.3 Possible Extensions

Proof rules for justifying the use of inner blocks etc. should be easy to define with the recognition of the states involved. In fact, such a rule is relied on informally in 3.4 (30-31). Rules for (recursive) procedures might be based on Hehner /79a/. The problem of abnormal termination of loops and appropriate proof rules has been considered in Bron /76a/, Bron /77a/. The "Meta-IV" exit/tixe construct (Jones /78b/) could be the basis of such extensions.

The "intermittent assertions" (Burstall /74a/, Manna /78a/) approach has recently produced some controversy. In spite of the arguments in Gries /79a/, it would appear that there is a case that such proofs are more natural for some programs. It is likely that

the summation operators required in normal inductive assertion proofs could be avoided by appropriate interpretation of predicates of two states. However, some work in this direction has failed to yield proof rules of wide generality.

Chapter 4

Development of Interfering Programs

The preceding chapters provide a basis which greatly eases the task of introducing the notion of interference. Particularly important is the fact that specifications are given by predicates of pairs of states. The decision to do this for sequential (non-interfering) programs is defended in section 3.5; it has also been invaluable in developing the ideas presented in this chapter.

The basic problem to be faced is that programs which do not run (or appear to run) in isolation may interfere with each other. This chapter concentrates on interference perceived through shared variables; chapter 6 reviews the connections with interference by communication. With the possibility of interference, the extensional view of the effect of an operation is inadequate: the effect of a parallel set of tasks can not be deduced from their post-conditions alone. One approach adopted in the literature is to prove first the separate operations correct in isolation and to subsequently establish non-interference. The position taken in chapter 6 is that this approach is unacceptable for a development method. Not only would little guidance be given in the design process, but also erroneous design decisions might remain undetected until after much work had been based on them.

The proof rules given in chapters 2 and 3 represent the results of an evolution brought about by application to many examples. Although several different versions have been tried, the proof rules in section 4.4 are relatively new. In this sense they must be considered to be less stable than the proof rules concerning sequential program development. The proof rules concerned with the notion of

interference are, however, justified with respect to an underlying semantics. Here an operational semantics is chosen and this decision is discussed in section 4.3 and chapter 7.

The key proposal in this chapter is to face the problem of interference throughout the development process; to make a place for it in the specification format; to recognise that it must be checked at any design step. The details of how this is currently done should be viewed more as an existence proof than as a final proposal for a development method. Indeed, the specific proof rules given here are restrictive in many ways (e.g. synchronisation, as such, has not been covered). Chapter 7 gives some indication of how extensions might be made.

For many problems, the overall specification makes no mention of interference. Basically, parallelism is used as an implementation technique - often to improve (potential) performance. Clearly a proof rule is required which copes with a decomposition into a family of processes which are to be executed in parallel. In order to prove such a decomposition correct, the component operations must be specified: it is these sub-specifications which must define and constrain interference.

The examples given in this chapter are small. As is mentioned elsewhere, a development method can only be judged on meatier problems and chapter 5 goes some way to fill this need.

4.1 Extensions to Specification Format

The view taken of specifications and denotations of operations in the absence of interference is basically that of relations over states. In the presence of interference this view is insufficient. Consider an example in which the two following sequences of statements are executed in parallel (assume X shared, T and U locals):

i T:=X; X:=T + 1;

ii U:=X; X:=U + 2;

For the first:

iii $x' = x + 1$

and for the second:

iv $x' = x + 2$

However, in parallel the result will be:

v $x' \in \{x + 1 \,..\, x + 3\}$

Furthermore, interference will bring with it a concern for the level of atomicity. Thus, if single statements were to be viewed as atomic, the effect of the two sequences:

vi T:=X; X:=T + 1

vii X:=X + 1

when executed in parallel with other processes sharing X, might differ.

The approach to specifications here is to retain the extensional view given by pre- and post-conditions but to add appropriate

specifications of the interference. In analogy to the extensional view, a rely-condition records the assumptions made and a guarantee-condition defines the commitments made.

As is mentioned above, the fact that relations permit more than one result (under-determined) may cover implementations in which non-determinism results from interference. In fact, sequential programs sometimes determine a specific result and thus resolve the freedom (e.g. square root) whereas parallelism can give rise to non-deterministic methods of finding a unique result (e.g. section 5.2).

The first part of specifying interference is the rely-condition. The intent is to record the interference which can be tolerated while still promising to meet the operation's commitments. Speaking operationally, the process whose specification is being given can no longer assume that the state remains untouched other than by its own assignments. It can, however, assume that if state s is changed to s' by some other process then this pair of states will satisfy the rely-condition. Examples of rely-conditions might be:

1 $x' = x$ the value of variable X is unchanged

2 $x' + y' = x + y$

3 lock $\Rightarrow$ ...

4 pointer chains remain linked

5 $x' \leqslant x$ X decreases monotonically

6 a list remains ordered

A rely-condition, then, is a relation on states; it is assumed to be total since this avoids the need to introduce a new subclass of states (cf. inv1); it must also be reflexive since "no change" is obviously a possibility. Where there are to be more than two processes, it is also necessary that the rely-condition be transitive. (Examples follow the logic form of the specification below.)

One of the results to be proved in a step of design which involves parallelism will now be the coexistence of processes. In order to check that their mutual interference is acceptable at the design stage, the specification of each process must also contain a guarantee-condition which defines the effect it can have on the state in one of its atomic transitions. Guarantee-conditions will also be total, reflexive relations over states. Notice that the guarantee-condition defines a single step and does not, as such, have to cover the interference which impinges on a process. However, such interference may influence what actual transition is made.

The format of a specification is then:

7 OP name
St set of states
pre $\mathcal{P}$(St)
post St x St
rely St $\leftrightarrow$ St
guar St $\leftrightarrow$ St

This is normally given via predicates:

8 OP
globals ...
pre St → Bool
post St x St → Bool
rely St x St → Bool
guar St x St → Bool

9 $(\forall s \in St;\ (\exists s' \in St;\ relyOP(s,s'))$

10 $(\forall s \in St;\ (\exists s' \in St;\ guarOP(s,s'))$

(There is here again a systematic rule for naming conditions when they are taken from the context of their specifications.)

The technique of specifying acceptable states by recording the global variables (cf. section 1.4) offers a great advantage: knowing that, for example, X is only available for reading says more than a post-condition that x' = x. The use of globals simplifies both post- and guarantee-conditions in that variables to which no explicit write access has been given cannot be changed.

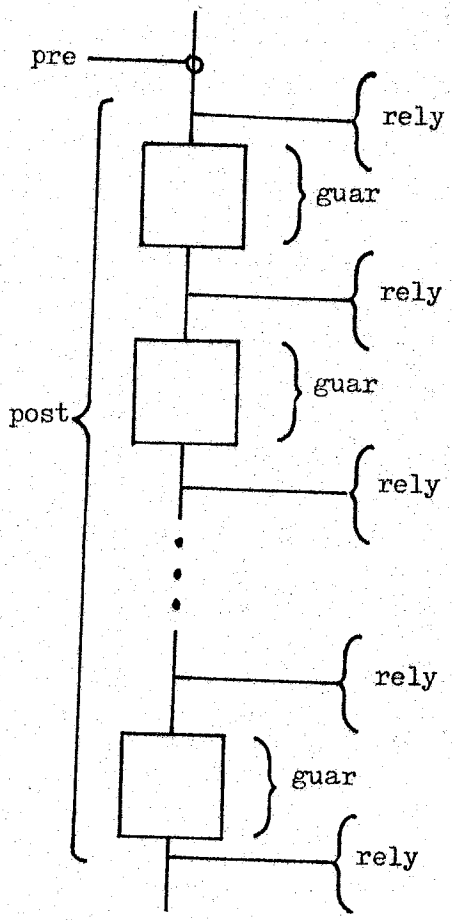
The individual parts of 8 can now be reviewed. The pre-condition is unchanged and the discussion in section 1.4 covers its interpretation. The post-condition should not be "completed" (cf. section 1.4) in the presence of interference: a single process cannot make any statement about the constancy of variables to which other processes might have access. The need to reason about such properties gives rise to the "dynamic invariant" in sub-section 4.4.1. The meaning of identifiers corresponding to global variables is as above (i.e. unprimed lower case identifiers denote the value

of the variable of the same name before execution and primed identifiers denote the value after execution). Looking ahead, proofs about the properties of a set of parallel tasks are able to rely on the conjunction of the post-conditions.

The rely-condition is also not subject to "completion" since there is no need to rely on properties of variables which are not visible. The x/x' etc. in a rely-condition denote values in two states which arise during any period in which the process to which the rely-condition belongs makes no changes.

The guarantee-condition <u>is</u> completed since this shows the lack of write access by a process. Here x/x' etc. denote values in any two states which can arise by executing an uninterrupted (sequence of) step(s) of the process in question.

These comments can be considered with the following diagram:



Notice that, in developing a program to such a specification, the possibility of interference before the first and after the last steps of the written code must be considered.

As a first example of a specification in the form of 8, consider the following:

11 P1
globals X:wr E1
post x' = f(x)
rely x' = x

12 P2
globals Y:wr E1
post y' = g(y)
rely y' = y

It should be possible to use these two processes in parallel: the completion of the guarantee-condition will indicate in each case that one process will not interfere with the other. In order to deduce that the effect of their parallel execution yields the obvious result, P1 and P2 must be executed in an environment which itself ensures that X and Y are unchanged.

The example of 11 and 12 is really one of non-interference. Such problems do, however, arise naturally in the design of highly interfering programs. For example, in section 5.3 there is a procedure ROOT which searches for the root of a (representation of a) tree. Code which computes in parallel:

13 R1:=ROOT (E1)
R2:=ROOT (E2)

can be shown to satisfy:

14 $r1' = root(f,e1) \wedge r2' = root(f,e2)$

because the ROOT procedure has only one global access and that is the ability to read the array F - there is also a rely-condition on the whole procedure being developed that:

15 $f' = f$

Suppose, for the next example, that two programs exist for the same task. If their performance varies for different initial values, it might be worth executing them in parallel. Of course, as soon as one process has computed the desired result the other process can terminate. The fact that the result has been found could be signalled by setting a Boolean variable (DONE) to TRUE. A corresponding clause must be conjoined to the post-condition. Suppose now that process P1 makes a premature assignment to DONE and then withdraws its notice by resetting DONE to FALSE. If P2 happens to see this change it might terminate prematurely. The result might be that at least part of its post-condition would not be fulfilled. The following rely- and guarantee-conditions express the requirements for the processes to coexist in an honest way:

16 Pi

globals X:rd El Y:wr El DONE:wr Bool

post $y' = f(x) \wedge done'$

rely $done \Rightarrow done' \wedge x' = x \wedge (y' = y \vee y' = f(x))$

guar $done \Rightarrow done' \wedge (y' = y \vee y' = f(x))$

The clause:

$done \Rightarrow done'$

prevents a process changing the value of DONE from TRUE to FALSE. Assuming the two processes are run in an environment in which X and Y cannot be changed, an obvious overall post-condition can be established.

An extended version of the same technique can be illustrated by the following example. Suppose it is required to find some value $i \in I$ which enjoys both properties $p: I \to Bool$ and $q: I \to Bool$. If these properties are independent, the task of examining each can be given to different processes. Each process will record its findings in an array (PV and QV respectively). If a value of I is found which satisfies P and for which the Q task has already set the corresponding QV index, then the variable R can be set to this value of I. Thus to find:

17 $i \in I$ st $p(i) \wedge q(i)$

execute PT and QT in parallel, where:

18 PT

globals PV:wr I $\xrightarrow[m]{}$ Bool QV:rd I $\xrightarrow[m]{}$ Bool R:wr I

pre $(\exists i \in I;\ p(i) \wedge q(i))$

post $p(r') \wedge q(r')$

rely $qv(i) \Rightarrow qv'(i)$

guar $(pv(i) \Rightarrow pv'(i)) \wedge (\sim pv(i) \wedge pv'(i) \Rightarrow p(i)) \wedge$
$(r' \neq r \Rightarrow pv'(r') \wedge qv'(r'))$

19 QT

globals PV:rd I $\xrightarrow[m]{}$ Bool QV:wr I $\xrightarrow[m]{}$ Bool R:wr I

pre ($\exists$i $\in$I; p(i) $\wedge$ q(i))

post p(r') $\wedge$ q(r')

rely pv(i) $\Rightarrow$ pv'(i)

guar (qv(i) $\Rightarrow$ qv'(i)) $\wedge$ (qv'(i) $\Rightarrow$q(i)) $\wedge$
(r' $\neq$ r $\Rightarrow$ pv(r') $\wedge$ qv(r'))

A more interesting example can be taken from Hoare /75a/. The idea is to compute the primes (up to some given N) using Eratosthene's "sieve" with two instances of a REMOVE process executing in parallel. Thus:

20
```
SIEVE:= {2 .. N} ;
A:=2;
B:=3;
while  A**2 ≤ N loop
     (REMOVE(A) || REMOVE(B));
     if  B**2 < N then A:=mins( { i ∈ SIEVE st B < i } );
     else   A:=B;
     endif;
     if  A**2 < N then B:=mins( {i ∈ SIEVE st A < i } );
     endif;
endloop
```

Here, only REMOVE is of interest.

21
```
REMOVE procedure (X:in Nat);
   globals  SIEVE:wr Nat-set
```

In a sequential solution, postREMOVE might be:

sieve' = sieve - {m*x st m $\in$ Nat}

but since other instance(s) of REMOVE might also be removing elements from SIEVE, this is not correct. An obvious (weak)

post-condition is:

22 $(\forall m \in Nat;\ m*x \notin sieve')$

But this, alone, is too weak - a possible implementation would be:

23 SIEVE:= $\{\}$

In order to prohibit such implementations a guarantee-condition can be used:

$c \in sieve \wedge c \notin sieve' \Rightarrow (\exists m \in Nat;\ m*x = c)$

Can the post-condition in 22 be realised in the presence of arbitrary interference? In fact, no! It relies on the fact that once an element has been removed, no other process reinserts it. Thus there is a rely-condition (and consequent clause of the guarantee-condition) that:

24 $c \in sieve \vee c \notin sieve'$

Summarising 21 to 24 yields:

25 REMOVE procedure (X:in Nat);
globals SIEVE:wr Nat-set
post $(\forall m \in Nat;\ m*x \notin sieve')$
rely $c \in sieve \vee c \notin sieve'$
guar $(c \in sieve \wedge c \notin sieve' \Rightarrow (\exists m \in Nat;\ m*x = c))$
$(c \in sieve \vee c \notin sieve')$

Notice that the conjunction of the post-conditions will not now be enough to give an overall post-condition. This is one of the cases where a dynamic invariant (cf. sub-section 4.4.1) is required to complete the overall correctness proof.

There are a number of special cases of 8 which are worth recognising. A rely-condition of $s' = s$ implies that an operation is run in an atomic fashion. Leading on from this, the defaults are a rely-condition of $s' = s$ and a guarantee-condition of TRUE. This links the form in 8 to the sequential case in 1.4(2). Two processes with incompatible rely-/guarantee-conditions cannot be run in parallel: some technique must be used to ensure their mutual exclusion.

4.2 Realisation

Given an extended notion of specification, it is necessary to modify the notion of realisation given in section 1.2. For a realisation with identical rely- and guarantee-conditions, the sat relation given above suffices. How can the interference constraints change? In analogy with 1.4(WKNPRE/STRPOST) it should be clear that it is acceptable for a putative realisation to rely on less or guarantee more than is stated in its specification. Thus:

1 (S', R', G') sat $(S, R, G) \Leftrightarrow$
$(\text{dom}S); S' \text{psat } S \wedge S' \text{defover dom } S \wedge$
$R \text{ psat } R' \wedge G' \text{ psat } G$

For example, in the specification of REMOVE in 4.1(25) a weaker rely-condition could be given by limiting, with a bounded quantifier, the assumption to the elements about which the post-condition makes a requirement. Similarly, in 5.2.4(4) the rely-condition need only be concerned with the part of X indicated by the set of indices in MINE.

In comparing 1 with 1.2(10) it should be remembered that the rely- and guarantee-relations are total.

4.3 Operational Semantics for Parallelism

Language design for parallelism is a large subject in its own right. In Owicki /75a/ both a "General" and a "Restricted Programming Language" are offered. This reflects a commonly accepted fear that parallel programming languages which are insufficiently constrained make Edsger Dijkstra's original concerns about the goto statement pale into insignificance. The trend in language design has been in the direction of providing more and more structure. Thus Dijkstra /68a/ used semaphores; critical regions were introduced in Brinch Hansen /73a/; Hoare /74a/ discusses monitors; and even the permissible paths are constrained by CSP (cf. Hoare /78a/). This degree of bundling may not be wise for all problems although it must be obvious that only by imposing structure does it become practical to reason about systems. The activity in the language design area indicates that it would be premature to focus attention on one particular approach here. By concentrating on the central concept of interference the comments will, to some extent, become more general. The only assumption made is that there is some way of executing a number of operations in parallel.

The programs presented below use the syntax of Ada. This is no more an indication of a wholehearted support for that language than was the similar decision to use PL/I in Jones /80a/. In comparison to the Pascal derivatives which cope with parallelism, the "rendezvous" concept of Ada (cf. Hoare /78a/ on CSP) is one of the better thought out new ideas in Ada. In fact, as indicated above,

the reasoning at the code level will be relatively informal. The interest here is to focus on the central issue of interference.

As observed in section 4.1, the extensional view of operations is not adequate to provide a basis from which the semantics of their parallel combination can be deduced. This presents a severe problem for the provision of a denotational semantics. The conclusion accepted by some researchers in this area is that the history must be recorded in the denotation by, for example, using resumptions:

1 $Res = St \rightarrow \wp(St \times Res)$

The parallel combination can then be built up by some form of non-deterministic merge of the steps. There are, however, problems in achieving "full abstraction" (cf. Hennessy /80a/).

The distinction between operational and denotational semantics is frequently exaggerated. For example, the "functional semantics" of Allen /72a/ indicates that the step to a "small state" is more important than the decision to use functional denotations. With a resumption style semantics the two methods are drawn even closer together. What remains in the denotational definition is the need to solve ordering problems (cf. Plotkin /76a/, Smyth /78a/, Scott /81a/) in order to support recursive definitions.

The decision made here is to present an operational semantics in terms of which the proof rules of the next section can be justified. It is, however, clear that the justification used in

section 3.3 is much more formal than that in section 4.5. The operational definition given uses a "control tree"-like concept as in Walk /69a/ (cf. Lucas /68b/ and Owicki /75a/).

The concrete syntax (cf. 3.1.1(5)) of the language to be defined is:

2 Program::= Stmt

Stmt::= Composition|Guardedif|While|Par|Basic

Par::= (|| Bag (Stmt))

The semantics is given by a relation over extended states. An extended state is a pair containing a Control and a state as in sub-section 3.1.1. A control contains a bag of components:

3 Control = Bag (Component)

4 Component:: [Stmt] Control

The nesting of Control within Control represents the calling sequence; the branching represents the potential for parallel (non-deterministic merge) execution. At any point in time, the collection of potential next steps are represented by the leaf nodes (i.e. a component with an empty bag as Control[+]). This implies that execution of the Basic statements is atomic and since this is not part of the central theme here, this consequence will be accepted. The semantic relation:

5 (Control x St) x (Control x St)

is presented by (conditional) rules of the form:

6 (c,s) ↔ {(c',s') st ... }

[+] stmt will be used for (stmt, emptybag())

where c is a leaf component of the control and s is the current St; c' is placed in the control in place of c and s' is the "next" element of St. For:

7 prog $\in$ Program

begin with:

8 co = (STOP, bag(prog))
so = {id → ? st id is an identifier of prog }

For sequential compositions it is only necessary to set up the appropriately nested control tree:

9 (P1; P2, s) ↔ {((P2, bag(P1)),s)}

For conditional statements:

10 IF = if e1 → P1 [] ... [] en → Pn fi

providing no ei can lead to abort, a truly non-deterministic choice is represented by a many-many relation:

11 (IF,s) ↔ {(Pi,s) st eval(ei,s)}

For iterative statements:

12 WH = while e loop B endloop

the standard expansion is used:

13 if eval (e,s) then
(WH,s) ↔ {((WH,bag (B)),s)}
else
(WH,s) ↔ {(NIL,s)}

For a parallel statement:

14 PAR = ($\|$ P1 ... Pn)

a bag of components is placed in the control (notice this is a single multi-element bag unlike the conditional which defines a set of possible single component bags):

15 $(\underline{PAR},s) \leftrightarrow \{((\underline{NIL}, bag(P1, ..., Pn)),s)\}$

Finally, to illustrate a basic statement:

16 $(\underline{v:=e},s) \leftrightarrow \{(\underline{NIL}, s \dagger \{v \mapsto eval(e,s)\})\}$

A leaf node of the control with a NIL statement is replaced by an empty bag.

4.4 Proof Rules

The basic task here is to present a proof rule for the justification of the parallel construct of the language. It is also necessary to review the effect of interference considerations on the rules of section 2.2 and sub-section 3.2.2 since it is obviously necessary to retain these tools for program development.

4.4.1 Parallel Combination

In order to show that:

1 $[\![(\| P1 \ldots Pn)]\!]$ sat a specification SPEC given by T,M,R,G

it is necessary to establish both that the overall effect is as required by $T[\![SPEC]\!]/M[\![SPEC]\!]$ and also that the interference considerations for the components match. As in chapter 3, the rule is first presented via relations and then the predicate form (PARCOOP) is given (in a denser style). The latter is the one normally used in justifying development steps. The first collection of conditions requires that the domains of the individual processes match the specification:

DOMi $[\![Pi]\!]$ defover $T[\![SPEC]\!]$

(Remember that the development of Pi must accept the possibility of interference even at the very beginning of its execution.)

The overall effect of executing the processes will be the intersection of their individual effects. As is pointed out above, however, the need to develop the processes under the assumption of

interference will mean that this is frequently too little or, in other cases, can only be made strong enough by an overly-complicated set of rely-conditions. The resolution of this problem is to bring in a dynamic invariant. The rôle of the dynamic invariant can be compared with that of invl in proofs of iterative constructs and, as in sequential development, the dynamic invariant is also a useful aid in choosing a design. The dynamic invariant (DINV) is a relation over two states: the starting state and any other that can arise during execution of the parallel processes. Thus:

RESULT $DINV \cap \underline{int}\ \{ M[\![Pi]\!]\ \ \underline{st}\ 1 \leq i \leq n \}\ \ \underline{psat}\ \ M[\![SPEC]\!]$

Notice that although the use of $M[\![Pi]\!]$ suggests an extensional view of the processes, this has to be established with recognition of interference (as defined by the rely-condition).

The dynamic invariant obviously has to be shown to hold. The initial condition is:

INVBASIS $T[\![SPEC]\!]\ ;\ E \subseteq DINV$

The guarantee-conditions of the individual processes show what transitions are possible. Thus, for each process, it is necessary to show:

INVPRESi $DINV;\ G[\![Pi]\!] \subseteq DINV$

There is also the danger that some process in the environment of the whole parallel construct could disturb the state. The rely-condition of the specification can be used to establish:

INVPRESENV $DINV;\ R[\![SPEC]\!] \subseteq DINV$

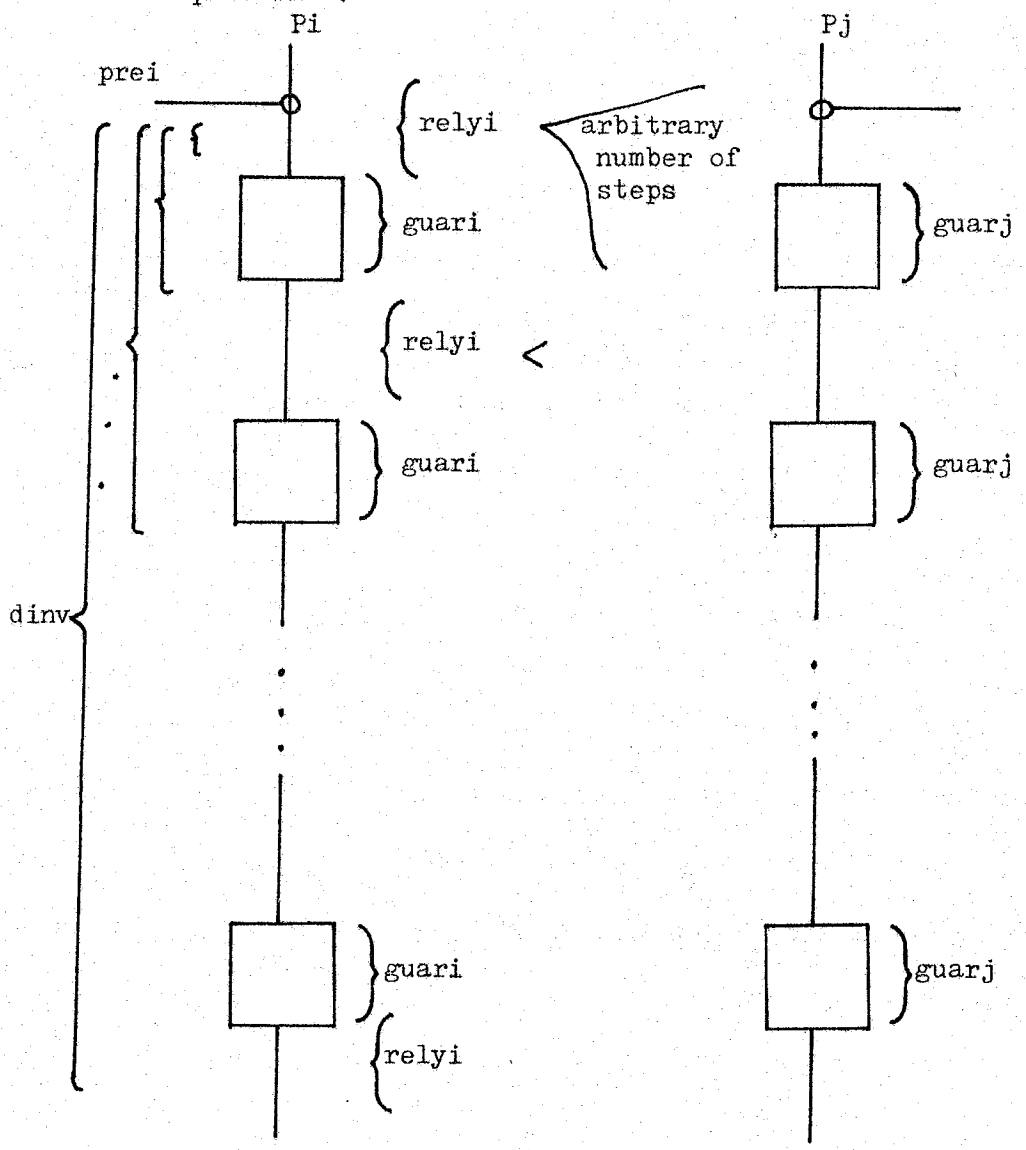
Finally, it is necessary to establish that the processes co-exist harmoniously both with each other and with the environment:

INTERFEREi $\bigcup_{j \neq i} G[\![P_j]\!] \subseteq R[\![Pi]\!]$

INTERFEREENV $R[\![SPEC]\!] \subseteq R[\![Pi]\!]$

(The guarantee-condition of the specification has been assumed to be TRUE - cf. sub-section 4.4.3.)

The diagram of section 4.1 can be extended to show two processes:



The interference matching must be established in both directions. The rely-condition can be thought of as the post-condition of an interfering operation which is slotted between any two atomic steps of a process. For this reason the rely-condition must be inherited in the development process (cf. sub-section 4.4.3).

The predicate form of 1 is:

PARCOOP $[\![(\| P1 \ldots Pn)]\!]$ sat (St,pre,post,rely,guar)

if with:

dinv: St x St $\rightarrow$ Bool

the following conditions hold:

DOMi $(\forall s \in St$ st pre(s); prei(s))

INVBASIS $(\forall s \in St$ st pre(s); dinv(s,s))

INVPRESi $(\forall s,s',s'' \in St;\ dinv(s,s') \wedge guari(s',s'') \Rightarrow dinv(s,s''))$

INVPRESENV $(\forall s,s',s'' \in St;\ dinv(s,s') \wedge rely(s',s'') \Rightarrow dinv(s,s''))$

INTERFEREi $\bigwedge_{j \neq i} (\forall s,s' \in St;\ guarj(s,s') \Rightarrow relyi(s,s'))$

INTERFERENV $(\forall s,s' \in St;\ rely(s,s') \Rightarrow relyi(s,s'))$

RESULT $(\forall s,s' \in St;\ dinv(s,s') \wedge \bigwedge_i posti(s,s') \Rightarrow post(s,s'))$

As a first example of the use of the PARCOOP proof rule a simplified version of the problem of section 5.2 is considered. Suppose it is desired to reduce the value of some variable X to the minimum of its initial value and any element of S which satisfies some predicate p:

2 $x' = mins(\{n \in S$ st $p(n)\} \cup \{x\})$

Two subsets of the set S could be sought such that they cover S (it is not necessary that {S1,S2} is a partition):

3 $S1 \cup S2 = S$

Then the type of the processes is:

4 Pi (Si:in Int-set)
globals X:wr Int

If Pi were to be considered in isolation it would be possible to define its final value precisely; in the presence of interference only a bound can be given. Thus postPi is:

5 $(\forall n \in Si;\ p(n) \Rightarrow x' \leqslant n)$

If the fellow process were to cheat by setting X to some low value then subsequently recanting, Pi could never safely ignore values. To define its reliance on honest behaviour of its fellows the rely-condition is:

6 $x' \leqslant x$

The obvious design will be to ensure that p(x') is true for any value of X differing from its initial value. In order to prove such a dynamic invariant true, each process must have a guarantee-condition:

7 $x' \neq x \Rightarrow (x' < x \wedge p(x'))$

This is a case where the conjunction of the process post-conditions (cf. RESULT) would not yield the overall post-condition. Here, it would be possible to patch things up by complicating prePi

and relyPi but it is preferable to employ a dynamic invariant:

8 $x' = x \vee p(x')$

This can be seen to satisfy INVBASIS and INVPRESi. The conjunction of the post-conditions and the dynamic invariant are sufficient to establish 2. (This was, in fact, an example where the guarantee-conditions were derived from the dynamic invariant in the design process.) An even stronger case for the use of a dynamic invariant can be made in the example of sub-section 5.2.8.

The next example continues the study of the prime algorithm presented in 4.1(20-25). Since the REMOVE operation is total, the DOMi conditions are vacuously satisfied. Since the conjunction of the post-conditions are inadequate to ensure (RESULT):

9 $sieve' = sieve - (\{ m*x1 \ \underline{st} \ m \in Nat \} \cup \{ m*x2 \ \underline{st} \ m \in Nat \})$

it is necessary to think of a dynamic invariant. (In the actual design, this would come first.) Thus:

10 $(\forall c \in sieve - sieve';\ (\exists m,n \in Nat;\ m*n = c))$

Clearly INVBASIS is satisfied because the dynamic invariant is reflexive. Furthermore, for either process, the guarantee-condition is absorbed by the dynamic invariant (INVPRESi):

11 $(\forall c \in sieve - sieve';\ (\exists m,n \in Nat;\ m*n = c)) \wedge$
$(\forall c \in sieve' - sieve";\ (\exists m \in Nat;\ m*x1 = c)) \Rightarrow$
$(\forall c \in sieve - sieve";\ (\exists m,n \in Nat;\ m*n = c))$

Assuming that no process external to 4.1(20) is changing SIEVE, then INVPRESENV is immediate. Also immediate is the containment of the guarantee-condition by the rely-condition (INTERFEREi).

Before leaving aside the examples, it is perhaps worth highlighting a few points which arise in the larger examples of chapter 5. In sub-section 5.3.4 a problem similar to that with 4.1(20-25) arises. The obvious post-condition for EQUATE12 would be:

12 $f' = f \dagger \{root(f,e1) \mapsto root(f,e2)\}$

but this is too strict because of the parallel path compression which might be going on. The attempt to perform the design step prompts a dynamic invariant and guarantee- and rely-conditions which partition the fields of effect. The interest in this problem is that the CLEANUP is essentially performing "garbage collection" and a basically similar approach appears to handle the design of the "On-the fly Garbage Collector" of Dijkstra /78a/. The example in section 5.3 is one where not only was the dynamic invariant documented as part of the design discovery, but also the design and justification suggested a program better than that first sketched. Furthermore, the method has also made it possible to solve a problem which was left open in Jones /80c/.

It is interesting to note that the idea of a dynamic invariant could be used in discussing sequential programs. With some manipulation, it is even possible to see a sequential proof rule like 3.2.2(IWHILEUP) as an instance of PARCOOP.

4.4.2 Guarantee Conditions

It must be remembered that guarantee-conditions are expected to hold for the atomic steps of a process. Thus guarantee-conditions must be inherited by the development process. It is possible (cf. section 4.2) to strengthen the implied promise and still satisfy a specification. This situation will frequently arise in decomposition where some sub-components will not need the full freedom of an overall specification. A special case of this is where (e.g. ROOT in EQUATE of section 5.3) some sub-components require only read access to the global variables.

It is normally easy to show that guarantee-conditions are fulfilled and the justification is normally fairly informal. It is, however, important to remember that interference (as defined by the rely-condition) can occur between atomic steps. Thus, even if assignment is atomic:

```
1   X:=1;
    Y:=Y + X
```

only permits claims to be made about y' which follow from the rely-condition on X.

Notice that it is possible to avoid the strict rule of Owicki /75a/ that only one global reference is permitted per assignment. Even if assignment is not atomic:

```
2   X:=X + 1
```

guarantees $x' > x$ if the rely-condition gives $x' = x$ or,

alternatively, $x' \geqslant x$ if the rely-condition defines $x' \geqslant x$.
Some practical examples of two global references in an assignment occur in section 5.3.

4.4.3 Effect on Sequential Proofs

The use of the parallel construct is, of course, just another design step like those of refinement in chapter 2 and decomposition in chapter 3. A series of design steps may go through both data refinement and decomposition to some sequential construct before parallelism is introduced. Equally, a specification in the form of 4.1(8) might be realised by some sequential construct. It is this situation, where an implementation is sought to satisfy a specification which includes rely- and guarantee-conditions, which is of concern here.

Where data refinement is to be used the rules about weakening rely-conditions and strengthening guarantee-conditions (cf. 4.2(1)) suffice. Examples of this are given in chapter 5. One interesting point is that the analog of the "potential problem" discussed in sub-section 2.2.6 actually arises in the refinement of sub-section 5.2.8.

Where a decomposition into a control structure is made, it should be obvious that the sub-components must inherit the rely-condition (in a possibly weaker form) and the guarantee-condition (in a possibly stronger form) of the specification. In a step of decomposition it is obviously necessary to compose the rely-relation

between each component. The proof rules for the sequential combinators of chapter 3 are modified in an obvious way. Their use is illustrated in chapter 5.

Section 3.4 refers to the rule of "active decomposition"; an analogous rule avoids unconsidered copying of interference specifications from one level of design to another.

4.5 Justification of Proof Rule

It only remains to show that the relational form of the proof rule of sub-section 4.4.1 is valid with respect to the operational semantics of section 4.3. It would be excessively heavy to set up a full set of notation (cf. Walk /69a/) for this one proof and, for this reason, the argument given here is less formal than the proof in section 3.3.

Any leaf node of the control tree is "available" to be performed. Suppose one of the available leaf nodes is an element of Par (4.3(2)), say

i $Pa = (\|\ P1 \ldots Pn)$

Notice that Pa may not be the only leaf node - call any others the "interference" of Pa. If the Pa node is chosen for execution (and the state value is s) then it is replaced in the Control by a bag of P1 to Pn (cf. 4.3(15)). The first step of the proof is to show that when each Pi is removed from the tree, the state (say) s_i' is such that:

ii $M[\![Pi]\!] : s \leftrightarrow s_i'$

This will, in general, only be true if $R[\![Pi]\!]$ is valid. So, consider changes to the state made by other than Pi. If there are no such changes then $R[\![Pi]\!]$ holds because it is required to be reflexive. If a change is made by some Pj also an element of Pa, then by 4.4.1(INTERFEREi):

iii $G[\![Pj]\!] \subseteq R[\![Pi]\!]$

If a state change is caused by some leaf in the interference of Pa then by 4.4.1(INTERFEREENV)

iv $R[\![Pa]\!] \subseteq R[\![Pi]\!]$

Any sequence of the above is also contained because $R[\![Pi]\!]$ is transitive. So $R[\![Pi]\!]$ is preserved providing all components have been developed so as to meet their specifications.

Now consider $T[\![Pi]\!]$ and $M[\![Pi]\!]$

4.4.1(DOMi) gives:

v $T[\![Pa]\!] \subseteq T[\![Pi]\!]$

and thus:

vi $M[\![Pi]\!] : s \leftrightarrow s_i'$

But because $R[\![Pi]\!]$ is accepted in the design of Pi, vi will remain true while all other components of Pa are run to termination. This argument is general for all $i \in \{1..n\}$.

Now, to see that the dynamic invariant is preserved, observe that it is reflexive (cf. 4.4.1(INVBASIS)); is preserved by any element of Pa (cf. 4.4.1(INVPRESi)); and is preserved by the environment (cf. 4.4.1(INVPRESENV)).

All processes must terminate since this is a part of their specification and thus eventually all post-conditions and the dynamic invariant will hold. At the point at which the bag of processes (and their subsequent developments) becomes empty, 4.4.1(RESULT) can be used to justify the overall post-condition.

Chapter 5

Examples

The examples in this chapter show the use of the method described in chapters 2 - 4 on more interesting problems. Given the constraints of a thesis, the examples tackled here are of reasonable size although they are clearly not on the same scale as applications of the sequential parts of "VDM".

For each of the problems to be considered a specification is given which makes no mention of parallelism. Furthermore, the initial design in each case is of a sequential program. The sorting problems (section 5.1) illustrate how a study of data structures via data type invariants can yield more insight into a design than an algorithmic definition. The sequential solution of the problem in section 5.2 illustrates design by decomposition and that in section 5.3 design by data refinement using theories of data types.

The parallel solutions in section 5.2 employ both the basic interference ideas, with control via monitor-like tasks, and an interesting data refinement. The first refinement proof in section 5.3 is forced to handle parallelism in the same step of design. Furthermore, the rather intricate rely- and guarantee-conditions were a tool to aid the discovery of the algorithms given for the equivalence relation problem.

For each specification, a number of different designs are given and justified.

5.1 Sorting and Searching

This, rather short, section trespasses on an area which is thoroughly described in the literature (cf. Knuth /73a/). The normal method of describing sorting techniques is to present a sketch of the algorithm in some more or less formal (pseudo-code) language. This description, because of its opacity, is almost always supported by examples. The position taken here is that much more insight can be gained by studying the data structures involved. Since one general structure (a list) would cover most algorithms, the taxonomy can best be tackled via the data type invariants.

A more general point lies behind the specific examples given here. It can be argued that the amount of engineering knowledge recorded about software is very limited: nearly all software designs begin "from scratch". The obvious advantages of re-using some of the (enormous) existing reservoir of code has not made it happen. Chapter 6 of Jones /80a/ indicates how an outline of an algorithm with precise specifications for its components might be used to record design. Here, it is suggested that recording facts about data structures may be an even better way of communicating designs for some problems.

The examples of this section also offer the chance to exhibit some of the list notation developed in sub-section 2.1.6.

5.1.1 Binary Search

The task of locating an element in an ordered array can be specified by:

1 SEARCH
globals L: rd El-list1 E: rd El IND: wr (inds L) FOUND: wr Bool
pre isascending (l)
post if e ∈ elems l then found' ∧ l(ind') = e else ~found'

The post-condition is equivalent to the slightly less obvious form:

2 if found' then l(ind') = e else e ∉ elems l

The by-now-standard way to solve this problem is to use "binary search" (cf. Reynolds /79a/). One way of understanding this idea is to study the invariant on valid states. Suppose a variable I is to be used to record the lower end of the search range and J the upper, the invariant becomes:

3 isascending (l) ∧ 1 ≤ i ∧ j ≤ len l ∧
(if found then l(ind) = e else
e ∉ (elems subl(l,1,i-1) ∪ elems subl(l,j+1,len l)))

The new variables have to be declared in a block. No formal proof rule is used to justify this. Initialisation to establish the invariant is easy to choose. Thus:

4
```
declare
    I,J: Int;
begin
    FOUND:=FALSE;
    I:=1;  J:=len L;
    while ¬(FOUND ∨ I > J) loop BODY endloop;
end
```

with:

5 BODY
globals ... I: wr Int J: wr Int
pre $i \leq j \wedge \{i \,..\, j\} \subseteq$ inds l
post $(j' - i' < j - i) \vee$ found'

Notice that postBODY is very weak because most of the important conditions are expressed in 3. The loop used in 4 can be proved to satisfy 1 using the 3.2.2(IWHILEUP) proof rule. The domain conditions are all immediate. The predicate summarising the iterations at the "front" of the loop need only constrain:

6 $l' = l \wedge e' = e$

With this definition, the BASIS and COMP conditions are also immediate. RESULT becomes:

7 $l' = l \wedge e' = e \wedge (found' \vee i' > j') \wedge$
(if found' then l'(ind') = e'
else e' $\notin$ (elems subl(l',1,i'-1) $\cup$ elems subl(l',j'+1,len l')))
$\Rightarrow$
if found' then l(ind') = e else e $\notin$ elems l)

Termination can be established using a cross-product of FOUND (TRUE > FALSE) and J-I.

The next stage of decomposition might yield:

8 [PICKIND;
if L(IND) = E → FOUND:=TRUE
[] L(IND) < E → I:=IND + 1
[] L(IND) > E → J:=IND - 1 fi
] sat BODY

with:

9 PICKIND
globals I: rd Int J: rd Int IND: wr Int
pre $i \leq j$
post $i \leq ind' \leq j$

It would be possible to develop a design, satisfying the specification in 1, with two tasks working independently in the interval defined by I and J. This would give rise to:

10 I:=1; J:=len L; (P1 || P2)

11 Pi (for i = 1 or 2)
globals L: rd El-list1 E: rd EL I: wr Int J: wr Int
post $j < i \vee j = i \wedge l(i) = e$
rely $i \leq i' \wedge j' \leq j \wedge l' = l \wedge e' = e$
guar $i \leq i' \wedge j' \leq j$

But there is very little incentive for pursuing such a solution: there would clearly be a requirement for synchronisation during the setting of I and J and yet the time to test for equality with E is very short. Furthermore, the points which arise are made more clearly with the example in section 5.2. (Milner /80a/ does give a parallel solution to a related problem; searching for the zero point of a function is used which shifts somewhat the balance of computation versus housekeeping.)

5.1.2 Sorting

A second illustration of the way data type invariants capture a design can be found in internal, in-place sorting:

1 SORT
globals L: rd Key-list
post isascending (l') ∧ ispermutation (l,l')

An obvious approach is to introduce a new variable I and then use an invariant:

2 isascending (subl(l,1,i))

This invariant is to be preserved by a loop which increases I to the value of the list length (say N). There are two basic alternatives. The first is to absorb, at each step, the I + 1 st element of L, thus:

3 for I in (2 .. N) loop BODY (I) endloop

with front as:

4 ispermutation (subl(l,1,i), subl(l',1,i)) ∧
subl(l,i+1,len l) = subl(l',i+1,len l)

5 BODY(I)
globals L: rd Key-list
pre i ∈ inds l
post (∃ j ∈ {1..i+1} ; subl(l,1,i) = del(subl(l',1,i+1),j) ∧
l'(j) = l(i+1)) ∧
subl(l',i+2,len l) = subl(l,i+2,len l)

Implementations of BODY(I) might either shuffle L(I+1) down ("straight insertion") or find J by a binary search and move all elements above J at one time ("binary insertion").

An alternative to slavishly accepting the I+1st element of L for insertion is to select the lowest remaining element of L. The

effect of this design decision can be seen immediately by an extra clause which can be conjoined to the invariant (2):

6 $(\forall l \in \underline{\text{elems}}\ \text{subl}(l,1,i),\ r \in \underline{\text{elems}}\ \text{subl}(l,i+1,\underline{\text{len}}\ l);\ l \leq r)$

Thus:

7 for I in (1..N-1) loop BODY(I) endloop

With rest as:

8 $\text{subl}(l',1,i) = \text{subl}(l,1,i) \wedge$
$\text{ispermutation}\ (\text{subl}(l',i+1,\underline{\text{len}}\ l),\text{subl}(l,i+1,\underline{\text{len}}\ l))$

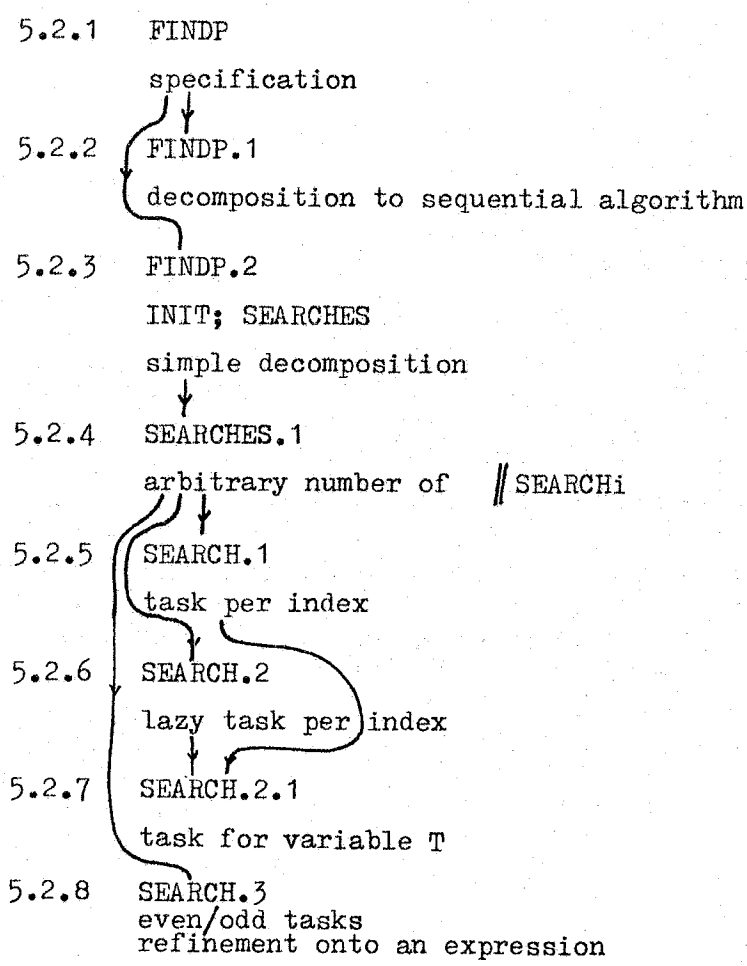
9 BODY(I)
globals L: rd Key-list
pre $i \in \underline{\text{inds}}\ l$
post $(\exists j \in \{i+1\ ..\ \underline{\text{len}}\ l\}\ ;\ \text{subl}(l',i+2,\underline{\text{len}}\ l) = \text{del}(\text{subl}(l,i+1,\underline{\text{len}}\ l),j)$
$l'(i+1) = l(j)) \wedge$
$\text{subl}(l',1,i) = \text{subl}(l,1,i)$

A greater challenge is the QUICKSORT algorithm. This algorithm is studied in Allen /72b/. One of the most interesting facets of QUICKSORT is the reasoning concerning termination. This would probably be handled more elegantly than in the original report by using bag ordering on the indices of inverted elements. The mistake is made in the report itself of using sets in the design: QUICKSORT is so tied to manipulation of indices that this abstraction is somewhat unnatural.

5.2 Locating an Array Element

The problem discussed in this section is taken from Owicki /76a/ although several alternative implementations are given before one corresponding to the original design appears (sub-section 5.2.8). The implementations give an idea how interferences interact. In addition, sub-section 5.2.7 shows the refinement of a variable into a task and 5.2.8 the refinement of a variable into an expression in terms of other variables.

The specification in sub-section 5.2.1 makes no mention of parallelism and sub-section 5.2.2 gives a simple sequential solution. The remainder of the section explores how parallelism can be used to increase the (potential) performance. The development is broken into several steps as illustrated in the following diagram.



5.2.1 FINDP

The task to be considered is, given an array with indices 1 to N, find the lowest index such that the corresponding element of the array satisfies some predicate (say P). That is, set RES to be an index to the array such that P is false for all elements of the array with lower index and P is true for the array element indexed by RES. By convention, RES is set to some value greater than N if no element of the array satisfies P. Thus:

```
1   procedure FINDP;
       globals  X: rd array (Ind) of Val
                RES: wr Inde
    spec
       post (∀ i ∈ {1 .. res'-1} ; ~ p(x(i))) ∧ satp(x,res')
       rely  x' = x  ∧ res' = res
    end
```

With a pure function:

```
2   function P(V: in Val) return Bool
```

and:

```
3   subtype Ind is Int range 1 .. N
4   subtype Inde is Int range 1 .. N+1
5   satp(x,t) ≙ t ≤ N ⇒ p(x(t))
```

(The global array X is not essential to the problem; it only makes some of the rely-and guarantee-conditions more interesting.) The condition relyFINDP is not used explicitly in the sequential solution since isolation is assumed. To see that such a result exists notice that postFINDP is equivalent to:

6 if ($\exists$ i $\in$ Ind; p(x(i))) then res' = mins ({ i $\in$ Ind st p(x(i)) })
else res' = N+1

Although solutions, much less parallel designs, are not yet being discussed, it is easy to see that RES could be initialised to N+1 and that a dynamic invariant of satp(x,res) might be preserved.

5.2.2 FINDP.1

The first design to be considered is a purely sequential loop. The requirement is to achieve postFINDP (5.2.1(1)). A standard technique for designing a loop is to begin with the invariant, and an equally standard way of seeking an invariant is to introduce a temporary (CTR) which weakens the final post-condition into an expression which is both easy to satisfy by initialisation and to preserve. Such an invariant (inv1) is:

1 ($\forall$ i $\in$ {1 .. min(ctr,res)-1} ; ~p(x(i))) $\wedge$ satp(x,res)

Thus the design step can be shown by:

2 ‖ declare CTR: Inde
begin
RES:=N+1; CTR:=1;
while CTR $\leq$ N loop BODY endloop ‖ sat FINDP

The fact that X is not overwritten can be recorded by a front predicate:

3 x' = x

but since only read access is possible, no proof that this is

preserved need be given. It is easy to see that BODY needs a specification like:

4 BODY
globals X: rd array (Ind) of Val
RES: wr Inde
CTR: wr Inde
pre ctr $\leq$ N
post ctr' > ctr

The proof uses 3.2.2(IWHILEUP): the DOMINIT condition follows because of the types of the variables; DOMLOOP is immediate; DOMX is vacuous; and DOMBODY is immediate. Even the conditions involving relations are relatively simple: BASIS and COMP are both trivial; RESULT becomes:

i $front(s,s') \wedge invl(s') \wedge \sim(ctr' \leq n') \Rightarrow postFINDP(s,s')$

ii $x' = x \wedge (\forall i \in \{1..min(N+1,res')-1\};\ \sim p(x'(i))) \wedge satp(x',res') \Rightarrow$
$(\forall i \in \{1..res'-1\};\ \sim p(x(i))) \wedge satp(x,res')$

The termination (STOP and DECR) conditions follow easily with a "variant function":

5 N+1 - ctr

The main requirement on BODY is that imposed by the invariant (1). A possible design is:

6 [if P(X(CTR)) then RES := CTR; CTR:=N+1;
else CTR:=CTR+1
endif]

(Note that it would be preferable to use a for loop with exit statement but such control constructs are not considered in the

current work - cf. discussion in sub-section 3.5.3.)

5.2.3 FINDP.2

As the name (FINDP.2) indicates, this sub-section presents an alternative to the development in sub-section 5.2.2. That is, a new development from the specification 5.2.1(1) is proposed here. The design step is, in fact, a trivial decomposition preparatory to the introduction of parallel tasks. The decomposition of FINDP can be presented as:

```
1   procedure FINDP is
        declare T: Inde
    begin
        T:=N+1;
        SEARCHES
            globals X:rd array (Ind) of Val
                    T:wr Inde
            pre  satp(x,t)
            post consid(x,t',Ind) ∧ satp(x,t')
            rely x' = x ∧ t' = t
        RES:=T;
    end
```

The identification of "consid" as a separate predicate aids the subsequent presentation:

2 $\text{consid}(x,t,s) \triangleq (\forall i \in s;\ p(x(i)) \Rightarrow t \lesssim i)$

The validity of relySEARCHES follows from relyFINDP (5.2.1(1)) and the fact that 1 makes T local. (Notice that both rely-conditions could be weakened by only requiring that the value does not alter in a way which changes whether or not it satisfies P.)

As is required of a development method, having made and justified this design decision, it need not be considered again: subsequent development is concerned only with SEARCHES.

5.2.4 SEARCHES.1

This sub-section records and justifies the decision to implement SEARCHES by a family of parallel tasks. An array (GRPS) is introduced in which the i'th element records the set of (X) indices for which the i'th task is responsible.

An obvious objective is to arrange that it is always true that:

1 $satp(x,t') \wedge x' = x$

Thus the individual tasks must be constrained so that any change made to T will represent an index for which the corresponding element of X satisfies P. Furthermore, each task must consider the set of indices for which it is responsible; if the value of T could increase as well as decrease, this would be impossible. Suppose SEARCHi is responsible for checking the indices:

2 $\{1,3,5\}$

where P is satisfied by X(3) only and suppose further that T has the value 2 during the execution of SEARCHi but is reset to 4 just before termination; then the lowest index would not be found. Thus a rely-condition of:

3 $t' \leq t$

is required (notice that this is reflexive and transitive). This is

in addition to assumptions on X. A clause corresponding to 3 will also be required for the guarantee-condition.

Thus the design of SEARCHES can be shown as:

```
4   declare GRPS: constant array (Tind) of Ind-set:= ...;
        task type SEARCH;
        task body SEARCH is
            globals X: rd array (Ind) of Val
                    T: wr Inde
            MINE: constant Ind-set:=GRPS (SEARCH' index);
            post  consid (x',t',mine)
            rely  x' ⌈ mine = x ⌈ mine  ∧  t' ≤ t
            guar  t' ≠ t ⇒ t' < t  ∧  satp(x,t')
        end
        SEARCH ARRAY: array (Tind) of SEARCH;
    begin
    end  - -  waits for all tasks to terminate
```

(Notice that the post-condition alone could be satisfied by setting T to one - this is ruled out by the guarantee-condition. It is also interesting to note that the post- and guarantee-conditions do not prevent a task looking outside its set MINE of indices although a stronger rely-condition would be required to make this safe.)

The correctness argument uses 4.4.1(PARCOOP) with the envisaged dynamic invariant:

5 $x' = x \wedge satp(x',t')$

The DOMi conditions are vacuous; INVBASIS becomes:

i preSEARCHES(s) ⇒ dinv(s,s)

which is immediate; the INVPRESi conditions become:

ii $dinv(s,s') \wedge guarSEARCHi(s',s'') \Rightarrow dinv(s,s'')$

which requires a simple case analysis as to whether T changes or not. The INVPRESENV condition is obvious; the INTERFEREi conditions become:

iii $\bigwedge_{j \neq i} guarSEARCHj(s,s') \Rightarrow relySEARCHi(s,s')$

which is immediate. Also immediate is the acceptability of the interference from the environment (INTERFERE ENV). The RESULT condition becomes:

iv $dinv(s,s') \wedge \bigwedge_{i} postSEARCHi(s,s') \Rightarrow postSEARCHES(s,s')$

v $x' = x \wedge satp(x',t') \wedge consid(x',t',\underline{union}\ \underline{rng}\ grps) \Rightarrow$
$postSEARCHES(s,s')$

which follows providing the range of GRPS covers the index set:

vi $Ind \subseteq \underline{union}\ \underline{rng}\ grps$

In these interference proofs (iii) the rely-and guarantee-conditions are taken from instances of the same task but, since 4 provides only a specification, the different instances could be implemented by different code.

The next step could be to implement the (abstract) object T as a task and achieve appropriate atomicity via "rendezvous". This is done in sub-section 5.2.7. However, SEARCH is first refined somewhat before T is considered.

Here again, the criteria for a development method are met in that this design step and its justification will not have to

be reconsidered. In fact, the sequential implementation of sub-section 5.2.2 is a special case of 4. Notice also that a great deal of freedom has been left: there is an obvious incentive to read T in order to avoid unnecessary work but this is not required; instances of SEARCH could even spawn their own tasks.

5.2.5 SEARCH.1

Three implementations of the specification for SEARCH given in 5.2.4(4) are considered. That studied in the current sub-section can be viewed as being maximally parallel in that it employs a task per index (of X). Two consequences of this decision should be noted. Firstly the declaration of SEARCHARRAY in 5.2.4(4) becomes:

```
1   SEARCHARRAY: array (Ind) of SEARCH;
```

Secondly the array GRPS is not required and the unit sets which would have been allocated to the local variables named MINE are now assigned as elements to local variables named ME. Thus the specification for SEARCH in 5.2.4(4) can be specialised to:

```
2   task body SEARCH is
        globals  X: rd array (Ind) of Vals
                 T: wr Inde
        ME: constant Ind:=SEARCH'index;
        post  consid (x',t', {me} )
        rely  x'(me) = x(me) ∧ t' ≤ t
        guar  t'≠ t  ⇒ t' < t ∧ satp(x,t')
    end
```

The simplest implementation of SEARCH is to evaluate P at X(ME). If this test yields true the code must be careful not to risk increasing T (i.e. guarSEARCH must be respected). This facet of the problem can be postponed (cf. sub-section 5.2.7) by using a specification within the implementation as follows:

```
3   task body SEARCH
    begin
        if P(X(ME)) then
            SETT
            globals T: wr Inde
            post   t' ≤ me
            rely   t' ≤ t
            guar   t' ≠ t ⇒ t' = me ∧ t' < t
        endif;
    end
```

In order to establish that 3 is correct with respect to 2, it is necessary to use 3.2.2(IFT) extended in the way discussed in sub-section 4.4.3. Condition DOMX is fulfilled because of the variable types and DOMTH is vacuously true. The RESULTH condition becomes:

i $p(x1(me)) \wedge relySEARCH(s1,s2) \wedge postSETT(s2,s3) \wedge relySEARCH(s3,s4) \Rightarrow postSEARCH(s1,s4)$

ii $p(x1(me)) \wedge x2(me) = x1(me) \wedge t2 \leq t1 \wedge t3 \leq me \wedge x3 = x2 \wedge t4 \leq t3 \wedge x4(me) = x3(me) \Rightarrow consid(x4,t4,\{me\})$

The RESULTEL condition becomes:

iii $\sim p(x1(me)) \wedge x3(me) = x1(me) \Rightarrow consid(x3,t3,\{me\})$

Furthermore, the rely-conditions match since:

iv relySEARCH(s,s') ⇒ relySETT(s,s')

Finally, it is enough to pass guarSEARCH on to guarSETT as above since SETT brings about the only state change in 3.

Before considering a realisation of SETT, sub-section 5.2.6 introduces a lazier algorithm.

5.2.6 SEARCH.2

Suppose that the function P (5.2.1(2)) is very expensive to evaluate. There would then be an advantage in checking T to determine whether evaluation could be avoided. (In fact, given m processors, it would be optimal to initiate m tasks of the form 5.2.5(3) and the rest of the form 1.) Thus an alternative implementation of 5.2.5(2) might be:

```
1   task body SEARCH
    begin
        if ME < T then
                TESTP
                globals  X: rd array (Ind) of Vals
                         T: wr Inde
                post  postSETT
                rely  relySETT
                guar  guarSETT
        endif
    end
```

The correctness argument for 1 is similar to that given in the preceding sub-section; the only interest being in the RESULTEL

condition. As should be clear from the specification of TESTP in 1, the body of 5.2.5(3) is a possible implementation.

It is now time to consider how the behaviour of T is to be realised.

5.2.7 SEARCH.2.1

Sub-sections 5.2.5 and 5.2.6 use T as a shared variable. This is a convenient abstraction but the required monotonic behaviour is best realised via a "monitor". This corresponds to one of the uses of monitors envisaged in Hoare /74a/. In Ada this will be programmed as a task with entries; the "rendezvous" concept gives the required atomic behaviour.

Thus the program becomes:

```
1   procedure FINDP is
        task  TOP is
            entry  SET (V: in Inde);
            entry  READ (V: out Inde);
        end;
        task body TOP is
            T: Inde;
            TEMP: Inde;
        begin
            accept SET (V: in Inde) do  T:=V end;
            loop
                select
                    accept SET (V: in Inde) do TEMP:=V end;
                    if TEMP < T  then  T:=TEMP; endif;
                or
                    accept READ(V: out Inde) do V:=T; end;
                or
                    terminate
                end select;
            end loop;
        end;
```

```
2       begin
            TOP.SET(N+1);
            declare
                task type SEARCH;
                task body SEARCH is
                    ME: constant Ind:=SEARCH'index;
                    TEMP: Inde;
                begin
                    TOP.READ(TEMP);
                    if ME<TEMP then
                        if P(X(ME)) then TOP.SET(ME);
                        endif;
                    endif;
                end;
                SEARCHARRAY: array (Ind) of SEARCH;
            begin
            end;  - - causes wait
            TOP.READ(RES);
        end
```

An indication of how much less verbose CSP can be than Ada is gained by comparing 1 and 2 with 6.2 (4-8).

5.2.8 SEARCH.3

As can be seen from the name of this development step, an alternative approach to implementing 5.2.4(4) is considered in this sub-section. SEARCH.1 and SEARCH.2 give maximum parallelism. Here, the partition of the indices into groups is accepted with, perhaps, the idea of creating one process for each processor. With each process a global variable can be associated in which the

process records the position of any index(ices) where it finds P to be true. Only one process has write access to each variable but the tasks "communicate" by reading each other's variables and thus avoid searching beyond a point where P has been found to be true.

This step can best be thought of as an unusual data refinement of T of 5.2.4(4) as:

1 mins ({ti st i∈T ind})

A special case of this approach is to split the indices into odd and even values. This then yields the design originally envisaged in Owicki /76a/. This pre-defined division again avoids the need for the array GRPS.

The appropriate specialisations of 5.2.4(4) use a post-condition of:

2 consid(x',t',evens(N))

where:

3 evens(n) = {i∈Nat st i≤n ∧ iseven(i)}

The rely- and guarantee-conditions are obvious.

FINDP must now include appropriate declarations of ET and OT and initialisation of both to N+1. The refinement of T onto min(ET,OT) shows the use of a true shared variable implementation. Thus:

4 task body ESEARCH is
globals X: ... ET: ... OT: ...
EC: Inde;
post consid (x', min(et',ot'), evens(N))
rely x' ⌈evens(N) = x ⌈evens(N) ∧ et' = et ∧ ot' ≤ ot
guar et' ≠ et ⇒ et' < et ∧ satp(x,et')
end

5 OSEARCH mutatis mutandis

Considering the refinement (cf. sub-section 4.4.3), it is easy to see that the guarantee-condition of ESEARCH is a strengthening (under the retrieve function) of guarSEARCH:

i guarESEARCH (s,s') ⇒ guarSEARCH (retr(s), retr(s'))

ii (et' ≠ et ⇒ et' < et ∧ satp(x,et')) ∧ ot' = ot ⇒
(min(et',ot') ≠ min(et,ot) ⇒
min(et',ot') < min(et,ot) ∧ satp(x,min(et',ot')))

Is relyESEARCH a weaker form of relySEARCH? Unfortunately, no! The difficulty is analogous to the "potential problem" discussed in sub-section 2.2.6. It is not true that:

iii min(et',ot') ≤ min(et,ot) ⇒ et' = et ∧ ot' ≤ ot

Moreover, there is no way of stating an appropriate condition on the more abstract level. There is, however, no problem in reproving acceptable interference between ESEARCH and OSEARCH. The INTERFEREi condition of 4.4.1(PARCOOP) becomes:

iv (ot' ≠ ot ⇒ ot' < ot) ∧ et' = et ⇒ et' = et ∧ ot' ≤ ot

(Clearly what is missing in the refinement step is the recognition that only ESEARCH has write access to ET - it would be worth

observing how often this property arises before seeking to extend the proof rules.)

The code for ESEARCH becomes:

```
6   task body ESEARCH is
        EC: ...;
    begin
        EC:=2;
        while  EC < min(ET,OT) loop
            if P(X(EC)) then ET:=EC endif;
            EC:=EC + 2;
        endloop
    end
```

This loop checks ET at each iteration. The version of Ada given in DoD /80a/ takes a rather odd position on shared variables: on the one hand they are permitted and on the other it is very difficult to ensure that a defined result is obtained. In order to minimise the use of SHARED-VARIABLE-UPDATE, the freedom in 4 could be exploited to check OT less often than is done in 6.

The justification of this step can be made with a suitably extended version of IWHILEUP with invl as:

7 $ec \leqslant N+2$

and preBODY as:

8 $ec \leqslant N$

all of the domain conditions are immediate. Then with "front" as:

9 consid(x',min(et',ot'), evens(ec'-2))

(X constant because read access only), BASIS becomes:

i S1; relyESEARCH; INIT; relyESEARCH $\subseteq$ FRONT

which follows from "consid" being vacuously true for the empty set; EC is local. BODY obviously satisfies the post-condition:

10 ec' = ec+2 $\wedge$ consid(x',min(et',ot'), {ec})

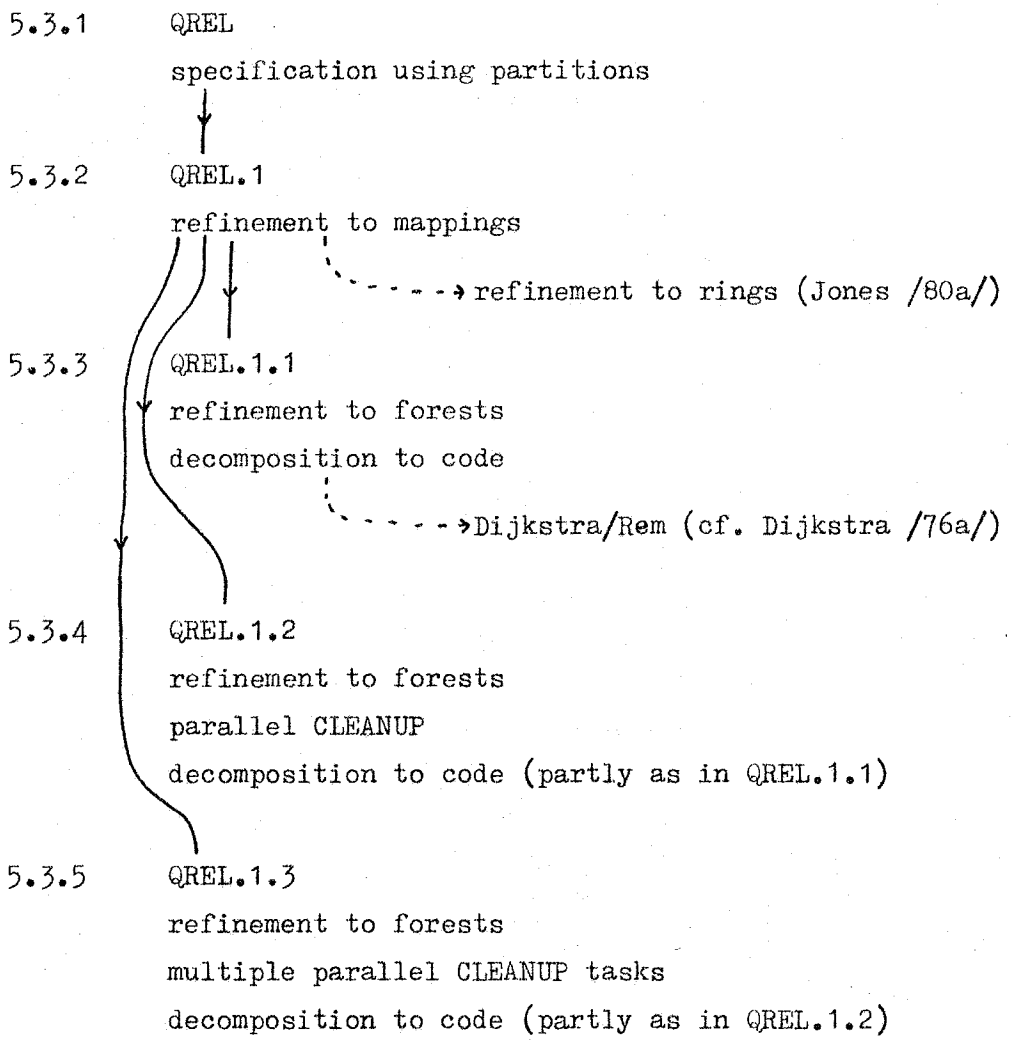
From which COMP follows since the interference can only decrease OT; RESULT is straightforward.

The preservation of guarESEARCH follows since X and ET are not changed between the test and the assignment.

5.3 Recording Equivalence Relations

The problem of recording equivalence relations occurs frequently and is widely discussed in the literature (e.g. Tarjan /75a/, Morris /72a/, Dijkstra /76a/, Jones /79a/). Most solutions are based on the Fischer/Galler algorithm. The solutions presented here are intended to illustrate the rigorous development method rather than to contribute efficient algorithms. (The approach adopted differs, even for the sequential case, from that in Jones /79a/. The changes have been made partly in response to criticisms by Lockwood Morris.) The solutions which employ parallelism have been designed with the aid of the methods of chapter 4. Once again several alternative, multi-stage, developments are shown from one specification (see diagram below). The parallel solution of sub-section 5.3.4 has something in common with the "On-the-fly Garbage Collector" discussed in Dijkstra /78a/ (a development of which has also been sketched using the methods of this dissertation.)

One point of interest in this section which has not occurred in the other examples is the acceptability of one piece of code to satisfy two or more specifications. Some points about a CSP approach to this problem are made in section 6.2.



5.3.1 QREL

A natural mathematical view of an equivalence relation is the partitioning which is induced. Partitions (cf. sub-section 2.1.8) are used as a basis of the specification. The equivalence relation can be thought of as an abstract data type with constructor operations which initialise (INIT) and record new equivalences (EQUATE) and a selector operation (TEST) which determines whether two elements are equivalent.

The specifications of the three operations, and an indication that results satisfying the specifications exist, are:

1 procedure INIT;
globals P: wr Partition (El)
post p' = {{e} st e ∈ El}
end

The fact that postINIT can be fulfilled with an object satisfying the invariant follows from 2.1.8(2).

2 procedure EQUATE (E1: in El, E2: in El);
globals P: wr Partition (El)
post p' = {S ∈ p st e1 ∉ S ∧ e2 ∉ S} ∪ {union {S ∈ p st e1 ∈ S ∨ e2 ∈ S}}
end

Existence of such a result follows from 2.1.8(3).

3 function TEST (E1: in El, E2: in El) return RES: Bool;
globals P: rd Partition (El)
post res' ⇔ (∃ S ∈ p st e1 ∈ S ∧ e2 ∈ S)
end

Existence is immediate.

To show that this specification is not "biased" it is necessary to show that equivalence under the operations implies equality of the underlying objects (i.e. members of Partition (El)). Since the only selector operation is TEST, it is necessary to prove:

4 (∀ e1,e2 ∈ El, p, p' ∈ Partition (El);
((∃ S ∈ p; e1 ∈ S ∧ e2 ∈ S) ⇔ (∃ S' ∈ p'; e1 ∈ S' ∧ e2 ∈ S')) ⇒
p = p')

Consider any:

i S ∈ p

then:

ii $\{\} \subset S \subseteq El$ i, 2.1.8(1)

from the equivalence given (as the antecedent in 4) there must exist:

iii $S \subseteq S' \in p'$ ii, 4

and similarly:

iv $S' \subseteq S$

therefore:

v $S' = S$ iii, iv

but since i is general, then

vi $p = p'$ i, v

The specification given in 1-3 is an abstraction of many representations (e.g. lists of pairs, Boolean arrays, Fischer/Galler trees, ring structures). In fact 4 has shown that partitions are an abstraction for any valid implementation of the problem.

5.3.2 QREL.1

The final algorithms presented in this section represent the equivalence classes as trees. This sub-section introduces an intermediate representation which both isolates some of the problems and which provides a starting point for other, non-tree, algorithms (e.g. the ring version in Jones /80a/).

The data structure used here is a mapping from the given set of elements (El) to an arbitrary set of keys (Key):

1 Totmap (El) = El $\xrightarrow[m]{}$ Key

The relation between 1 and the partition of the preceding sub-section can be given by:

2 retrp: Totmap (El) → Partition (El)
retrp(m) ≙ {{e st m: e ↦ k} st k ∈ rng m}

It is easy to give a constructive argument for the adequacy of 1 - but notice that adequacy does rely on the absence of empty sets in partitions (cf. 2.1.8(1)). In view of the invariant on 2.1.8(1) the model in 1 might appear to be simpler. It should, however, be noted that the latter model would be biased and should, therefore, not be used as a specification.

The retrieve function (2) gives a clear indication of what the operations on the representation must be. Thus:

3 procedure INIT1;
globals M: wr Totmap (El)
post dom m' = El ∧ isoneone (m')
end

Clearly, such an m' exists providing the cardinality of the Key set is at least as great as that of El. The result is not, however, unique. Condition 2.2.1 REFINE (RESULT) becomes:

4 (∀m,m' ∈ Totmap (El); postINIT1(m,m') ⇒
postINIT(retrp(m), retrp(m'))

which is immediate. The DOMAIN condition is vacuous for total operations.

The EQUATE operation is to be modelled by:

5 procedure EQUATE1(E1: in El, E2: in El);
globals M: wr Totmap (El)
post $m' = m \dagger \{e \mapsto m(e2)$ st m: $e \mapsto m(e1)\}$
end

(Note that the choice of which element's key to preserve is arbitrary and that a more general post-condition could cover both cases.) Clearly m' must exist and, again, the DOMAIN condition is vacuous. The RESULT condition becomes:

6 $(\forall m,m' \in$ Totmap (El); postEQUATE1(m,e1,e2,m') $\Rightarrow$
postEQUATE(retrp(m),e1,e2,retrp(m')))

Now:

i $retrp(m') = \{\{e$ st $m':e \mapsto k\}$ st $k \in$ rng $m'\}$ 2

ii $= \{\{e$ st $m':e \mapsto k\}$ st $k \in$ rng $m' \wedge k \neq m'(e1)\} \cup$
$\{\{e$ st $m':e \mapsto m'(e1)\}\}$ i

iii $= \{\{e$ st $m:e \mapsto k\}$ st $k \in$ rng $m \wedge k \neq m(e1) \wedge k \neq m(e2)\} \cup$
$\{\{e$ st $m:e \mapsto k\}$ st $k = m(e1) \vee k = m(e2)\}$ ii, 5

iv $= \{S \in retrp(m)$ st $e1 \notin S \wedge e2 \notin S\} \cup$
$\{$union $\{S \in retrp(m)$ st $e1 \in S \vee e2 \in S\}\}$ 2

$\therefore$ postEQUATE(retrp(m),e1,e2,retrp(m')) 5.3.1(2), iv

Finally:

7 function TEST1 (E1: in El, E2: in El) return RES: Bool
globals M: rd Totmap (El)
post $res' \Leftrightarrow m(e1) = m(e2)$
end

All conditions are trivially fulfilled here.

The mapping solution is useful in that it splits the development task, but the search which is implied by 5 would be unacceptable for an implementation which had to cater for large sets El. It is exactly the elimination of this search which makes the tree-like structure of the Fischer/Galler algorithm attractive. As is pointed out elsewhere, it is an essential feature of a development method that complete and justified design steps need not be rechecked: 3,5,7 are used as the specification for three alternative designs.

5.3.3 QREL.1.1

The basic idea of the Fischer/Galler algorithm is to represent equivalence classes by (representations of) trees. In terms of the design decision in the preceding sub-section, the root of such a tree can be thought of as the key for the elements in the tree. The advantage of this representation is that EQUATE can be performed by grafting the root of one tree onto some point in the other tree: this changes the keys of all elements in that class at one step. The search time is proportional only to the average depth of the trees (a point which must be considered again below).

The required data structure is (cf. sub-section 2.1.9):

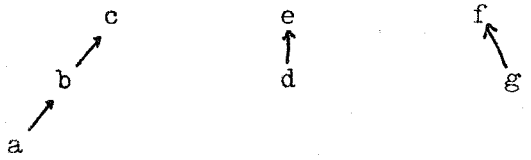
1 Forest (El)

The relation to 5.3.2(1) is defined by:

2 retrm: Forest (El) $\rightarrow$ Totmap (El)
retrm(f) $\triangleq \{e \mapsto root(f,e)$ st $e \in El\}$

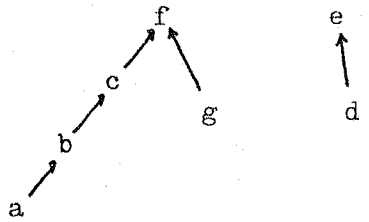
An example of a forest and the effect of a change might be:

f1



EQUATE (a,g): f1 ↦ f2

f2



It is again easy to argue that the representation is adequate. It is, however, important to notice that there are many ways of representing the same mapping: optimisation consists of compressing the trees while preserving their "meaning".

The retrieve function again gives a clear idea of how to re-specify the operations. Thus:

3
```
procedure  INIT11
    globals  F: wr Forest(El)
    post  f' = {}
end
```

Clearly f' must exist and is well-founded. DOMAIN conditions (for all three operations) are vacuous. The 2.2.1 REFINE(RESULT) condition becomes:

4 $(\forall f,f' \in \text{Forest(El)};\ \text{postINIT11}(f,f') \Rightarrow \text{postINIT1}(\text{retrm}(f),\text{retrm}(f')))$

which is immediate.

Then:

5 procedure EQUATE11(E1: in El, E2: in El);
globals F: wr Forest(El)
post f' = if root(f,e1) = root(f,e2) then f
else f † {root(f,e1) ↦ root(f,e2)}
end

(As in the preceding sub-section, it would be possible to give a more general post-condition.) The existence of such a (valid) f' follows from 2.1.9(7) because:

i root(f,e1) ≠ root(f,e2) ⇒ root(f,e2) ∉ coll(f,root(f,e1))

The interesting property is the RESULT condition:

6 for f,f' ∈ Forest(El), e1, e2 ∈ El
postEQUATE11(f,e1,e2,f') ⇒
postEQUATE1(retrm(f),e1,e2,retrm(f'))

Begin by considering the case:

i root(f,e1) = root(f,e2)

ii f' = f i, 5

iii retrm(f)(e1) = retrm(f)(e2) i, 2

iv ∴ retrm(f)(e) = retrm(f)(e1) ⇔
retrm(f)(e) = retrm(f)(e2) 3

v postEQUATE1(retrm(f),e1,e2,retrm(f')) 5.3.2(5), iv, ii

Alternatively:

vi root(f,e1) ≠ root(f,e2)

vii f' = f † {root(f,e1) ↦ root(f,e2)} vi, 5

let rm = retrm(f), rm' = retrm(f') consider:

viii	$e \in El$ st rm: $e \mapsto rm(e1)$	
ix	root(f,e) = root(f,e1)	viii, 2
x	$e \in$ coll(f,root(f,e))	2.1.9(4)
xi	$e \in$ coll(f,root(f,e1))	ix, x
xii	root(f',e) = root(f,root(f,e2))	vii,xi,2.1.9(9)
xiii	= root(f,e2)	xii, 2.1.9(2)
xiv	$\therefore (\forall e$ st rm: $e \mapsto rm(e1)$; rm'(e) = rm(e2))	viii, xiii

Now consider:

xv	$e \in El$ st $\sim$ rm: $e \mapsto rm(e1)$	
xvi	root(f,e) $\neq$ root(f,e1)	xv, 2
xvii	isdisj(coll(f,root(f,e)),coll(f,root(f,e1)))	xvi, 2.1.9(6)
xviii	isdisj(reach(f,e),reach(f,e1))	xvii, 2.1.9(5)
xix	root(f,e1) $\in$ reach(f,e1)	2.1.9(3)
xx	$\therefore$ root(f,e1) $\notin$ reach(f,e)	xviii,xix
xxi	root(f',e) = root(f,e)	xx, 2.1.9(8)
xxii	$\therefore (\forall e$ st $\sim$rm: $e \mapsto rm(e1)$; rm'(e) = rm(e))	xv, xxi
xxiii	rm' = rm $\dagger$ {e $\mapsto$rm(e2) st rm: $e \mapsto rm(e1)$}	xiv, xxii
	$\therefore$ postEQUATE1(retrm(f),e1,e2,retrm(f'))	xxiii

The selector operation on forests becomes:

```
7   function TEST11 (E1: in El, E2: in El) return RES: Bool
        globals F: rd Forest(El)
        post res' ⇔ root(f,e1) = root(f,e2)
    end
```

The result obviously exists since "root" is total and the RESULT condition is trivial.

The use of the forest properties has had a considerable simplifying effect on the proofs of this sub-section: the proofs given here have not been clouded by results which relate to the data structure in general rather than its use here. Furthermore, sub-section 2.1.9 contains a body of results which can be used for other problems.

The use of the tree structures has now been justified and 3,5,7 can be taken as a specification for code which is to be designed. The forest will be represented by an array (1 .. N) where a zero value corresponds to an element not in the domain of the mapping. This simple refinement step is handled informally. Implementation of INIT11 presents a typical case for the use of 3.2.2(FORARB): the order in which the array elements are initialised is immaterial. The actual code is:

```
8   for CTR ∈ {1 .. N} loop F(CTR):=0 endloop
```

This would have to be implemented in Ada by a conventional for loop.

The implementations of EQUATE11 and TEST11 both require a ROOT function:

```
9   function ROOT11(E: in El) return RES: El is
        PTR: El;
    begin
        PTR:=E;
        while F(PTR) ≠ 0 loop
            PTR:=F(PTR);
        endloop;
        return PTR;
    end
```

This can be shown to satisfy:

10 postROOT11(f,e,res') $\triangleq$ res' = root(retrf(f),e)

by using 3.2.2(IWHILEDN) with:

i rest as ptr' = root(retrf(f),ptr)

ii preBODY as f(ptr) $\neq$ 0

Then:

```
11  procedure EQUATE11(E1: in El, E2: in El) is
        R1, R2: El;
    begin
        R1:=ROOT11(E1);
        R2:=ROOT11(E2);
        if R1 ≠ R2 then F(R1):=R2;  endif
    end
```

can be proved correct using the 3.2.2(SEQ) rule.

Similarly the correctness of 12 with respect to 7 can be checked by inspection:

```
12  function  TEST11(E1: in El, E2: in El) return RES: Bool is
        R1, R2: El;
    begin
        R1:=ROOT11(E1);
        R2:=ROOT11(E2);
        return (R1 = R2);
    end
```

The implementation of 8, 9, 11 and 12 could still lead, in carefully selected cases, to unbalanced trees and several papers have considered how to compress trees. Tarjan /75a/ keeps track of the "weight" of a tree and uses this to govern the order of the

grafting; Dijkstra /76a/ compresses the trees during EQUATE but not during TEST; "Rem's algorithm" (also Dijkstra /76a/) attempts to make the EQUATE more symmetrical (unfortunately there are cases where this algorithm actually extends trees). It is now time, with the groundwork done, to consider how parallelism can be used to help with tree compression.

5.3.4 QREL.1.2

The problem of tree depth can be tackled by a design using parallelism. In fact, an aside in Jones /79a/ mentioned that it would be advantageous to compress (or CLEANUP) trees using a parallel, low priority, process. At that time, it was thought that the interruption of the CLEANUP process would be quite messy. In the event, the use of the methods of chapter 4 has shown that it is possible to design the algorithms so that the CLEANUP process does not need to be reinitialised after interruption. (In earlier joint work with Wolfgang Henhapl, it was found that the interesting results were those where readers looked for counter examples before bothering to read the proof; presentations of these parallel algorithms have given rise to the same phenomenon.) Although the proof rules of chapter 4 can only be used to establish correctness, the general ideas of interference do appear to give a framework for thinking about possible designs.

The parallel CLEANUP provides an interesting example for the use of the interference ideas proposed here. Given the near-linear performance of the algorithms employing both compression and tree

weights, it is not being claimed that the parallel algorithms offer any great advantage.

The overall program structure can be shown by the following Ada package body:

```
1   package body QREL is
        F: array (El) of El          - - array representation of forest
    begin
        INIT12; ...          - - spec below
        declare
            task CLEANUP12;
            task OPS12 is
                entry EQUATE12(E1: in El, E2: in El);
                entry TEST12(E1: in El, E2: in El, RES: out Bool);
            end;
            task body CLEANUP12    - - spec below
            task body OPS12 is
                begin
                    loop
                        select
                            accept EQUATE12(E1: in El, E2: in El);
                                    - - spec below
                            end;
                        or
                            accept TEST12(E1: in El,E2: in El, RES:out
                                    - - spec below          Bool);
                            end;
                        end select;
                    end loop
                end
            end
        begin
            ...     - -  use of EQUATE12/TEST12
        end
    end
```

The specification and implementation of INIT12 is the same as for INIT11 because there is no danger of interference.

The specification of postEQUATE11 in 5.3.3(5) effectively prohibits any interference. If an interfering CLEANUP routine is to be tolerated, a looser condition must be derived from postEQUATE1 of 5.3.2(5). The parallel algorithm employs the same basic tree-like data structure. Considering the retrieve function in 5.3.3(2) it is clear that it is the overall tree groupings which must be conserved: EQUATE cannot tolerate interference which changes the root of any tree. A predicate to express this is:

2 $\text{rootunch}(f,f') \triangleq (\forall e \in El;\ \text{root}(f',e) = \text{root}(f,e))$

EQUATE12 is obviously concerned mainly with roots and it would be desirable to show the limitation of its effect by a predicate which states that the "body" of the forest is unchanged:

3 $\text{bodyunch}(f,f') \triangleq (\forall e \in \underline{\text{dom}}\ f;\ f': e \mapsto f(e))$

Using 2 and 3 appropriately in the interference conditions is not enough to control the development of code. It is often necessary to think ahead in a design and here it is clear that EQUATE12 requires some ROOT (cf. 5.3.3(9)) operation and CLEANUP12 must also be able to scan along the tree. Arbitrary changes to the structure of the trees could be difficult to cope with in the programs to be written. With some experimentation a sufficient constraint can be formulated as:

4 ordpres(f,f') ≙ (∀e ∈ dom f; f'(e) ∈ rreach(f,e))

5 rreach(f,e) ≙ reach(f,e) - {e}

(cf. 2.1.9.(3) for reach).

With the aid of these auxiliary definitions (2-5) the specification becomes:

6
```
entry EQUATE12(E1: in El, E2: in El);
    globals  F: wr Forest(El);
    post  root(f',e1) = root(f,e2)
    rely  rootunch(f,f') ∧ ordpres(f,f')
    guar  (∀ e ∈ El; f'(e) = f(e) ∨ e = root(f,e1) ∧ e ≠ root(f,e2) ∧
                                        f': e ↦ root(f,e2))
end
```

(Notice that if the preservation of the other roots was required by the post-condition, it would be very difficult to formulate a useful dynamic invariant.) It is clear that a result exists which matches the guarantee- and post-conditions.

The corresponding specification of the CLEANUP process has a weaker rely-condition than is given by guarEQUATE12; clearly this is acceptable.

7
```
task  CLEANUP12
    globals
    post  TRUE
    rely  bodyunch(f,f')
    guar  rootunch(f,f') ∧ ordpres(f,f')
end
```

The vacuous post-condition is interesting: CLEANUP12 is not required to have any final effect; it is only required to continually seek to optimise the data structure. Of course, without

some additional guidance about performance, one valid implementation of CLEANUP12 is to do nothing.

EQUATE12 is the more difficult case. The interaction of TEST12 with the CLEANUP12 routine occurs in only one direction. Thus:

8 entry TEST12 (E1: in El, E2: in El, RES: out Bool);
globals F: rd Forest(El)
post res' ⇔ root(f,e1) = root(f,e2)
rely rootunch(f,f') ∧ ordpres(f,f')
end

(Notice that, from the guarantee-condition implied by the "read only" access, it would be possible to run more than one instance of TEST12.)

Having used the general ideas of interference as a prompt to suggest specifications, it is now necessary to prove that the operations co-exist and achieve the desired effects. The design step considered here forces consideration of the data refinement at the same time as the interference. In fact, given that 1 causes CLEANUP12 to execute when OPS12 is waiting for a "rendezvous", the first task is to show that CLEANUP12 running alone is an identity transformation when viewed under the retrieve function 5.3.3(2):

i guarCLEANUP12(f,f') ⇒ retrm(f') = retrm(f)

which follows from:

ii $(\forall e \in El;\ root(f',e) = root(f,e))$

The next task is to establish that, under the retrieve function, EQUATE12 run in parallel with CLEANUP12 satisfies the specification of EQUATE1. The rely- and guarantee-conditions have fixed the "spheres of influence" of the two tasks. The dynamic invariant is:

i $(\forall e \in El;\ root(f',e) = root(f,e) \vee$
$root(f,e) = root(f,e1) \wedge root(f',e) = root(f,e2))$

With this the 4.4.1 PARCOOP(INVBASIS) condition is trivial. The INVPRES condition for CLEANUP12 becomes:

9 $dinv(f,e1,e2,f') \wedge guarCLEANUP12(f',f'') \Rightarrow dinv(f,e1,e2,f'')$

which follows from:

i $root(f'',e) = root(f',e)$

The INVPRES condition for EQUATE12 is simple, and the dynamic invariant is preserved by the environment since F is local to the package.

The domain rules of both operations are vacuously true.

For the INTERFERE conditions it is easy to see in one case that the guarantee- and rely-conditions are equivalent and that in the other the rely-condition is an immediate consequence.

For RESULT:

10 $dinv(f,e1,e2,f') \wedge postEQUATE12(f,e1,e2,f') \Rightarrow$
$postEQUATE1(retrm(f),e1,e2,retrm(f'))$

follows from:

i $(\forall e \in El;$ <u>if</u> $root(f,e) = root(f,e1)$ <u>then</u> $root(f',e) = root(f,e2)$
<u>else</u> $root(f',e) = root(f,e))$

As would be expected, the work involved in showing that TEST12 coexists with CLEANUP12 is less. The conditions can be established with an identically TRUE dynamic invariant. The proof follows by inspection of the conditions in 4.4.1(PARCOOP).

It is now possible to consider code to match the specifications in 6, 7 and 8. Once again a development step by data refinement is followed by operation decomposition. Since both EQUATE12 and TEST12 are concerned with roots of the trees, it is worth considering a specification for finding roots. This will only require read access to the global F so no guarantee-condition need be considered. The rely-condition, however, must be no stronger than the weakest condition where this function is to be used. Thus:

```
11  function  ROOT12 (E:in El) return RES: El;
        global  F: rd Forest (El)
        post  res' = root(f,e)
        rely  rootunch(f,f') ∧ ordpres(f,f')
    end
```

With this function the code for EQUATE11 (5.3.3(11)) satisfies the specification in 6 and the code for TEST11 (5.3.3(12)) satisfies the specification in 7. (It would also be possible to search for the roots in parallel.) Even more interestingly, the specification in 11 is satisfied by the original code for finding roots (cf. 5.3.3(9)).

The problem of implementing the tree compression routine (7) is solved in two ways. A first algorithm looks for any "dog's legs" and shortens the path appropriately. Thus:

```
12  task CLEANUP121 is
        CUR: El;
        NEXT: El;
    begin
        loop          - - forever
            for  CUR:=1 .. N loop
                if F(CUR) ≠ 0 then
                    NEXT:=F(CUR);
                    if F(NEXT) ≠ 0 then
                        F(CUR):=F(NEXT);
                    endif;
                endif;
            endloop;
        endloop;
    end
```

It is only necessary to show that guarCLEANUP12 is preserved under the assumption of interference which satisfies relyCLEANUP12. The main step in this proof is:

i $f: cur \mapsto next \wedge f: next \mapsto nnext \Rightarrow (f\dagger\{cur \mapsto nnext\})(cur) \in reach(f, cur)$

It is interesting to note that the assignment in 12 has two references to the global variable. The rely-condition shows that this is one of the cases where it is safe to break Owicki's rule (cf. Owicki /75a/).

An alternative approach to tree compression is presented because, with its initial search to the root of a tree, it matches more closely what has been done in sequential algorithms. Clearly with relyCLEANUP12 (cf. 7), it is not possible to deduce relyROOT12 (cf. 11). A function with a suitably weakened rely-condition must have a weaker post-condition. Thus:

13
```
function  ROOT122(E: in El) return RES: El;
     globals  F: rd Forest (El)
     post res ∈reach (f,e)
     rely  bodyunch (f,f')
end
```

With this function it is possible to show that:

14
```
task body CLEANUP122 is
     CUR: El;  R: El;  TEMP: El;
begin
     loop           - - forever
          for CUR:=1 .. N loop
               R:=ROOT122(CUR);
               TEMP:=CUR;
               while TEMP ≠ R loop
                    F(TEMP), TEMP:=R,F(TEMP);
               endloop
          endloop
     endloop
end
```

(Note: a first attempt at this code made the innermost loop terminate only on a root - the rather subtle error was uncovered in the proof attempt.)

Finally, how is ROOT122 to be implemented? It can be shown that, once again, the code in 5.3.3(9) satisfies this specification. Thus the one piece of code has been shown to satisfy three specifications with increasing interference (more precisely, less reliance on lack of interference).

5.3.5 QREL.1.3

An interesting question now arises: is it possible to run more than one instance of the CLEANUP process at a time? This was an open problem in Jones /80c/ which has been resolved by a study of the dynamic invariants and interference specifications.

Clearly the specification of 5.3.4(7) is too restrictive since its rely-condition is not a consequence of its guarantee-condition. The situation can be analysed as follows. The weakest guarantee-condition for CLEANUP that can be accepted by, for example, TEST is that any changes to the trees do not cause an effect to the mappings as retrieved by 5.3.3(2) (the "meaning of a tree"). A predicate to cover this is:

1 $\text{equimap}(f,f') \triangleq (\forall e \in El;\ \text{root}(f',e) = \text{root}(f,e))$

However, a CLEANUP process can only provide this guarantee if it can rely on something. It would appear that a rely-condition of "equimap" would be desirable. Although this would cover the interaction with other instances of the CLEANUP task or with TEST, it will not cover the case of interaction with EQUATE. What is the minimum requirement for CLEANUP to be able to guarantee "equimap"? It would appear to need to rely on the fact that no other process splits a group. This condition is strictly weaker than "equimap" and can be defined by:

2 $\text{growing}(f,f') \triangleq (\forall d,e \in El;\ \text{root}(f,d) = \text{root}(f,e) \Rightarrow \text{root}(f',d) = \text{root}(f',e))$

It is also necessary to reconsider the other part of the interference consideration of sub-section 5.3.4. The rather strict order preserving predicate in 5.3.4(4) can no longer be guaranteed if more than one process is working on the "body" of the tree at the same time. If, however, all attempt to check order is abandoned then, for example, an attempt to find the root of a tree may get caught in a "whirlpool" caused by continual reversing of the order of nodes. A weaker condition is:

3 wkordpres(f,f') $\triangleq$ ($\forall$d,e $\in$ El st d $\neq$ e; d $\in$ reach(f,e) $\Rightarrow$ e $\notin$ reach(f',d))

(This can be characterised by an analogy with a face-saving management policy which mandates that if d works for e at one point in time, e will never work for d however senior d becomes.)

Collecting up the thoughts outlined above leads to the following specifications:

4
```
task CLEANUP13
    globals F: wr Forest(El)
    post  TRUE
    rely  growing(f,f') ∧ wkordpres(f,f')
    guar  equimap(f,f') ∧ wkordpres(f,f')
end
```

5
```
entry EQUATE13 (E1: in El, E2: in El);
    globals  F: wr Forest(El)
    post  root(f',e1) = root(f,e2)
    rely  equimap(f,f') ∧ wkordpres(f,f')
    guar  (∀ e ∈ El; f'(e) = f(e) ∨
                e = root(f,e1) ∧ e ≠ root(f,e2) ∧
                    f': e ↦ root(f,e2))
end
```

6 entry TEST13(E1: in El, E2: in El, RES: out Bool);
globals F: wr Forest(El)
post res' ⇔ (root(f,e1) = root(f,e2))
rely equimap(f,f') ∧ wkordpres(f,f')
end

It is now appropriate to consider the correctness of an implementation along the lines of 5.3.4(1) but with parallel instances of CLEANUP. The basis for the justification is, once again, the specifications in 5.3.2(5,7).

The actual coexistence of two instances of CLEANUP follows from 4 in which it can be seen that the rely-condition is a consequence of the guarantee-condition. This of course only covers correctness; it is possible for parallel instances of CLEANUP to cause the trees to get deeper! Although the timing would be unlikely to occur, it does suggest that the number of parallel tasks should not be increased indiscriminately.

The argument that CLEANUP preserves, under the retrieve function, identical mappings is similar to that in the last sub-section and follows from "equimap".

The proof that EQUATE13 performs the required function in the presence of interference is more interesting. The dynamic invariant of the preceding sub-section can still be established:

7 $(\forall e \in El;\ root(f',e) = root(f,e) \vee$
$root(f,e) = root(f,e1) \wedge root(f',e) = root(f,e2))$

The proof (using 4.4.1(PARCOOP)) of the domain conditions is vacuous;

the INVBASIS condition is trivial; INVPRES follows from "equimap" for CLEANUP and from guarEQUATE13; the acceptability of each other's interference is immediate in one case and straightforward in the other; the proof of the RESULT condition is exactly as in the preceding sub-section.

The proof that TEST 13 satisfies its specification is straightforward.

Finally, the code for specifications 4-6 must be considered. As elsewhere, it is found that one and the same piece of code satisfies more than one specification. The code of 5.3.4(12) satisfies 4; the code of 5.3.3(11) satisfies 5; and the code of 5.3.3(12) satisfies 6.

It is interesting to speculate on further developments of this problem. There are situations in which it is necessary to permit simultaneous updating and enquiry. The difficulty which is often encountered is how to specify the required result. The equivalence relation problem is shown in sub-section 6.1.5 to be one which is amenable to simultaneous update and interrogation because there is a sense in which the answers to the queries change monotonically. An alternative possibility is to have more than one instance of the EQUATE process active at the same time. An implementation would probably require some locking. This would then provide an example on which synchronisation and deadlock problems could be investigated. This topic leads naturally into other approaches and a discussion of the limitations of the current work.

Chapter 6

Alternatives

Virtually all sequential programming languages accept the notion of a state which is modified by assignment statements. Similarly, the first attempts to produce parallel programs were built around the notion of shared variables. For many applications of parallelism this approach is still necessary in order to achieve adequate performance. There is, however, a movement to avoid the problems of shared states altogether and to achieve co-operation solely by communication. The main argument in favour of this approach being that it is easier to reason about such programs. This chapter reviews alternative methods within both approaches.

6.1 Shared Variable Parallelism

The first landmark in the attempt to bring some order into the analysis and construction of parallel programs is Dijkstra /68a/. This section considers some of the subsequent attempts to provide a formal basis for reasoning about programs which interfere with one another during execution. (Another area of comparison which is not pursued here is the work on distributed databases, e.g. Lindsay /79a/.)

6.1.1 Proof Rules

One of the earliest attempts to provide proof rules for reasoning about parallel programs is Hoare /75a/. Processes are analysed by their degree of interaction:

disjoint processes

competing processes

cooperating processes

communicating processes

In order to reason about parallel processes, notions like "commutative" operations are introduced. At least in the way this idea is presented in Hoare /75a/, these notions are applicable only to actual code. Unlike the notion of interference in chapter 4, it would be difficult to use such ideas in a development method.

Proof rules for monitors have also been given (e.g. Adams /81a/ and references therein).

Hehner and Lengauer have continued the work on commutative operations. They have, in fact, used it to propose a novel approach to the definition of potential parallelism. With only slight exaggeration, the standard approach to defining parallelism could be characterised by:

i) develop a number of processes whose interaction would be unsafe

ii) introduce synchronisation to make the interaction safe

iii) avoid the danger of deadlock

The approach in Lengauer /81a/ is:

i) develop a program which can be read as though it were sequential

ii) observe relaxations which make it possible to execute some steps in parallel.

The advantage claimed for this approach is that the amount of effort invested in step ii can be varied: the program is correct at any stage.

How are the "relaxations" of order handled? Commutativity with respect to some predicate P is defined via predicate transformers:

1 wp (S1; S2, P) = wp (S2; S1, P)

Independence is defined syntactically. E. Hehner tackled the problem of sub-section 5.2.5 at the January 1981 WG2.3 meeting as follows:

2 $T:=N+1;\ \overset{N}{\underset{i=1}{;}}\ Si$

(The large semi-colon can be read as "for".)

3 Si = if P(X(i)) then Mi fi

4 Mi = if i < T then T:=i fi

The formulae 2-4 could be read as a sequential program. If, however, it is observed that Mj and Mk commute and that each of the P(X(j)) tests is independent of other Sk, then it is clear that all of the tests may be started in parallel and the Mi executed (each atomically) on a first-come, first-served basis. In order to handle the version of this same problem which is considered in sub-section 5.2.6, 2 becomes:

5 $T:=N+1;\ \overset{N}{\underset{i=1}{;}}\ Ri$

6 Ri = if i < T then Si fi

In this case R_j is not independent of other Mk and some weaker notion would be required in order to achieve the potential parallelism.

As was discussed at the WG2.3 meeting, this approach will require some more thought before it is possible to use it in a development method in the sense of section 0.1.

6.1.2 Owicki/Gries

The "Owicki/Gries" method for developing parallel programs is referred to above. The approach proposed in Owicki /75a/ and Owicki

/76a/ is to prove (complete) programs correct in isolation and then to show that the proofs do not interfere. Applying this to larger problems, it would be necessary to perform a multi-stage development of a program postponing the issue of interference until the final code was developed. This clearly violates the basic tenet of section 0.1 in that an early error made in a long development might be discovered only in the interference check.

In practice, the situation is even worse. The "einmischungsfrei" property is quite strong and, even for correct programs, the proofs may have to be rearranged in order to satisfy the condition. Thus, not only may errors cause revision but the need for revision is also inherent in the approach.

As exemplified in section 5.3, the Owicki rule on only one global reference per assignment is too strict: the notion of interference makes it possible to be more specific about the reliance on the behaviour of other processes.

6.1.3 Z

The work which is known as "Z" is still evolving. The comments here are based on Abrial /79a/, Abrial /79b/, Abrial /80a/, Abrial /80b/ and discussions with Jean-Raymond Abrial and his colleagues. As discussed in sub-section 1.6.2 the "Z" approach to data types is also model oriented and is thus rather similar to the style used in "Meta-IV". Sequential combination of operations is specified in "Meta-IV" by the use of combinators; in "Z" such specification is

normally achieved by adding statement counters to the state. (Here, and elsewhere, the comparison must distinguish between those points of basic difference and those which are just questions of usage.) The statement counters can then be used in the definition of "firing conditions".

A class defines a set of objects - including sets to be used as "states". The "class functions" (cf. "operations") are then given firing conditions which define both a pre-condition and also any constraints governing the order of execution of class functions. When a class function is fired, it is considered to execute atomically. This approach makes it possible to give some very pleasing proofs about deadlock (cf. Lamsveerde /79a/). Unfortunately, it lends itself less well to a development method since subsequent decomposition could introduce new dangers of deadlock.

Another important difference from the methods used in this dissertation, is that "Z" specifications tend to put more in the state. In fact, this is really (only) an issue of style. But the approach adopted here is to make a clear distinction between values required in the state and those which might be put there for purposes of the proof. In contrast, a "Z" specification might have all initial values and all future inputs in the state. It is easy to argue that these should, at least, be clearly distinguished from the essential state components. But, moreover, both the experience with operational semantics definitions and the prejudice against ghost variables in refinement proofs cause doubt about the wisdom of using state components for things like a statement counter.

6.1.4 Interference Method

This sub-section reviews some of the alternatives considered in the preparation of chapter 4. The alternatives are described in terms of specification although it must be appreciated that the adoption of a different specification style would necessitate changes to the proof rules.

The acceptance of a guarantee-condition as a predicate of two states prompts re-examination of the decision to allow post-conditions to refer to the initial state. In order words, it might be hoped that a specification could be given by a guarantee-condition governing the dynamic behaviour and a post-condition defining the final state. Although this works for some problems, it was found not to cover, for example, the specification of postEQUATE in section 5.3.

The possibility of needing to specify systems in which updating can proceed in parallel with enquiries (cf. sub-section 6.1.5) prompted another possible view of post-conditions. The idea would be to interpret the post-condition as being true at some point in time but not necessarily of the final state. This approach might be investigated with a view to covering intermittent assertions (cf. sub-section 3.5.3).

There are cases where the restriction to transitive rely-conditions is inconvenient. It might, for example, be useful to be able to state that some variable either remains constant or increases by one. Lifting the restriction on transitivity poses problems with more than two tasks.

6.1.5 Temporal Logics

A number of facts prompt consideration of logics in which the notion of time is recognised in the operators (e.g. modal logic uses sometimes $\Diamond$, always $\Box$). The most urgent extension required for the interference method is consideration of synchronisation in general and deadlock in particular. For example, in proving Dekker's semaphore implementation correct (cf. Dijkstra /68a/) it is necessary to assert that something will change in the future. Using Rod Burstall's "aslongas" operator:

1 aslongas $\sim s1 \Rightarrow \Diamond$ turn = 1

2 aslongas $(s1 \wedge \text{turn} = 1) \Rightarrow \Diamond \sim s2$

Furthermore, even some specifications appear to require temporal statements. Suppose a system is to permit parallel updating and retrieval of information. Informally, it is easy to state that any answers given must have been true at some point in time. For some problems this can be formalised with post-conditions (e.g. for the equivalence relation problem of section 5.3:

$$(\sim res' \Rightarrow root(f,e1) \neq root(f,e2)) \wedge$$
$$(res' \Rightarrow root(f',e1) = root(f',e2))$$

but this only works because the results are, in a sense, monotonic). There are, however, many problems where a formal statement of the informal criterion above would require temporal operators.

In fact such temporal specifications are already in use. The behaviour specifications of sub-section 6.2.1 use the trace as a way

of defining a time sequence; even the interpretation of pre- and post-conditions requires a discussion of before and after execution. Thus, the choice is between factoring out the notion of time as against making all assertions and proofs in some temporal logic. The advantage of pre-packaging certain rules (e.g. 3.2.2(SEQ)) which reflect the notion of time is that proofs of program correctness need only use standard predicate calculus. The potential advantages of using a temporal logic include the possibility of stating more general conditions. For example, it might be possible to merge the pre- and rely-conditions (post- and guarantee-conditions) into an assumption (promise).

Manna /79a/ uses modal logic to state and prove properties of programs. It might be necessary to design a new system in which the predicates are properties of two states. See also Sernadas /80a/.

6.1.6 Denotational Semantics

As is pointed out above, the arguments based on the operational semantics definition of the parallel language are less formal than those based on the denotational semantics of the non-deterministic sequential language. Although the operational definition could be made more formal, earlier experience with language definition styles suggests that proofs based on denotational semantics would be less cumbersome.

The denotational semantics of non-determinism are given in terms of:

1 $M[\![P]\!] \subseteq Tr$

2 $Tr = St \times St$

Resumptions can be used to record the history of a computation:

3 $Res = St \rightarrow (St \times Res)$

It is then possible to define merges of resumptions. The effect of parallelism is to force such a merge to be non-deterministic leading to denotations like:

4 $Res = St \rightarrow \wp(St \times Res)$

One of the arguments for denotational semantics is the possibility of being "fully abstract". That is, two objects with the same "behaviour" will be connected with the same denotation. This is not, however, that easy with denotations like 4 because the number of steps is retained. For example, Hennessy /80a/ observes the difficulty with:

5 x:=x; x:=x versus x:=x

An interesting way out of this problem might be to base the semantics on the notion of interference. For example:

6 $Res \rightarrow \wp(Tr)$

6.2 Communication Based Parallelism

The intricacy of the reasoning required to handle shared variables has led to a search for more tractable ways of representing parallel processes. The approach considered in this section confines all interaction between processes to communication: separate processes are not allowed to refer to each other's state.

Not surprisingly there are different flavours within the overall approach. Kahn /73a/ assumes (infinite) buffering of communications whereas CSP (Hoare /78a/) and CCS (Milner /80a/) require that all processes involved in a communication synchronise for message transmission. The slightly artificial assumption of infinite buffering can simplify some proofs. The unbuffered approach does, however, leave open the possibility of writing processes which introduce any desired degree of buffering. For this reason the CSP approach is the one discussed here. (It is also worth noting that shared variable parallelism can be simulated by defining processes in place of shared variables - see below.)

CSP can be seen as resulting from the culmination of several trends in programming language design. One aspect of CSP provides a useful generalisation of the guarded commands of Dijkstra /75a/. Thus the sort example of Dijkstra /76a/ can be generalised to work on an array A of N elements:

1 $*([\ [\!]i \in \{1: N-1\}\ ] : A(i) > A(i+1) \rightarrow swap)$

achieving a post-condition of:

2 $(\forall i \in \{1{:}N-1\} \; ; \; A(i) \leq A(i+1))$

(Note that the syntax of Hoare /78a/ is not followed too slavishly!)

To this useful extension of guarded commands is added the concept of communication. Thus a process which can be used like a shared variable might be written:

```
3   SVAR = var i;
           ( [ [] p∈ Proc]  p?i∈ Nat →
                    * ( [ [] p∈ Proc]  p?i ∈ Nat → skip
                      [] [ [] p∈ Proc]  p!i      → skip
                      )
           )
```

(In some cases the mapping of array variables onto arrays of processes requires some care. For example, in the equivalence relation problem of section 5.3 there is some danger of delay or even deadlock if too much of the procedural logic is put into the processes which simulate the array elements.)

A third trend leading to CSP-like languages is the concept of monitors which control shared variables. CSP processes not only localise storage but they also add a path-expression-like control. Consider, for example, a CSP version of the program given in sub-section 5.2.6:

```
4   ([ || i ∈ {1 .. N}   i: EL
     ||(loc: FINDP || top: TOP)
     ||...
    )
```

```
5   EL = var c;
       ( [ [] p ∈ Proc ]  p?c ∈ Val →
           *( [ [] p ∈ Proc ]  p?c ∈ Val  → skip
            [] [ [] p  Proc ]  p!c        → skip
       )    )

6   FINDP = top! N + 1;
          ( [ || i ∈ {1..N} ]  SEARCHi);
          top ? i →  ! i

7   SEARCHi = (top ? t → if i < t then
                         (i ? v ∈ Val → if p(v) then top ! i)
              )

8   TOP = var t;
         ( [ [] p ∈ Proc ]  p?t ∈ Nat →
                 *( [ [] p ∈ Proc ]  p?v ∈ Nat  →
                                  (if v < t then t:=v)
                  [] [ [] p ∈ Proc ]  p!t → skip
                  )
         )
```

Notice that both 5 and 8 require that the "variables" are initialised before use and that each value is local to one process.

An even better example of the path-expression-like control in CSP is given by studying programs for the behaviour of buffers. A simple single buffer can be represented by:

```
9   (in: ...
    || b: B1
    || out: ...
    )

10  B1 =  *(in ? e ∈ El → out ! e → skip)
```

Multiple buffering can be achieved in several ways: with appropriate naming, single buffers can simply be strung together; one process can have an internal state containing the whole buffer; a parallel collection of buffer processes can preserve the correct order of passed elements by handing batons between them. This last example allows parallel reading and writing in the buffer and is an interesting example of synchronisation:

11 Bi = var e;
(bi⊖1 ? RDB → [[p ∈ Proc] p ? e ∈ El → bi ⊕ 1! RDB →
bi⊖1 ? WRB → [[p ∈ Proc] p ! e → bi ⊕ 1! WRB →
skip)

Various semantic models are being considered for CSP. In the deterministic case the denotation of a process can be viewed as a tree or, isomorphically, sets of (prefix-closed) strings - see Hoare /80b/. A richer model is required to cope with parallelism and Hoare /81a/ uses a relation on sets of (prefix-closed) strings and "refusal sets" of alphabet symbols. Such models are the basis for the correctness of proof rules but the issues involved are sufficiently difficult that even the notion of equivalence is an active topic of research.

6.2.1 Proof Rules

As with sequential programs, the very first proofs given (Roscoe and Hoare) were proofs of program equivalence. The next stage is the proof of programs with respect to specifications and, here, the work in Zhou /81a/ is considered. Communication is constrained to occur

on channels and a trace is a sequence of communications. The external behaviour of a process is taken to be its set of possible traces. The notion of specification is to define a permissible set of traces (for example via a grammar). Zhou /81a/ gives inference rules which are proved consistent with a denotational semantics for the process language.

This approach is applied in Zhou /81b/ to the problem of protocols. There are two interesting points about this application. Firstly, it is a completely new approach to an important problem. The standard approach to "specifying" protocols is to provide an abstract program for each of the nodes (e.g. IBM/a/ uses a finite state language called FAPL). The new approach rightly rejects this algorithmic approach and states that the overall specification of a protocol is that it behaves like a buffer (cf. 6.2(10)). In addition, the specification must state under what degree of misbehaviour of the communication medium (or a lower level protocol) this ideal view can be achieved. This, of course, provides a genuine specification of WHAT the protocol should achieve.

The second interesting point about the approach of Zhou /81b/ is that it can be thought of as taking the interference approach to communication based parallelism. The (noisy) wire process being, in some sense, the inverse of the rely-condition of chapter 4.

Hoare /81b/ extends the system of Zhou /81a/ to tackle total correctness.

There is room for some doubt about a proof or development method based on stream assertions alone. The cause of doubt is similar to that which is evoked by the denial of a state in the property oriented view of abstract data types (cf. sub-section 1.6.3).

Zhou /81a/ states:

"In this paper, we regard a process as being defined not by its internal states and transitions, but rather by its externally observable behaviour ..."

This approach appears to match only some problems: it is significantly more difficult to specify a stack than a queue; the equivalence relation problem becomes unclear unless a state is used in the specification; even the problem of section 5.2/6.2 (4-8) becomes difficult purely in terms of messages.

6.2.2 Development Method

As is pointed out above, the history of the work on sequential programs suggests that work on development methods will follow the work on the proof of extant programs. Milner /80a/ moves towards a development method with the notion of "environments". Apt /80a/ applies an approach, similar to that of Owicki and Gries, to CSP programs: CSP processes are developed in isolation and then "joint cooperation between isolated proofs" is shown. The version of CSP considered includes non-deterministic guards and distributed termination. (Barringer /81a/ applies this method to Ada tasks.) A similar approach (though not covering distributed termination) is described in Levin /80a/:

i) sequential proof

ii) satisfaction proof covering possible messages

iii) non-interference proof to show that the earlier proofs "match"

As is argued in sub-section 6.1.2, this approach does not appear to meet the criteria for a development method set out in section 0.1. But the similarity with the Owicki/Gries approach does, paradoxically, offer hope that a development method could be created. The essential step is to repeat the work of chapter 4 for communication based parallelism. That is, the notion of interference should be recognised and elevated to part of the specification and proof method.

Chapter 7

Conclusions

It is pointed out in the introduction that the current dissertation is a report on work-in-progress rather than a completely satisfactory, all-embracing, program development method. It is now appropriate to review some of the limitations and directions for further work.

7.1 Achievements

It is claimed that the methods described for the development of sequential programs are satisfactory. Specifications in the VDM style have been written for both a large number and a wide range of systems; the proof rules have also been shown to be useful in a wide range of examples. There is still, clearly, a great deal to be done in the dissemination of these methods and some of the technical problems to be solved in order to make this possible are themselves far from trivial (cf. section 7.4). The main contribution of this dissertation is the sounder basis provided for the methods used in Jones /80a/ especially as concerns non-deterministic programs.

The claims in the area of developing parallel programs are more modest. The foremost limitation stems from the concentration on the shared variable view of parallelism. Within this class of problems, however, it does appear that the attempt to meet head-on the notion of interference has worked. The examples illustrate that a development method (in the sense of section 0.1) can be built around specifications which record the acceptable and potential interference. The proof rules employed will certainly continue to evolve, but even in their current form they have proved useful guides to design.

7.2 Limitations

Within the field of shared-variable parallelism, the notion of synchronisation has not been treated. As is pointed out in sub-section 6.1.5 and section 7.3, an extended form of logic notation may be necessary to handle such issues as deadlock. It is, however, hoped that the additional specification required can be fitted into the framework of rely- and guarantee-conditions.

The program developed in section 5.2 is arranged so as to terminate gracefully; that in section 5.3 makes no provision for termination. The general problem of "distributed termination" (cf. Francez /80a/) is one which requires greater study.

A specific problem with the scheme of proof proposed in chapter 4 manifests itself when a combined effect on several variables is to be described. One example where this can be seen is if an additional Boolean output is required in the example of section 5.2 (also the example given in 4.1(18,19)).

7.3 Further Work

The necessary extensions to cope with synchronisation problems might be tackled by using some temporal logic (cf. sub-section 6.1.5). An alternative approach proposed in Lauer /81a/ is to use only language (construct)s about which it is possible to establish freedom from deadlock etc.

The incentives to find a denotational semantics for the parallel language (full abstraction etc.) are stated elsewhere. It is clear that the least constrained parallel languages tend to force a rather operational view of what is meant by programs. There is, however, some hope that the higher level constructs like the "rendezvous" of the Ada language might be easier to define denotationally than operationally.

The parallel designs shown here are all fairly obvious relaxations of sequential thinking. An important aim for the future would be methods that encouraged the use of new, more radical, approaches to the use of parallelism.

If the communication view of parallelism is to supplant the shared variable approach, it will be important to carry the notion of interference over to the new area. As is pointed out, this has already been achieved in the area of protocol validation. The question still remains whether the step from the proof method of Owicki /76a/ to the development method discussed here can be repeated based on, for example, Levin /80a/ or Apt /80a/.

7.4 Bringing into Practice

The aim of the overall endeavour, of which this dissertation is but a part, is to bring systematic program development methods into use by software engineers. Many of the problems are common to both sequential and parallel programs.

One useful aid would be the development of a machine support system for handling development histories. What should such a system do? The temptation to provide a system for "executing specifications" should be resisted - this inevitably leads users away from the proper goal of a specification as an abstract (thinking) model. Some useful objectives might be:

i) Syntax checking of formulae

ii) Type checking of function arguments

iii) Information retrieval

iv) To support routine operations like substitution

v) To provide appropriate instances of proof rules

vi) To handle simple proof checking or generation

vii) To trace the impact of changes

viii) To draw consequences from definitions

These objectives will require the adoption of a fairly rigid syntax for definitions etc. Care will have to be taken that this does not unnecessarily hinder expression. It is for this reason that a support tool is seen more as a series of useful functions which can be invoked rather than a controlling environment.

The progress of rigorous methods will largely be governed by the extent to which their use becomes easier with time. It is therefore crucial that any support system should aid with the collection of general results like the theories of data types discussed above.

Earlier experience with the sequential parts of VDM have shown the importance of educational material in getting systematic methods adopted. There is clearly an urgent need for a soundly based course on the design of parallel programs.

References

Abrial /79a/: Non-Deterministic System Specification, by J-R. Abrial and S.A. Schuman. In Semantics of Concurrent Computation (Proc. Evian, France), Springer-Verlag, LNCS No.70.

Abrial /79b/: Specification Language, by J-R. Abrial, S.A. Schuman and B. Meyer. In Construction of Programs (ed. McKeag), Cambridge University Press.

Abrial /80a/: The Specification Language Z: Syntax and Semantics, by J-R. Abrial, PRG, Oxford University.

Abrial /80b/: The Specification Language Z: Basic Library, by J-R. Abrial, PRG, Oxford University.

Abrial /82a/: Book to be published by J-R. Abrial, Prentice-Hall Int.

Adams /81a/: On Proof Rules for Monitors, by J.M. Adams and A.P. Black (to be published).

Allen /72a/: A Formal Definition of Algol 60, by C.D. Allen, D.N. Chapman and C.B. Jones, IBM Hursley, TR12.105.

Allen /72b/: The Formal Development of Algorithms, by C.D. Allen and C.B. Jones. IBM Hursley, TR12.110.

Apt /80a/: A Proof System for Communicating Sequential Processes, by K.R. Apt, N. Francez and W.P. de Roever. ACM Toplas, Vol. 2, No.3, pp. 359-385.

Backus /78a/: Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, by J. Backus, CACM, Vol. 21, No.8, pp. 613-641.

Barringer /81a/: Axioms and Proof Rules for Ada Tasks, by H. Barringer and I. Mearns. Dept of Computer Science, Manchester University, Internal document.

Bekić /74a/: A Formal Definition of a PL/I Subset (2 parts), by H. Bekić, D. Bjørner, W. Henhapl, C.B. Jones and P. Lucas. IBM Vienna, TR25.139.

Bjørner /78a/: The Vienna Development Method: The Meta-Language, by D. Bjørner and C.B. Jones (eds), Springer-Verlag Lecture Notes in Computer Science, No.61.

Bjørner /80a/: Abstract Software Specifications, by D. Bjørner (ed.), Springer, LNCS No.86.

Bjørner /81a/: Towards a Formal Description of Ada, by D. Bjørner and O.N. Oest (eds), Springer, LNCS, No.98.

Bothe /79a/: Specification and Verification of Abstract Data Types, by K. Bothe, Humbolt-Universitaet zu Berlin Mathematik Seminarbericht Nr 13.

Bothe /80a/: A Generalised Abstract Data Type Concept, by K. Bothe, Humbolt-Universitaet, Berlin, Preprint No.3.

Brinch Hansen /73a/: Operating System Principles, by P. Brinch Hansen, Prentice-Hall.

Bron /76a/: A Proposal for Dealing with Abnormal Termination of Programs, by C. Bron, M.M. Fokkinga and A.C.M. de Naas, Dept App. Maths, Twente Univ. of Technology, Memo Nr.150.

Bron /77a/: Exchanging Robustness of a Program for a Relaxation of its Specification, by C. Bron and M.M. Fokkinga, Mem Nr.178, Dept App. Maths, Twente Univ. of Technology.

Burge /75a/: Recursive Programming Techniques, by W.H. Burge, Addison-Wesley.

Burstall /69a/: Proving Properties of Programs by Structural Induction, by R. Burstall, CJ, Vol. 12, No.1.

Burstall /74a/: Program Proving as Hand Simulation with a Little Induction, by R.M. Burstall, Procs IFIP Congress 1974, pp. 308-312, North Holland.

Burstall /77a/: Putting Theories Together to Make Specifications, by R.M. Burstall and J.A. Goguen, Int. Jt Conf on AI, Boston.

Burstall /80a/: An Informal Introduction to Specifications using CLEAR, by R.M. Burstall and J.A. Goguen.

Burstall /80b/: The Semantics of CLEAR, a Specification Language, by R.M. Burstall and J.A. Goguen, in Bjørner /80a/.

Cooper /66a/: The Equivalence of Certain Computations, by D.C. Cooper, BCS Comp J., Vol. 9, No.1.

Dahl /78a/: Can Program Proving be made Practical? by O-J. Dahl, Lectures presented at EEC-CREST course on Prog. Foundations, Toulouse.

de Bakker /73a/: A Calculus for Recursive Program Schemes, by J.W. de Bakker and W.P. de Roever, in Automata, Languages and Programming, ed. Nivat, North-Holland.

Dershowitz /79a/: Proving Termination with Multiset Orderings, by N. Dershowitz and Z. Manna, Comm. ACM 22/8, Stanford.

Dijkstra /68a/: Co-operating Sequential Processes, by E.W. Dijkstra, Nato Advanced Study Institute, Academic Press.

Dijkstra /75a/: Guarded Commands, Non-determinacy and Formal Derivation of Programs, by E.W. Dijkstra, CACM, Vol. 18, No.8.

Dijkstra /76a/: A Discipline of Programming, by E.W. Dijkstra, Prentice-Hall.

Dijkstra /78a/: On-the-Fly Garbage Collection: An Exercise in Cooperation, by E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.M.F. Steffens, Comm ACM, Vol. 21, No.11, pp. 966-974.

DoD /80a/: Reference Manual for the Ada Programming Language (Proposed Standard Document), U.S. Dept of Defense.

Donahue /75a/: Complementary Definitions of Programming Language Semantics, by J. Donahue, Univ. Toronto Tech. Report. CSRG-6Z.

Dungan /79a/: Bibliography on Data Types, by D.M. Dungan, ACM Sigplan Notices, Vol. 14, No.11, pp. 31-59.

ECMA /76a/: American National Standard Programming Language PL/I, ECMA TC 10 and ANSI. x3.53-1976.

Ehrig /80a/: Algebraic Implementation of Abstract Data Types, by H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz, Technische Universität Berlin, 80-32.

Ehrig /81a/: Algebraic Theory of Parameterised Specifications with Requirements, by H. Ehrig, 6th CAAP, Genova, Italy.

Fielding /80a/: The Specification of Abstract Mappings and their Implementation as B^+-Trees, by E. Fielding, Oxford Univ., Monograph PRG-18.

Floyd /67a/: Assigning Meanings to Programs, by R.W. Floyd, Proc. of Symposia in Appl. Math., Vol. 19.

Francez /80a/: Distributed Termination, by N. Francez, ACM TOPLAS, Vol. 2, No.1.

Goguen /75a/: Abstract Data Types as Initial Algebras and the Correctness of Data Representations, by J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, IEEE Conference Proc. - Computer Graphics, Pattern Recog. and Data Structure, IEEE Cat. No. 75CH0981-1C.

Goldstine /47a/: Planning and Coding of Problems for an Electronic Computing Instrument, by H.H. Goldstine and J. von Neumann, Report for the U.S. Army Ord. Dept.

Gries /79a/: Is Sometime Ever Better than Alway ? by D. Gries, ACM TOPLAS, Vol. 1, No. 2, pp. 258-267.

Guttag /77a/: Abstract Data Types and the Development of Data Structures, by J. Guttag, Comm ACM, Vol. 20, No. 6.

Guttag /80a/: Formal Specification as a Design Tool, by J. Guttag and J.J. Horning, Proceedings of ACM POPL conference, pp. 251-261.

Hantler /75a/: EFFIGY Reference Manual, by S.L. Hantler and A.C. Chibib, IBM (Research) RC 5225.

Hehner /79a/: do considered od: A Contribution to the Programming Calculus, by E.C.R. Hehner, Acta Informatica, Vol. 11, pp. 287-304.

Henderson /76a/: A Lazy Evaluator, by P. Henderson and J.H. Morris, Third Symposium, POPL.

Henderson /80a/: Functional Programming Application and Implementation, by P. Henderson, Prentice-Hall Int.

Hennessy /80a/: Full Abstraction for a Simple Parallel Programming Language, by M.C.B. Hennessy and G.D. Plotkin.

Hitchcock /72a/: Induction Rules and Termination Proofs, by P. Hitchcock and D. Park, IRIA Procs.

Hitchcock /74a/: An Approach to Formal Reasoning about Programs, by P. Hitchcock, Ph.D. Thesis, Warwick Univ.

Hoare /69a/: An Axiomatic Basis of Computer Programming, by C. Hoare, CACM.

Hoare /71a/: Proof of a Program - FIND, by C.A.R. Hoare, CACM, Vol. 14, pp. 39-45.

Hoare /72b/: Proof of Correctness of Data Representations, by C.A.R. Hoare, Acta Informatica, Vol. 1, pp. 271-282.

Hoare /74a/: Monitors: An Operating System Structuring Concept, by C.A.R. Hoare, Comm ACM 17/10, pp. 549-557.

Hoare /75a/: Parallel Programming: An Axiomatic Approach, by C.A.R. Hoare, In Computer Langs, Vol. 1, pp. 151-160, Pergamon Press.

Hoare /78a/: Communicating Sequential Processes, by C.A.R. Hoare, CACM, Vol. 21, No.8, pp. 666-677.

Hoare /80a/: Oxford M.Sc. Lectures, by C.A.R. Hoare.

Hoare /80b/: A Model of Communicating Sequential Processes, by C.A.R. Hoare, in On the Construction of Programs, pp. 229-254, CUP.

Hoare /81a/: A Non-deterministic Model for Communicating Sequential Processes, by C.A.R. Hoare, S.D. Brookes and A.W. Roscoe.

Hoare /81b/: A Calculus of Total Correctness for Communicating Processes, by C.A.R. Hoare, Oxford University, PRG Monograph 23.

IBM /a/: Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic.

Jackson /75a/: Principles of Program Design, by M.A. Jackson, Academic Press.

Jackson /78a/: Information Systems: Modelling, Sequencing and Transformations, by M.A. Jackson, Proc. 3rd Int. Conf. Soft Eng., IEEE.

Jackson /80a/: Structural Design of the Information System, by M.A. Jackson.

Jones /72a/: Formal Development of Correct Algorithms: An Example Based on Earley's Recogniser, by C.B. Jones, SIGPLAN Conf., SIGPLAN NOT 7/1.

Jones /73a/: Formal Development of Programs, by C.B. Jones, IBM Hursley Lab., TR12.117.

Jones /77a/: Structured Design and Coding: Theory versus Practice, by C.B. Jones, Informatie, Jaargang 19, nr 6, pp. 311-319.

Jones /77b/: Implementation Bias in Constructive Specifications of Abstract Objects, by C.B. Jones.

Jones /78a/: The Meta-Language: A Reference Manual, by C.B. Jones, in Bjørner /78a/.

Jones /78b/: Denotational Semantics of Goto: An Exit Formulation and its Relation to Continuations, by C.B. Jones, in Bjørner /78a/.

Jones /79a/: Constructing a Theory of a Data Structure as an Aid to Program Development, by C.B. Jones, Acta Informatica, Vol. 11, pp. 119-137.

Jones /80a/: Software Development: A Rigorous Approach, by C.B. Jones, Prentice-Hall Int.

Jones /80b/: The Rôle of Formal Specifications in Software Development, by C.B. Jones, in Life Cycle Management, Infotech State of the Art Report, Ser. 8, No. 7.

Jones /80c/: Tentative Steps Towards a Development Method for Interfering Programs, by C.B. Jones (to be published).

Jones /81a/: Towards More Formal Specifications, by C.B. Jones, in Software Engineering Entwurf u. Specifikation. (Proc. Berlin), Tübner Verlag.

Kahn /73a/: A Preliminary Theory for Parallel Programs, by G. Kahn, IRIA report, No. 6.

Kahn /77a/: Coroutines and Networks of Parallel Processes, by G. Kahn and D.B. MacQueen, in IFIP 77 procs.

Kamin /80a/: Final Data Type Specifications: a New Data Type Specification Method, by S. Kamin, Proceedings of ACM POPL conference, pp. 131-138.

King /76a/: Symbolic Execution and Program Testing, by J.C. King, Comm. ACM, Vol. 19, No. 7.

Knuth /73a/: The Art of Computer Programming: Vol. 3/Sorting and Searching, by D.E. Knuth, Addison-Wesley.

Lamsweerde /76a/: Formal Derivation of Strongly Correct Parallel Programs, by A. van Lamsweerde and M. Sintzoff, M.B.L.E., Report R338.

Lauer /71a/: Consistent Formal Theories of the Semantics of Programming Languages, by P.E. Lauer (Thesis), IBM, TR25.121.

Lauer /81a/: COSY. An Environment for Development and Analysis of Concurrent and Distributed Systems, by P.E. Lauer and M.W. Shields, in Software Engineering Environments, (ed) H. Huenke, North-Holland.

Lehmann /78a/: Algebraic Specification of Data Types: a Synthetic Approach, by D.J. Lehmann and M.B. Smyth, Univ. of Leeds, Dept of CS, Report 115.

Lengauer /81a/: A Methodology for Programming with Concurrency, by C. Lengauer and E.C.R. Hehner, talk at CONPAR 81, Nurenberg, Germany.

Levin /80a/: Proof Rules for Communicating Sequential Processes, by G.M. Levin, thesis, Cornell Univ.

Lindsay /79a/: Notes on Distributed Databases, by B.G. Lindsay et al. IBM Research Report RJ 2571.

Linger /79a/: Structured Programming, Theory and Practice, by R.C. Linger, H.D. Mills and B.J. Witt, Addison-Wesley.

Lovengreen /80a/: Parallelism in ADA, by N.H. Voengreen, Master Thesis, Tech. Univ. Denmark.

Lucas /68a/: Two Constructive Realisations of the Block Concept and their Equivalence, by P. Lucas, IBM Lab. Vienna, ULD-Version 2, Tr 25.085.

Lucas /68b/: On the Formal Description of PL/I, by P. Lucas and K. Walk, Annual Review in Automatic Programming, Vol. 6, Part 3, pp. 105-181, Pergamon Press, New York, London.

Lucas /69a/: On the Documentation of Programming Ideas, by P. Lucas and K. Walk, IBM Lab Vienna, LN 25.3.064.

MacLane /79a/: Algebra (2nd ed), by S. MacLane and G. Birkoff, Macmillan.

Manna /69a/: Properties of Programs and the First-Order Predicate Calculus, by Z. Manna, J ACM, Vol. 16, No.2.

Manna /78a/: Is "sometime" Sometimes Better than "always"? by Z. Manna and R. Waldinger, Comm ACM, Vol. 21, No. 2, pp. 159-172.

Manna /79a/: The Modal Logic of Programs, by Z. Manna and A. Pnueli, Sixth ICALP Conference, Graz.

Milner /71a/: An Algebraic Definition of Simulation between Programs, by R. Milner, Stanford, Memo AIM-142, Report No. CS-205.

Milner /80a/: A Calculus of Communication Systems, by R. Milner, Springer Verlag, Lecture Notes in Computer Science, Vol. 92.

Morris /72a/: A Correctness Proving Using Recursively Defined Functions, by J.H. Morris Jr., in Rustin /72a/.

Morris /73a/: Advice on Structuring Compilers and Proving them Correct, by F.L. Morris, ACM Symposium on POPL, Boston.

Mosses /77a/: Making Denotational Semantics Less Concrete, by P.D. Mosses, Arhus University.

Naur /66a/: Proof of Algorithms by General Snapshots, by P. Naur, BIT 6, pp. 310-316.

Naur /69a/: Programming by Action Clusters, by P. Naur, BIT, Vol. 9, pp. 250-258.

Naur /72a/: An Experiment on Program Development, by P. Naur, BIT, Vol. 12, pp. 347-365.

Naur /74a/: Concise Survey or Computer Methods, by P. Naur, Student Litteratur.

Needham /72a/: Protection Systems and Protection Implementations Capability Addressing via Indirection, by R. Needham, FJCC, APIPS Conference Publications, Vol. 41, pt I, pp. 571-578.

Owicki /75a/: Axiomatic Proof Techniques for Parallel Programs, by S.S. Owicki, thesis, Dept of Computer Science, Cornell University, TR 75-251.

Owicki /76a/: Verifying Properties of Parallel Programs: An Axiomatic Approach, by S.S. Owicki and D. Gries, Comm. ACM, Vol. 19, No. 5, pp. 279-285.

Park /80a/: On the Semantics of Fair Parallelism, by D.M.R. Park, in Bjørner /80a/.

Plotkin /76a/: A Powerdomain Construction, by G.D. Plotkin, SIAM J. Comput., Vol. 5, No. 3.

Pnueli /79a/: The Temporal Semantics of Concurrent Programs, by A. Pnueli, in Semantics of Concurrent Computation (Proc. Evian, France), Springer-Verlag.

Randell /78a/: Reliability Issues in Computing System Design, by B. Randell, P.A. Lee and P.C. Treleaven, ACM Comp. Surv., Vol. 10, No. 2, pp. 123-166.

Reynolds /79a/: Reasoning about Arrays, by J.C. Reynolds, CACM, Vol. 22, No. 5, p. 290.

Reynolds /81a/: The Craft of Programming, by J.C. Reynolds, Prentice-Hall Int.

Rustin /72a/: Formal Semantics of Programming Languages, by R. Rustin, Prentice-Hall.

Schonberg /81a/: An Automatic Technique for Selection of Data Representations in SETL Programs, by E. Schonberg, J.T. Schwartz and M. Sharir, ACM TOPLAS, Vol. 3, No. 2, pp. 126-143.

Scott /81a/: Lectures on the Mathematical Theory of Computation, by D.S. Scott, to be published as an Oxford PRG monograph.

Sernadas /80a/: Temporal Aspects of Logical Procedure Definition, by A. Sernadas, Information Systems, Vol. 5, pp. 167-187.

Smyth /78a/: Power Domains, by M. Smyth, J. Comp. Sys. Sci., Vol. 16, pp. 23-26.

Stoy /77a/: Denotational Semantics - the Scott-Strachay Approach to Programming Language Theory, by J.E. Stoy, MIT Press.

Tarjan /75a/: Efficiency of a Good but not Linear Set Union Algorithm, by R.E. Tarjan, J. ACM, Vol. 22, No. 2, pp. 215-225.

Tennent /81a/: Principles of Programming Languages, by R.D. Tennent, Prentice-Hall Int.

Turing /49a/: Checking a Large Routine, by A.M. Turing, in Report of a Conference on High Speed Automatic Calculating Machines, Univ. Math. Lab., Cambridge, pp. 67-69.

Veloso /79a/: Traversible Stack with Fewer Errors, by P.A.S. Veloso, ACM Sigplan Notices, 14/2, pp. 55-59.

Veloso /79b/: Traversible Stack with Fewer Errors: Addenda and Corrigenda, by P.A.S. Veloso, ACM Sigplan Notices 14/10, p. 76.

Walk /69a/: Abstract Syntax and Interpretation of PL/I, by K. Walk et al., IBM Lab Vienna, ULD-Version 3, TR 25.098.

Wand /77a/: Final Algebra Semantics and Data Type Extensions, by M. Wand, Univ. of Indiana, Comp. Science Dept, TR-65.

Welsh /80a/: Structured System Programming, by J. Welsh and M McKeag, Prentice-Hall International.

Wirth /71a/: Program Development by Stepwise Refinement, by N. Wirth, CACM, Vol. 14, No. 4.

Wulf /76a/: Abstraction and Verification in Alphard, by Wm. A. Wulf, R.L. London and M. Shaw, in New Directions in Algorithmic Languages, (ed.) S. Schuman.

Zhou /81a/: Partial Correctness of Communicating Sequential Processes, by Zhou Chao Chen and C.A.R. Hoare, Procs of Int. Conf. on Distributed Processes, Paris.

Zhou /81b/: Partial Correctness of Communication Protocols, by Zhou Chao Chen and C.A.R. Hoare, NPL workshop.

Zemanek /80a/: Abstract Architectures, by H. Zemanek, in Bjørner /80a/.

Zilles /80a/: An Introduction to Data Algebras, by S.N. Zilles, in Bjørner /80a/.

PROGRAMMING RESEARCH GROUP TECHNICAL MONOGRAPHS

JUNE 1981

This is a series of technical monographs on topics in the field of computation. Copies may be obtained from the Programming Research Group, (Technical Monographs), 45 Banbury Road, Oxford, OX2 6PE, England.

PRG-1 *(out of print)*

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-4 *(cancelled)*

PRG-5 Dana Scott
Data Types as Lattices

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages

PRG-7 Dana Scott
Continuous Lattices

PRG-8 Joseph Stoy and Christopher Strachey
OS6 – an Experimental Operating System for a Small Computer

PRG-9 Christopher Strachey and Joseph Stoy
The Text of OSPub

PRG-10 Christopher Strachey
The Varieties of Programming Language

PRG-11 Christopher Strachey and Christopher P. Wadsworth
Continuations: A Mathematical Semantics for Handling Full Jumps

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

PRG-13 Robert Milne
The Formal Semantics of Computer Languages and their Implementations

PRG-14 Shan S. Kuo, Michael H. Linck and Sohrab Saadat
A Guide to Communicating Sequential Processes

PRG-15 Joseph Stoy
The Congruence of Two Programming Language Definitions

PRG-16 C. A. R. Hoare, S. D. Brookes and A. W. Roscoe
A Theory of Communicating Sequential Processes

PRG-17 Andrew P. Black
Report on the Programming Notation 3R

PRG-18 Elizabeth Fielding
The Specification of Abstract Mappings and their Implementation as B^+-trees

PRG-19 Dana Scott
Lectures on a Mathematical Theory of Computation

PRG-20 Zhou Chao Chen and C. A. R. Hoare
Partial Correctness of Communicating Processes and Protocols

PRG-21 Bernard Sufrin
Formal Specification of a Display Editor

PRG-22 C. A. R. Hoare
A Model for Communicating Sequential Processes

PRG-23 C. A. R. Hoare
A Calculus of Total Correctness for Communicating Processes

PRG-24 Bernard Sufrin
Reading Formal Specifications