

SPECIFICATION CASE STUDIES

Second Edition

**Copyright © 1987, 1992 Prentice Hall International
(UK) Ltd**

**Appendices A and B may be copied for educational
purposes.**

**Edited by
Ian Hayes**

**With Contributions by
Bill Flinn
Roger Gimson
Steve King
Carroll Morgan
Ib Holm Sørensen
Bernard Sufrin**

Contents

I	Tutorials	1
1	Small examples of specification using mathematics	3
	<i>Ian Hayes</i>	
1.1	Introduction	3
1.2	A symbol table	3
1.3	File update	7
1.4	Sorting	9
1.5	Solutions to exercises	11
2	Block-structured symbol table	13
	<i>Ian Hayes</i>	
2.1	Introduction	13
2.2	Symbol table	13
2.2.1	The state	14
2.2.2	Operations	14
2.2.3	Errors	16
2.3	Block-structured symbol table	18
2.3.1	The state	18
2.3.2	Operations	20
2.3.3	Errors	23
2.4	Other approaches	25
2.5	Solutions to exercises	25
3	Telephone network	29
	<i>Carroll Morgan</i>	
3.1	Introduction	29
3.2	The specification	29
3.2.1	Call	31
3.2.2	HangUp	31
3.2.3	Engaged	32
3.3	Exercises	32
3.4	Solutions to exercises	33
3.5	Supplementary exercises	37

II	Software engineering	39
4	Unix filing system	41
	<i>Carroll Morgan and Bernard Sufrin</i>	
4.1	Introduction	41
4.2	Scope of the specification	43
4.3	The specification	43
4.3.1	Bytes and files	43
4.3.2	Reading and writing	44
4.3.3	File storage	47
4.3.4	Reading and writing stored files – framing	48
4.3.5	Hiding and simplification	50
4.3.6	Sequential access to files	50
4.3.7	Channel system	52
4.3.8	The access system	52
4.3.9	A file naming system	55
4.3.10	Pathnames and directories	56
4.3.11	Directories are files	57
4.3.12	The complete filing system	57
4.3.13	Honesty of definitions	60
4.3.14	Observation renaming and schema composition	61
4.3.15	Definition of error conditions	63
4.4	Summary	64
4.5	Appendix: differences from Unix	67
4.5.1	File size	67
4.5.2	Directory size	67
4.5.3	Storage medium capacity	68
4.5.4	Seek	68
4.5.5	Representation of numbers	69
5	CAVIAR	71
	<i>Bill Flinn and Ib Holm Sørensen</i>	
5.1	Introduction	71
5.2	The case study	72
5.3	Identification of the basic sets	73
5.4	The subsystems of CAVIAR	73
5.5	Modules	74
5.6	Module: <i>Resource_User</i> [T, R, U]	75
5.7	Specialisations of the resource–user system	79
5.7.1	Module: <i>Exclusive_Resource</i> [T, R, U]	79
5.7.2	Module: <i>Sole_Resource</i> [T, R, U]	80
5.7.3	Module: <i>Sole_Exclusive_Resource</i> [T, R, U]	81
5.7.4	The specification library	81
5.8	Classification and instantiation	81
5.8.1	Some laws for CAVIAR	81
5.8.2	Matching systems with models	83
5.8.3	Module: <i>Hotel_Reservation</i>	83
5.8.4	Module: <i>Transport_Reservation</i>	84
5.9	The meeting attendance subsystem	85
5.9.1	Module: <i>Resource_Pool</i> [T, X]	85

5.9.2	Module: <i>Meeting_Visitor</i>	86
5.10	The meeting resource subsystems	87
5.10.1	Module: <i>Diary_System</i> [<i>T</i> , <i>X</i> , <i>IX</i>]	87
5.10.2	Module: <i>Conference_Room_Booking</i>	88
5.10.3	Module: <i>Dining_Room_Booking</i>	90
5.10.4	Module: <i>Visitor_Pool</i>	91
5.10.5	The construction process	92
5.11	Module: <i>CAVIAR</i>	92
5.11.1	Combining subsystems to form the state	93
5.11.2	Operations that involve meetings only	94
5.11.3	Operations that involve visitors only	96
5.11.4	A general visitor removal operation	96
5.12	Conclusion	97
6	ICL Data Dictionary	99
	<i>Bernard Sufrin</i>	
6.1	Introduction	100
6.2	Overview of the Data Dictionary System	100
6.3	Access control	101
6.3.1	Abstract information structures of DDS	101
6.4	DDS dynamics: Part 1	106
6.4.1	The state of a running DDS	106
6.4.2	The display command	109
6.4.3	Setting the element context	109
6.5	Access-control information	110
6.6	DDS dynamics: Part 2	113
6.6.1	Inserting elements	113
6.6.2	Deleting elements	115
6.7	Prospects	115
6.8	Appendix: potential simplifications	116
7	Flexitime specification	121
	<i>Ian Hayes</i>	
7.1	Introduction	121
7.2	State	121
7.3	Operations	122
8	Simple assembler	125
	<i>Ib Holm Sørensen and Bernard Sufrin</i>	
8.1	Introduction	125
8.1.1	The structure of instructions	126
8.2	Requirements	127
8.2.1	Symbol definitions	127
8.2.2	Symbolic operands	128
8.2.3	Numeric operands	128
8.2.4	Symbolic opcodes	128
8.2.5	Operands of machine instructions	129
8.2.6	Opcode fields	129
8.2.7	Specification summary	130
8.2.8	Consequences of the specification	130

8.2.9	Discussion	132
8.3	High-level design	132
8.3.1	Design of the first phase	133
8.3.2	Design of the second phase	133
8.3.3	Putting the phases together	134
8.3.4	Correctness of the design	134
8.3.5	Discussion	136
III	Distributed Computing	137
9	The role of mathematical specifications	139
	<i>Roger Gimson and Carroll Morgan</i>	
9.1	Introduction	139
9.2	A first example	140
9.3	The first compromises	141
9.4	A compromise avoided	145
9.5	Modularity and composition of services	147
9.6	Conclusions	149
10	Authentication of usernames	151
	<i>Roger Gimson and Carroll Morgan</i>	
10.1	Nicknames and usernames	151
10.2	Authentication	151
10.3	Guest user	152
11	Time service – user manual	153
	<i>Roger Gimson and Carroll Morgan</i>	
11.1	Time service operation	153
11.2	Error reports	154
11.3	Modula-2 interface	154
12	Reservation service – user manual	155
	<i>Roger Gimson and Carroll Morgan</i>	
12.1	Introduction	155
12.2	Reservation service operations	157
12.3	Error reports	160
12.4	Modula-2 interface	161
IV	Transaction Processing	163
13	Application to industry	165
	<i>Ian Hayes</i>	
13.1	Introduction	165
13.2	Uses of formal specification	167
13.3	The specification process	168
13.3.1	Notation	169
13.4	A sample specification	169
13.4.1	Exceptional conditions specification	170
13.4.2	The state	170

13.4.3	The operations	171
13.4.4	Exception checking	172
13.5	Questions raised	173
13.5.1	Exceptional conditions	173
13.5.2	Interval control	175
13.5.3	Interaction between modules	176
13.6	Problems with specification	176
13.6.1	Communication problems	176
13.6.2	The right level of abstraction	177
13.6.3	Technical problems	177
13.7	Conclusions	178
13.8	Appendix: exceptional conditions manual	179
14	CICS restructure	183
	<i>Steve King</i>	
14.1	Introduction	183
14.2	The CICS program product	183
14.3	Early experiments	184
14.4	The decision to use Z	185
14.5	Education and tools	186
14.6	Results	187
14.6.1	Subjective results	187
14.6.2	Quantitative results	188
14.7	The Oxford–Hursley collaboration	190
14.8	Conclusions	192
15	CICS API specification	193
	<i>Steve King</i>	
15.1	Introduction	193
15.2	Using Z to describe interfaces	194
15.3	The CICS Application Programming Interface (API)	194
15.4	Reasons for specifying the API	196
15.5	How the specifications were written	197
15.6	Experiences	198
15.6.1	Communication problems	198
15.6.2	The ‘right’ level of abstraction	198
15.6.3	Putting modules together	199
15.6.4	Parallelism	200
15.6.5	Distributed systems	200
15.7	Results	201
15.8	Conclusions	202
16	CICS Temporary Storage	203
	<i>Ian Hayes</i>	
16.1	A single queue	203
16.1.1	Operations	204
16.1.2	Errors	207
16.2	Named queues	208
16.3	A network of systems	210
16.3.1	A note on the current implementation	211

17 CICS message system	213
<i>Ian Hayes</i>	
17.1 Message output	213
17.2 Multiple destinations	214
17.3 Message input	214
17.4 Send and receive	215
17.5 Combining input and output	215
17.6 Logical names	215
17.7 Multiple logical destinations	216
17.8 Domains of the operations	217
 V Appendices	 219
A Glossary: Z mathematical notation	221
A.1 Definitions and declarations	221
A.2 Axiomatic definitions	222
A.3 Generic definitions	222
A.4 Logic	223
A.5 Sets	224
A.6 Numbers	225
A.7 Binary relations	226
A.8 Functions	228
A.9 Orders	229
A.10 Sequences	229
A.11 Bags	231
A.12 Generalised bags	233
A.13 Free type definitions	234
 B Glossary: Z schema notation	 235
B.1 Schema definition	235
B.2 Schema operators	235
B.3 Operation schemas	239
B.4 Operation schema operators	240
 Index	 249

Foreword

Reading formal texts is like meeting people. Sometimes, you understand them straightaway like good friends who take to each other immediately. At other times, it is more difficult. You may parse what you read, but find it impossible to work out any meaning. With the latter, you have to be patient, ask questions, explore the surroundings; in other words, it is better for you to be introduced through some common good friends who volunteer to help you prepare for the first meeting.

Large computer programs, to say the least, pertain to the category of formal texts whose meanings are not immediately obvious! For that reason, people have been trying – for some time – to find out what kind of intermediate text would be best suited to play the role of go-between.

This book reports on experiments made at Oxford University within this framework: it shows how one may *communicate ideas and meanings* about existing (or even not-yet-written) computer programs, and this by using conventional mathematical notations of ordinary logic and elementary set theory.

The choice of a “standard” mathematical notation offers many advantages: it is easy for a scientifically trained reader to understand; it is rigorous; it denotes rich concepts (e.g., functions and their usual attributes: partiality *vs.* totality, domain, range, etc.); and it is an open notation, because you may enlarge it at will.

Liberated from the burden of obeying the idiosyncracies of a particular language, the authors of the various papers forming this book were free to experiment with *various styles* depending on the problem at hand (but also on their personal taste). In reading the book, I found it very exciting to discover how each situation was formalised in a way different from that of others (or from what I had in mind). This variety of style is reassuring: it indicates, if at all necessary, that there does not exist any “normal” way of describing things rigorously.

However, despite this variety, all authors seem to have encountered at some point or another a difficulty of the same nature—namely that of structuring the formal text. To this common problem they decided to give a common answer in the form of what is called a *Schema*, together with a corresponding *Schema Calculus*.

Roughly speaking, a Schema is a box within which certain *variables of interest* are together declared, given types, and mutually constrained. The Schema Calculus gives rules by which these boxes can be transformed or combined to produce other boxes. The main advantages of this mechanism are its simplicity and immediate “visibility”.

In this book, the main emphasis has been put on using ordinary mathematical notations in order to describe (*specify*) computer programs. Another important outcome of any mathematical approach is that of performing proofs: this will be vital, of course, in the process of *software design* by which specifications are gradually transformed in order to obtain concrete programs. Here is the subject of another book.

J.-R. Abrial
Paris, February 1986

Preface to the first edition

Over the last six years the Programming Research Group of the Oxford University Computing Laboratory has been the home for a number of projects which made extensive use of mathematics for the specification of computer systems. The style of specification which is used in this monograph emerged as a result of experience gained on these projects. Specifications are presented using the notation known as Z, which has evolved somewhat since it was originally introduced to us by Jean-Raymond Abrial.

The Software Engineering Project, which began in 1978, has been the backbone of the specification work. Both Tony Hoare and Bernard Sufrin were associated with the project since its inception. Project staff have included Jean-Raymond Abrial, Tim Clement, Martin Raskovsky, Ib Holm Sørensen, Stefan Sokolowski, and Phil Wadler.

The Distributed Computing Project began in 1982. Its goal was the specification and construction of a loosely-coupled distributed operating system based on the model of autonomous clients having access to shared services. Roger Gimson and Carroll Morgan have been with the project since it began. The application of mathematical specifications to distributed systems has had beneficial consequences both on the style of specifications and on the design of the systems which were produced.

The Transaction Processing Project also began in 1982. It was a collaborative project with IBM initiated to meet the challenge of applying mathematical specification methods in a commercial environment. From the outset Ib Holm Sørensen has run the project from the Oxford end and Peter Collins and John Nicholls have taken responsibility from the IBM end. Tim Clement was associated with the project during its first year; his successor was Ian Hayes. Rod Burstall, Cliff Jones, and Tony Hoare have acted as consultants. The project has demonstrated the benefits to be gained from applying the methods to existing software.

None of the above work could have been done without the generous external support we received. The Software Engineering Project and the Distributed Computing Project were both supported financially by the Science and Engineering Research Council (UK), and the Transaction Processing Project was supported by IBM (UK) Laboratories. And we have had indirect assistance from many industrial companies who by attending our specification courses have given us the means and motivation to prepare some of our tutorial material.

But we received more than just financial support from these sources, because of course our work could not have been done in a vacuum. The enthusiasm and constant encouragement of our industrial collaborators and SERC administrators helped us to concentrate on real problems and to find practical solutions. So to them we are doubly grateful.

The conviction that real software *can* be specified, and that ordinary mathematics is the proper tool, has been passed on to all of the authors in turn by Tony Hoare

and Jean-Raymond Abrial, and to them we owe the greatest debt.

It remains to thank our very painstaking (and forgiving) referees, and others who have helped us with their comments: Nigel Haigh, Jeff Sanders, and Jim Woodcock. And finally we must thank Martin Raskovsky for his marvellous program which made it possible to produce this book.

The authors
February, 1986

Preface to the second edition

In the six years since the first edition of *Specification Case Studies*, there has been a growing interest in mathematically-based specification techniques such as Z. In these years the Z notation has evolved, and many new books on Z have appeared. Of special note is the work of Mike Spivey in producing a semantics for Z [54, 55] and a reference manual for Z [56]. This reference manual has now become a de-facto standard for Z.

Inevitably, as Z has evolved some notational differences have been introduced. So the primary objective of this second edition is make its use of notation consistent with more recent books on Z. To that end it has been completely revised so that it is now consistent (with some exceptions noted in the text) with the second edition of the Z reference manual [57]. In addition, this book has been checked by the second edition of Mike Spivey's *fuzz* type checking program [53].

Use of this book The first edition of *Specification Case Studies* was envisaged as a way of collecting together a body of research work on mathematically-based specification. It has also been used as reference material for courses on specification.

Fortunately, there is a growing number of more introductory books on Z such as [50, 65], not to mention the Z reference manual itself. These books present a detailed introduction to the concepts and notations of Z. The present volume is complementary to these introductory books, because it provides a collection of more substantial case studies of using Z to specify realistic systems. Thus it can be used as a companion to an introductory text for a course on specification. Alternatively, as with the first edition, these more realistic examples may be studied directly by the researcher or more mathematically-literate practitioner to gain an understanding of the capabilities of the techniques in practice.

Changes from the first edition In revising this volume, as well as updating the notation to be consistent with the Z reference manual, we have also taken the opportunity to improve the exposition and, of course, correct errors in the first edition. Additionally, some more major changes have been made. We outline these here.

The CAVIAR specification in Chapter 5 has been revised to make use of an experimental notation which adds modularisation facilities to Z. At the time the first edition was produced, we were aware that the CAVIAR specification, with its reuse of a number of specification structures, could benefit from making this structure explicit in the specification document itself. However, it was not clear then how to do this. The modularisation features presented in Chapter 5 have been developed with the aims both to keep the extensions to Z as simple as possible and to retain the general flavour of Z. The modules added to Z by this extension can be viewed as super-schemas. The newcomer to Z should be aware that these modularisation facil-

ities are not a part of ‘standard’ Z, but we hope that they will make a contribution to the development of a modularisation facility for Z.

Many people have asked about the progress of the use of Z by IBM Hursley. To this end, two new Chapters 14 and 15 have been added to the book. Chapter 14 provides an update on the use of Z in restructuring part of IBM’s Customer Information Control System (CICS). For those interested in the experience of using such methods in practice, this chapter gives an update on one of the largest and longest-running industrial projects making use of Z.

Chapter 15 gives an overview of a more recent project undertaken at IBM Hursley to specify the CICS Application Programming Interface. This work can be viewed as an extension of the work presented in Chapter 13. However, it is on a larger scale. Chapter 15 reports on the progress made on some of the problems outlined in Chapter 13.

Reading the book The book is divided into four parts, each part beginning with an overview of its contents. Part I provides some tutorials introducing the basic concepts used throughout the book. For those without any background in such methods Part I is the obvious starting point.

The remaining parts of the book are the collections of case studies. Those in Part II are all independent studies. Part III contains work done as part of the Distributed Computing Project at the Programming Research Group and Part IV as part of work done in collaboration with IBM (UK) Laboratories, Hursley, related to CICS, a transaction processing system. The choice of which of these chapters is to be read first depends on the background of the reader. For example, readers familiar with the Unix filing system would be well advised to begin with Chapter 4.

An extensive glossary has been provided as two appendices at the end of the book. The glossaries are not intended to be read from beginning to end, but rather to act as a reference on the meaning of operators, etc. Appendix A contains the mathematical notation of Z and Appendix B the Z schema notation. The appendices are organised into sections covering related aspects. For example, when reading a case study that makes extensive use of sequences, it may be worthwhile to browse the Z sequence notation in Section A.10 to familiarise yourself with it.

For the second edition, the individual bibliographies of each chapter of the first edition have been combined into a single extended bibliography. In addition, a combined index has been provided. This indexes important terms used in the book as well as all definitions occurring as part of the case study specifications.

Related work This volume has been largely concerned with specification. For work on refinement of specifications to programs, a book by Carroll Morgan [45] treats the refinement calculus and a paper by Steve King [37] discusses the relationship between Z and the refinement calculus.

Z is closely related to the Vienna Development Method (VDM). Good starting points for accessing the literature on VDM are the introductory text by Cliff Jones [34] and the collection of case studies [35].

Workshops on Z have been held annually in the UK [5, 6, 48, 49] and workshops on VDM have been held regularly [3, 13]. The latter has now widened its title to Formal Methods Europe (FME). The network news group `comp.specification` discusses specification in general and the subgroup `comp.specification.z` discusses issues related to Z.

Acknowledgements We would like to thank Jonathan Bowen, Carroll Morgan and Bernard Sufrin, for their help with the conversion of the text of the first edition to \LaTeX . We would also like to acknowledge Mike Spivey for his collaboration in our attempt to make both the second edition of the reference manual and the case studies consistent, and for his help with the intricacies of his \LaTeX style used to produce this book. We would also like to thank Brendan Mahony for his help in producing the special fonts needed for the CAVIAR paper. Finally, the referees are to be thanked for their helpful and detailed comments on the second edition.

I.J.H.
June, 1992

Part I

TUTORIALS

Part I contains three chapters that provide tutorial introductions to the techniques used in the remainder of the book. Chapter 1 introduces the notion of using mathematics for specification through examples of a symbol table, file updating and sorting. Chapter 2 introduces the schema notation by way of a specification of a block-structured symbol table; it makes use of the simple symbol table specified in Chapter 1. Chapter 3 provides a slightly more advanced example, in which both the descriptive power of mathematics and the notational compactness of schemas are exploited.

Chapter 1

Small examples of specification using mathematics

Ian Hayes

1.1 Introduction

This chapter is intended for people experienced in programming but not necessarily in specification. It introduces specification using mathematics with the aid of a few simple examples:

- a symbol table with operations to look up, update, and delete symbols;
- a file update; and
- sorting a sequence.

The chapter does not address the area of specifying large systems nor does it use that part of Z designed to deal with building larger specifications. These aspects are dealt with in later chapters.

Since readers are assumed to have some background in logic and basic set operations, these are treated sparingly. Readers without such background are advised to refer to an introductory book that covers these areas, such as [65]. Definitions of all the mathematical operators are given in Appendix A.

1.2 A symbol table

The first example specifies a simple symbol table. The specification demonstrates the use of a mathematical function to specify a data type, and treats operations to update, lookup, and delete entries in the symbol table. *SYM* is used to stand for the set of symbols to be stored in the table, and *VAL* for the set of associated values.

As we assume no further structure on these types here, we introduce them as basic types.

$$[SYM, VAL]$$

A symbol table is modelled by a partial function from symbols, SYM , to values, VAL

$$st : SYM \rightarrowtail VAL$$

The arrow ' \rightarrowtail ' indicates a function from SYM to VAL that is not necessarily defined for all elements of SYM (hence 'partial'). The subset of SYM for which it is defined is its domain of definition

$$\text{dom}(st)$$

If a symbol s is in the domain of definition of st , that is, $s \in \text{dom}(st)$, then $st(s)$ is the unique value associated with s and hence $st(s) \in VAL$. The notation $\{s \mapsto v\}$ describes a function that is only defined for s

$$\text{dom}(\{s \mapsto v\}) = \{s\}$$

and which maps s to v

$$\{s \mapsto v\}(s) = v$$

More generally we can use the notation

$$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \dots, x_n \mapsto y_n\}$$

where all the x_k 's are distinct, to define a function whose domain is

$$\{x_1, x_2, \dots, x_n\}$$

and whose value for each x_k is the corresponding y_k . For example, if we let our symbols be names and values be ages, the following mapping:

$$st = \{\text{'John'} \mapsto 23, \text{'Mary'} \mapsto 19\}$$

maps 'John' onto 23 and 'Mary' onto 19. The domain of st is the set

$$\text{dom}(st) = \{\text{'John'}, \text{'Mary'}\}$$

and

$$\begin{aligned} st(\text{'John'}) &= 23 \\ st(\text{'Mary'}) &= 19 \end{aligned}$$

The range of st , $\text{ran}(st)$, is the set of values that are associated with the symbols in the table. For the example above

$$\text{ran}(st) = \{19, 23\}$$

The notation $\{\}$ is used to denote the empty function whose domain of definition is the empty set. Initially the symbol table is empty:

$$st = \{\}$$

The use of a function as a model for a symbol table is quite different to the normal use of functions in computing, where an algorithm is given to compute the value of the function for a given argument. Here instead we use it to describe a data structure.

There are many possible models that could be used to describe a symbol table. For example, we could use a list of pairs of symbol and value, or a binary tree containing a symbol and value in each node. These other models are not as abstract, because many different lists (or trees) can represent the same function, and it is simpler if the models for two symbol tables are equal whenever they give the same values for the same symbols.

It is possible to distinguish between two unordered list representations which, if regarded as symbol tables, are equal; on the other hand, for the function representation different functions represent different symbol tables. Thus the list and tree models of a symbol table tend to bias an implementor, working from the specification, towards a particular implementation. Of course, both lists and trees could be used to implement such a symbol table, but any reasoning we wish to perform which involves symbol tables is far easier if we use the partial function model rather than either the list or the tree models. In general, the key to a good specification is the choice of the model for the state of the system. It should have enough detail to allow different objects to have different values in the model, but no more.

Because some operations can change the symbol table, we represent the effect of an operation by the relationship between the value of the symbol table before the operation and its value after the operation. We use

$$st, st' : SYM \leftrightarrow VAL$$

where by convention the undecorated symbol table, st , represents the state before the operation and the dashed symbol table, st' , represents the state after the operation. The operation to update an entry in the table can be described mathematically by the following *schema*:

$ \begin{array}{l} \textit{Update} \\ st, st' : SYM \leftrightarrow VAL \\ s? : SYM \\ v? : VAL \end{array} $	$st' = st \oplus \{s? \mapsto v?\}$
--	-------------------------------------

A schema consists of two parts: the declarations (above the centre line) in which variables to be used in the schema are declared, and a predicate (below the centre line) containing properties of, and relating those variables. In the schema *Update* the second line declares a variable with name ' $s?$ ' which is the symbol to be updated. The third line declares a variable with name ' $v?$ ' which is the value to be associated with $s?$ in the symbol table. By convention names in the declarations ending in '?' are inputs and names ending in '!' are outputs; the '?' and '!' are otherwise just part of the name.

The predicate part of the schema states that it updates the symbol table (st) to give a new symbol table (st') in which the symbol $s?$ is associated with the value $v?$. Any previous value associated with $s?$ in the symbol table (if there was one) is lost.

The operator ' \oplus ' (function overriding) combines two functions of the same type to give a new function. The new function $f \oplus g$ is defined for an argument x if either

f or g are defined for x

$$\text{dom}(f \oplus g) = \text{dom}(f) \cup \text{dom}(g)$$

and has the value $g(x)$ if g is defined for x

$$x \in \text{dom}(g) \Rightarrow (f \oplus g)(x) = g(x)$$

otherwise it has the value $f(x)$

$$x \notin \text{dom}(g) \wedge x \in \text{dom}(f) \Rightarrow (f \oplus g)(x) = f(x)$$

For example,

$$\begin{aligned} & \{\text{'Mary'} \mapsto 19, \text{'John'} \mapsto 23\} \oplus \{\text{'John'} \mapsto 25, \text{'George'} \mapsto 62\} \\ &= \{\text{'Mary'} \mapsto 19, \text{'John'} \mapsto 25, \text{'George'} \mapsto 62\} \end{aligned}$$

For the operation *Update* the value of $st'(x)$ is $v?$ if $x = s?$, otherwise it is $st(x)$ provided x is in the domain of st . In this example we are only using ' \oplus ' to override one value in the symbol table function; the operator ' \oplus ' is, however, more general: both its arguments may be any functions of the same type.

The following schema describes the operation to look up an identifier in the symbol table:

<i>LookUp</i>	
$st, st' : SYM \leftrightarrow VAL$	
$s? : SYM$	
$v! : VAL$	
<hr/>	
$s? \in \text{dom}(st) \wedge$	
$v! = st(s?) \wedge$	
$st' = st$	

The second line of the declarations introduces an input with name ' $s?$ ' which is the symbol to be looked up. The third line of the declarations introduces an output with name ' $v!$ ' which is the value that is associated with $s?$ in the symbol table.

The first line of the predicate states that the identifier being looked up should be in the symbol table before the operation is performed; the above schema does not define the effect of looking up an identifier which is not in the table. The second line states that the output value is the value associated with $s?$ in the symbol table st . The final line states that the contents of the symbol table is not changed by a *LookUp* operation.

The operation to delete an entry in the symbol table is given by the following schema:

<i>Delete</i>	
$st, st' : SYM \leftrightarrow VAL$	
$s? : SYM$	
<hr/>	
$s? \in \text{dom}(st) \wedge$	
$st' = \{s?\} \triangleleft st$	

To delete the entry for $s?$ from the symbol table it must be in the table to start with:

$$s? \in \text{dom}(st)$$

The resultant symbol table st' is the symbol table st with $s?$ deleted from its domain. We use the domain exclusion operator ' \triangleleft '. Given a set s and a function f , the domain of $s \triangleleft f$ is the domain of f minus the set s

$$\text{dom}(s \triangleleft f) = \text{dom}(f) \setminus s$$

where ' \setminus ' stands for set subtraction. The values that are left in the resultant function are unchanged

$$x \in \text{dom}(s \triangleleft f) \Rightarrow (s \triangleleft f)(x) = f(x)$$

For example,

$$\begin{aligned} \{\text{'Mary'}, \text{'John'}\} \triangleleft \{\text{'Mary'} \mapsto 19, \text{'John'} \mapsto 25, \text{'George'} \mapsto 62\} \\ = \{\text{'George'} \mapsto 62\} \end{aligned}$$

Exercise 1.1 In place of a single *Update* operation define two separate operations, *Add*, to add a symbol and value if the symbol is not already in the table, and *Replace*, to replace the value associated with a symbol already in the table.

1.3 File update

The second example is a specification of a file update. It uses sets and functions to model a file update operation. Each record in the file is indexed by a key. *Key* stands for the set of all possible keys and *Record* the set of record values.

$$[Key, Record]$$

We model the file as a partial function from keys to records

$$f : Key \rightarrow Record$$

A transaction may either delete an existing record or provide a new record which either replaces an existing record or is added to the file. The transactions for an update of a file are specified as a set of keys $d?$ which are to be deleted from the file, and a partial function $u?$ giving the keys to be updated and their corresponding new records. We add the further restriction that we cannot both delete a record with a given key and provide a new record for that key. For example, if

$$\begin{aligned} f &= \{k_1 \mapsto r_1, k_2 \mapsto r_2, k_3 \mapsto r_3, k_4 \mapsto r_4\} \\ d? &= \{k_2, k_4\} \\ u? &= \{k_3 \mapsto r_5, k_5 \mapsto r_6\} \end{aligned}$$

then the resultant file f' is

$$f' = \{k_1 \mapsto r_1, k_3 \mapsto r_5, k_5 \mapsto r_6\}$$

The specification of a file update operation is given by the following schema:

<i>FileUpdate</i>	
$f, f' : Key \leftrightarrow Record$	
$d? : \mathbb{P} Key$	
$u? : Key \leftrightarrow Record$	
$d? \subseteq \text{dom}(f) \wedge$	
$d? \cap \text{dom}(u?) = \{\}$	
$f' = (d? \triangleleft f) \oplus u?$	

The original file f and the updated file f' are modelled by partial functions from keys to records. The keys to be deleted, $d?$, are a subset of Key . Hence $d?$ is an element of the powerset of Key (the set of all subsets of Key); the notation $\mathbb{P} Key$ is used to denote the powerset of Key . The updates $u?$ are specified as a partial function from Key to $Record$.

Only records already in the file f may be deleted. Hence the set of keys to be deleted $d?$ must be a subset of the domain of the original file ($d? \subseteq \text{dom}(f)$). We are precluded from trying both to delete a key and add a new record for the same key because the intersection of the deletions with the domain of the updates must be empty ($d? \cap \text{dom}(u?) = \{\}$). The resultant file f' is the original file f with all records corresponding to keys in $d?$ deleted ($d? \triangleleft f$), overridden by the new records $u?$.

The last line of *FileUpdate* could have equivalently been written

$$f' = d? \triangleleft (f \oplus u?)$$

Although it is not always the case that these two lines are equivalent, the extra condition that the intersection of $d?$ and $\text{dom}(u?)$ is empty ensures their equivalence in this case. The following lemma formalises this property.

Lemma Given $d? \cap \text{dom}(u?) = \{\}$ the following identity holds:

$$d? \triangleleft (f \oplus u?) = (d? \triangleleft f) \oplus u?$$

Proof First, we show that the domains of the two sides are equal.

$$\begin{aligned}
& \text{dom}(d? \triangleleft (f \oplus u?)) \\
&= \text{dom}(f \oplus u?) \setminus d? && \text{defn } \triangleleft \\
&= (\text{dom}(f) \cup \text{dom}(u?)) \setminus d? && \text{defn } \oplus \\
&= (\text{dom}(f) \setminus d?) \cup (\text{dom}(u?) \setminus d?) && \text{distribution} \\
&= (\text{dom}(f) \setminus d?) \cup \text{dom}(u?) && \text{as } d? \cap \text{dom}(u?) = \{\} \\
&= \text{dom}(d? \triangleleft f) \cup \text{dom}(u?) && \text{defn } \triangleleft \\
&= \text{dom}((d? \triangleleft f) \oplus u?) && \text{defn } \oplus
\end{aligned}$$

Second, for any key k in the domain, we show the two sides are equal. We prove this for the two cases: $k \in \text{dom}(u?)$ and $k \notin \text{dom}(u?)$.

(a) If $k \in \text{dom}(u?)$ then

$$\begin{aligned}
& k \notin d? && \text{as } \text{dom}(u?) \cap d? = \{\} \\
& (d? \triangleleft (f \oplus u?))(k) = (f \oplus u?)(k) && \text{as } k \notin d? \\
& \quad = u?(k) && \text{as } k \in \text{dom}(u?) \\
& ((d? \triangleleft f) \oplus u?)(k) = u?(k) && \text{as } k \in \text{dom}(u?)
\end{aligned}$$

(b) If $k \notin \text{dom}(u?)$ then

$$\begin{aligned}
 (d? \triangleleft (f \oplus u?))(k) &= (f \oplus u?)(k) && \text{as } k \in \text{dom}(d? \triangleleft (f \oplus u?)) \\
 &= f(k) && \text{as } k \notin \text{dom}(u?) \\
 ((d? \triangleleft f) \oplus u?)(k) &= (d? \triangleleft f)(k) && \text{as } k \notin \text{dom}(u?) \\
 &= f(k) && \text{as } k \in \text{dom}(d? \triangleleft (f \oplus u?))
 \end{aligned}$$

□

In the specification of *FileUpdate*, if we were not given the extra restriction then, as specified in the last line, the updating of records would have precedence over deletions. If the alternative specification were used then deletions would have precedence over updates. It is sensible to include the extra restriction in the specification as it allows the most freedom in implementation without any real loss of generality.

Exercise 1.2 Define an operation, *FileAdd*, to add a number of keys with associated records to a file. The keys should not already be contained in the file.

1.4 Sorting

The third example specifies sorting a sequence into non-decreasing order; it uses the mathematical theories of *sequences* and *bags* (also known as multi-sets). See glossary sections A.10 and A.11 for more details.

The input and the output of *Sort* are sequences of elements of a given type X , which has a total order ' \prec ' defined on it. A sequence is modelled as a partial function from the positive natural numbers, $\mathbb{N}_1 = \{1, 2, 3, \dots\}$, to the base type X as follows:

$$\text{seq } X == \{s : \mathbb{N}_1 \rightarrow X \mid (\exists n : \mathbb{N} \bullet \text{dom}(s) = 1 \dots n)\}$$

In \mathbf{Z} , sequences are finite, so there must exist a natural number n which is the length of the sequence. The domain of the function representing a sequence is the complete set of natural numbers ranging from 1 to n . A sequence may be empty, in which case it has length zero and the domain of the function modelling it is the empty set.

The notation of enclosing a list of elements in angle brackets is used to construct a sequence consisting of the list of elements. For example,

$$t = \langle a, b, c \rangle = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$$

The empty sequence is denoted by $\langle \rangle$.

We can select an element in a sequence by indexing the sequence with the position of the element. For example,

$$\begin{aligned}
 t(2) &= b \\
 s &= \langle s(1), s(2), \dots, s(\#s) \rangle
 \end{aligned}$$

where $\#s$ is the length of the sequence s (which is also the number of members of s viewed as its representation as a set of pairs).

The output of *Sort*, *out!*, must be in non-decreasing order. We can specify this by stating that *out!* is a member of the set of all non-decreasing sequences with elements

from the set X .

$$\begin{aligned} \text{NonDecreasing} = & \\ \{s : \text{seq } X \mid \forall i, j : \text{dom } s \bullet (i < j) \Rightarrow \neg (s(j) \prec s(i))\} \end{aligned}$$

For all pairs of indices, i and j , in the domain of a nondecreasing sequence s , if i is less than j , then $s(j)$ is not less than $s(i)$.

The output of *Sort* must contain the same values as the input, with the same frequency. We can state this property using bags. A bag is similar to a set except that multiple occurrences of an item in a bag are significant. We can model a bag of items of base type X – i.e. the items have type X – as a partial function from the items of type X to the frequency of occurrence of the items in the bag. We use the notation $\llbracket \dots \rrbracket$ to construct a bag. For example, the bag

$$\llbracket a, b, b, b \rrbracket = \{a \mapsto 1, b \mapsto 3\}$$

contains one copy of a and three copies of b . The following gives some examples of how sets, bags and sequences are related:

$$\begin{aligned} \{1, 2, 2\} &= \{1, 2\} = \{2, 1\} \\ \llbracket 1, 2, 2 \rrbracket &\neq \llbracket 1, 2 \rrbracket = \llbracket 2, 1 \rrbracket \\ \langle 1, 2, 2 \rangle &\neq \langle 1, 2 \rangle \neq \langle 2, 1 \rangle \end{aligned}$$

In specifying *Sort* we would like to say that the bag formed from all the elements in the output sequence is the same as the bag of elements in the input sequence, because it is only the order that changes. We introduce the function *items* which forms the bag of all the elements in a sequence. For example,

$$\begin{aligned} \text{items}(\langle \rangle) &= \llbracket \rrbracket = \{\} \\ \text{items}(\langle 1 \rangle) &= \llbracket 1 \rrbracket = \{1 \mapsto 1\} \\ \text{items}(\langle 1, 2, 2 \rangle) &= \text{items}(\langle 2, 1, 2 \rangle) = \llbracket 1, 2, 2 \rrbracket = \{1 \mapsto 1, 2 \mapsto 2\} \\ \text{items}(\langle 1, 2, 3 \rangle) &= \text{items}(\langle 2, 1, 3 \rangle) = \llbracket 1, 2, 3 \rrbracket = \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\} \end{aligned}$$

More precisely, *items* can be defined as follows:

$\begin{aligned} &\text{items} : \text{seq } X \rightarrow \text{bag } X \\ &\forall x : X; s, t : \text{seq } X \bullet \\ &\quad \text{items} \langle \rangle = \llbracket \rrbracket \wedge \\ &\quad \text{items} \langle x \rangle = \llbracket x \rrbracket \wedge \\ &\quad \text{items}(s \frown t) = \text{items } s \uplus \text{items } t \end{aligned}$

Each element x that occurs in the sequence, $x \in \text{ran}(s)$, is mapped onto its frequency of occurrence in the sequence (i.e. the size of the set of positions in the sequence that have value x). (The above definition of *items* is more restricted than the definition given in the glossary; the latter is generalised to operate on any relation.) For example, if $s = \langle 3, 4, 3, 6 \rangle$ then

$$\text{ran}(s) = \{3, 4, 6\}$$

and

$$\begin{aligned} \{i : \text{dom}(s) \mid s(i) = 3\} &= \{1, 3\} \\ \{i : \text{dom}(s) \mid s(i) = 4\} &= \{2\} \\ \{i : \text{dom}(s) \mid s(i) = 6\} &= \{4\} \end{aligned}$$

Hence

$$\begin{aligned}
 \text{items}(s) &= \{3 \mapsto \#\{1, 3\}, 4 \mapsto \#\{2\}, 6 \mapsto \#\{4\}\} \\
 &= \{3 \mapsto 2, 4 \mapsto 1, 6 \mapsto 1\} \\
 &= \llbracket 3, 3, 4, 6 \rrbracket
 \end{aligned}$$

We can now define that the output of the sort must have the same values, with the same frequencies, as the input:

$$\text{items}(\text{out}!) = \text{items}(\text{in}?)$$

The complete specification of sorting is given by the combination of the properties that the output is both ordered and has the same content as the input.

<i>Sort</i>
$\text{in}?, \text{out}! : \text{seq } X$
$\text{out}! \in \text{NonDecreasing} \wedge$ $\text{items}(\text{out}!) = \text{items}(\text{in}?)$

The output of the sort is non-decreasing. The output sequence must contain the same elements as the input, with the same frequency.

Sort is an example of a non-algorithmic specification: *it specifies what Sort should achieve but not how to go about achieving it*. The advantage of a non-algorithmic specification is that its meaning may be more obvious than one which contains the extra detail necessary for it to be algorithmic. The specification is given in terms of the (defining) properties of the problem without biasing the implementor towards a particular form of algorithm. There are many possible sorting algorithms. For example, both bubble sort and quick sort may be used to meet the above specification. These algorithms have quite different efficiencies. The implementor should be allowed the freedom to choose the most appropriate. Furthermore, any particular sorting algorithm is more constraining than the above specification.

Exercise 1.3 Rewrite the sort specification to sort a sequence with no duplicates into strictly ascending order.

Exercise 1.4 Write a specification of a merge of two non-decreasing sequences to produce a non-decreasing sequence. Duplicates from the two inputs should all appear in the output sequence. Hint: the sum of two bags b and c is denoted $b \uplus c$; the frequency of occurrence of an item in $b \uplus c$ is the sum of its frequencies in b and c .

1.5 Solutions to exercises

These solutions are samples; there is more than one correct solution to each exercise.

Solution 1.1

<i>Add</i>
$st, st' : SYM \leftrightarrow VAL$ $s? : SYM$ $v? : VAL$
$s? \notin \text{dom}(st) \wedge$ $st' = st \cup \{s? \mapsto v?\}$

<i>Replace</i>
$st, st' : SYM \leftrightarrow VAL$ $s? : SYM$ $v? : VAL$
$s? \in \text{dom}(st) \wedge$ $st' = st \oplus \{s? \mapsto v?\}$

Solution 1.2

<i>FileAdd</i>
$f, f' : Key \leftrightarrow Record$ $a? : Key \leftrightarrow Record$
$\text{dom}(a?) \cap \text{dom}(f) = \{\} \wedge$ $f' = f \cup a?$

Solution 1.3

$NoDuplicates == \{s : \text{seq } X \mid \forall i, j : \text{dom } s \bullet i \neq j \Rightarrow s(i) \neq s(j)\}$
 $Ascending == \{s : \text{seq } X \mid \forall i, j : \text{dom } s \bullet i < j \Rightarrow s(i) \prec s(j)\}$

<i>SortNoDup</i>
$in?, out! : \text{seq } X$
$in? \in NoDuplicates \wedge$ $out! \in Ascending \wedge$ $\text{ran}(in?) = \text{ran}(out!)$

The last predicate of *SortNoDup* could also have been written

$$items(in?) = items(out!)$$

Solution 1.4

<i>Merge</i>
$in1?, in2?, out! : \text{seq } X$
$in1? \in NonDecreasing \wedge$ $in2? \in NonDecreasing \wedge$ $out! \in NonDecreasing \wedge$ $items(out!) = items(in1?) \uplus items(in2?)$

Chapter 2

Block-structured symbol table

Ian Hayes

2.1 Introduction

A specification of a symbol table suitable for the sequential processing of block-structured languages is given. This specification is intended to demonstrate how, using Z, a specification can be built from components.

A simple symbol table suitable for a single block is described first; it has operations to look up, add, replace and delete entries. This simple symbol table is the same as that given in Chapter 1. The treatment given here emphasises how such a specification can be built using the schema notation of Z [44, 57] and includes a treatment of error conditions not given earlier. Readers not familiar with the mathematics used here should consult Chapter 1 for a more detailed explanation.

Section 2.3 specifies a block-structured symbol table in terms of a sequence of simple symbol tables, one for each nested block. Operations are given to search the *environment* for a symbol, and to start and finish nested blocks; the operations on a simple symbol table are *promoted* to work on the symbol table corresponding to the smallest enclosing block.

2.2 Symbol table

A symbol table associates a unique value (from the set VAL) with a symbol (from the set SYM):

$$[SYM, VAL]$$

The operations allowed on a symbol table are to,

- look up the value associated with a symbol in the table;

- add a symbol with corresponding value, providing the symbol is not already in the table;
- replace the value associated with a symbol already in the table; and
- delete a symbol and its associated value from the table.

*To specify a symbol table, we first give a model of its state and a description of its initial state, then we specify each of the operations in terms of the relationship between the state before an operation, the inputs to the operation, the outputs from the operation, and the state after the operation.*¹

2.2.1 The state

The state of a symbol table can be modelled by a partial function from symbols to values:

$$ST == SYM \leftrightarrow VAL$$

Initially the symbol table is empty:

$$st_0 == \{ \}$$

2.2.2 Operations

Each operation on a symbol table transforms a symbol table before (st) into a symbol table after (st'):

$$\Delta ST \triangleq [st, st' : ST]$$

The definition of each operation must include declarations of the ‘before’ and ‘after’ states of the operation; rather than write out these declarations in full in each definition, we introduce a schema ΔST that contains just these declarations and include this schema in the definition of each operation as an abbreviation for the declarations. The ‘ Δ ’ (for ‘change’) in ‘ ΔST ’ is just part of the name of the schema; we allow Greek letters in names. By convention names beginning with ‘ Δ ’ are used for schemas that contain before and after state components.

We use the horizontal form of schema definition here. The horizontal form consists of an opening square bracket, declarations optionally followed by a vertical bar and a predicate (ΔST does not include the optional predicate but ΞST , below, does), and finishing with a closing square bracket.

Error handling and the operation to look up a symbol do not modify the symbol table:

$$\Xi ST \triangleq [\Delta ST \mid st' = st]$$

The schema ΞST declares the before and after states (in ΔST) and constrains them to be equal; this schema describes the effect on the state of inquiry-like operations (such as looking up a symbol in the symbol table) and error handling; neither of these modifies the state. The ‘ Ξ ’ (for no change) in ‘ ΞST ’ is again just part of the name.

¹Within this chapter paragraphs in italics have been added to explain the specification method, notation, and conventions. They would not normally appear in such a specification.

By convention names beginning with ‘ Ξ ’ are used for schemas which are written to express that there is no change. Expanding the definition of ΞST we get the following schema:

$st, st' : ST$
$st' = st$

To look up the value $v!$ associated with a symbol $s?$ we use the operation *LookUp*.

<i>LookUp</i>
ΞST
$s? : SYM$
$v! : VAL$
$s? \in \text{dom}(st) \wedge$ $v! = st(s?)$

The schema ΞST is used in the definition of *LookUp* to declare the before and after states (st and st') and to constrain them to be equal. The convention of using the ΞST schema saves writing out all the state components and the equality constraint explicitly.

A schema may be included in the declaration part of a schema; the declarations of the included schema are merged with the other declarations and its predicates are conjoined with the predicates of the schema. Expanding the definition of *LookUp* we get the following schema:

$st, st' : ST$
$s? : SYM$
$v! : VAL$
$st' = st \wedge$ $s? \in \text{dom}(st) \wedge$ $v! = st(s?)$

Schema inclusion is used extensively in Z to construct concise specifications.

A new symbol and corresponding value may be added to the symbol table provided the symbol is not already in the table.

<i>Add</i>
ΔST
$s? : SYM$
$v? : VAL$
$s? \notin \text{dom}(st) \wedge$ $st' = st \cup \{s? \mapsto v?\}$

This schema uses ΔST to include the declarations of the before and after states.

The value associated with a symbol already in the table may be replaced using the following operation:

<i>Replace</i>
ΔST
$s? : SYM$
$v? : VAL$
$s? \in \text{dom}(st) \wedge$
$st' = st \oplus \{s? \mapsto v?\}$

An entry in the symbol table may be deleted using the following operation:

<i>Delete</i>
ΔST
$s? : SYM$
$s? \in \text{dom}(st) \wedge$
$st' = \{s?\} \triangleleft st$

2.2.3 Errors

The operations *LookUp*, *Replace*, and *Delete* all have a precondition that the symbol is present in the table: $s? \in \text{dom}(st)$. *Add* has a precondition that the symbol is not already in the table: $s? \notin \text{dom}(st)$.

If the precondition of an operation is not met for a call on the operation, its effect is not defined: the operation may do as it chooses. It may do nothing; it may abort the program perhaps giving an appropriate error message; it may attempt the operation and give incorrect results or scramble the state so that later calls to operations give incorrect results or abort.

If we want operations to give error messages when their preconditions are not met, then we need to specify error alternatives to cover these cases. If the alternatives cover all such cases – the precondition of the operation, complete with alternatives, is true – we call such a specification robust.

To specify error alternatives for the symbol table operations, we introduce a status report that is returned by all operations. The status reports needed are defined by the following alternatives:

<i>Report</i> ::= <i>OK</i>
<i>Symbol_not_present</i>
<i>Symbol_present</i>
<i>Not_within_any_block</i>
<i>Symbol_not_found</i>

For the *LookUp*, *Replace*, and *Delete* operations, if the symbol is not present an error is reported and the symbol table is not modified.

NotPresent ΞST $s? : SYM$ $rep! : Report$
$s? \notin \text{dom}(st) \wedge$ $rep! = \text{Symbol_not_present}$

The schema ΞST is included in the above schema to introduce the declarations of the before and after states and to constrain them to be equal.

An *Add* operation can fail if the symbol is already present in the table.

Present ΞST $s? : SYM$ $rep! : Report$
$s? \in \text{dom}(st) \wedge$ $rep! = \text{Symbol_present}$

Successful operations return a report of *OK*:

$$\text{Success} \triangleq [rep! : Report \mid rep! = OK]$$

The operations with error handling follow:

$$\begin{aligned} STLookUp &\triangleq (LookUp \wedge \text{Success}) \vee \text{NotPresent} \\ STAdd &\triangleq (Add \wedge \text{Success}) \vee \text{Present} \\ STReplace &\triangleq (Replace \wedge \text{Success}) \vee \text{NotPresent} \\ STDelete &\triangleq (Delete \wedge \text{Success}) \vee \text{NotPresent} \end{aligned}$$

Either a *LookUp* operation is successfully performed (if $s? \in \text{dom}(st)$), in which case a report of *OK* is given, or the *LookUp* operation reports that the symbol is not present.

The conjunction ‘ \wedge ’ of two schemas is formed by merging their declarations (variables common to both declarations must have the same type) and conjoining their predicates. Below we expand the *STLookUp* operation. We do not normally find it necessary to expand such definitions to understand the specification but the expansions are intended to help those who are not familiar with the notation. The expansion of

$$LookUp \wedge \text{Success}$$

is

ΞST $s? : SYM$ $v! : VAL$ $rep! : Report$
$s? \in \text{dom}(st) \wedge$ $v! = st(s?) \wedge$ $rep! = OK$

In this case there are no variables common to the declarations of the schemas *LookUp* and *Success*.

The disjunction ‘ \vee ’ of two schemas is formed by merging their declarations (variables common to both must have the same type) and disjoining their predicate parts. The expansion of

$$(LookUp \wedge Success) \vee NotPresent$$

is

ΞST $s? : SYM$ $v! : VAL$ $rep! : Report$
$(s? \in dom(st) \wedge v! = st(s?) \wedge rep! = OK)$ \vee $(s? \notin dom(st) \wedge rep! = Symbol_not_present)$

In this case the declarations in ΞST and the declarations of $s?$ and $rep!$ are common and have the same types, and hence can be merged. Note that no constraint (other than that indicated by its type declaration) is placed on the value of $v!$ returned in the error case.

Exercise 2.1 Give expanded forms of the two schemas *STReplace* and *STDelete*.

2.3 Block-structured symbol table

We now describe a symbol table suitable for use in processing (e.g. compiling) a block-structured language such as Algol 60 and its many descendants. In such languages each variable declaration is associated with a block and a variable may be referred to only from within the block with which it is associated. Blocks may be nested within other blocks to an arbitrary level. Each nested block must be completely enclosed by the block in which it is included. For example, consider the fragment of Algol 60 in Figure 2.1. The outer block *A* declares variables *x* and *y* of type integer. These variables may be referred to anywhere within block *A*, except that the variable *y* of block *A* may not be referred to within block *B* because there is a variable with the same name declared in block *B*: within block *B* the outer (block *A*) declaration of *y* is *hidden* by the declaration of *y* in block *B*. We refer to those parts of the program in which a variable may be referred to as being within the *scope* of that variable.

A symbol table suitable for processing of block-structured languages should support the scoping rules of block-structured languages. It should have operations for starting and finishing blocks as well as operations to access, add, replace and delete entries in the table.

2.3.1 The state

The simple symbol table described in Section 2.2 is suitable only for keeping track of the variables of a single block. At a given point in a program we need to keep

```

begin A
  integer x, y;
  ...
  x := 2; y := 3;                                (1)
  ...
  begin B
    real y; integer z;
    ...
    y := 0.5; x := z;                            (2)
    ...
  end B;
  ...
  y := x;                                         (3)
  ...
end A

```

Figure 2.1: Example block-structured program

track of all the variables declared in all the blocks enclosing that point. This can be done by associating a simple symbol table with each block enclosing the point. To keep track of the order in which the blocks are nested we arrange the symbol tables into a sequence so that, if a block A encloses another block B , the symbol table for A precedes the symbol table for B in the sequence. We model a block-structured symbol table by a sequence of symbol tables

$$BST == \text{seq } ST$$

The first symbol table in the sequence is for the outermost block.

In the example given above, the block-structured symbol table within block A but excluding block B (e.g. at the positions marked (1) and (3)) is a sequence containing a single symbol table

$$\langle \{x \mapsto \text{integer}, y \mapsto \text{integer}\} \rangle$$

where x , y and z are constants representing the corresponding variable names, and *integer* and *real* are constants representing the corresponding types.

Within block B (e.g. at the position marked (2)) the sequence contains two symbol tables

$$\langle \{x \mapsto \text{integer}, y \mapsto \text{integer}\}, \{y \mapsto \text{real}, z \mapsto \text{integer}\} \rangle$$

At any point within a program at most one variable of a given name may be referenced. We refer to the variables that may be referenced at a given point, along with their associated information, as the *environment* of that point. An environment may be represented as a simple symbol table. In the example above, the environment within block A but excluding block B (namely (1) and (3)) is

$$\{x \mapsto \text{integer}, y \mapsto \text{integer}\}$$

and within block B it is equal to the symbol table for block A overridden by the symbol table for block B

$$\begin{aligned} & \{x \mapsto \text{integer}, y \mapsto \text{integer}\} \oplus \{y \mapsto \text{real}, z \mapsto \text{integer}\} \\ & = \{x \mapsto \text{integer}, y \mapsto \text{real}, z \mapsto \text{integer}\} \end{aligned}$$

In general, the environment corresponding to a block-structured symbol table consisting of a sequence of symbol tables is given by overriding the symbol tables in sequence. For example, for the sequence

$$\langle st_1, st_2, \dots, st_n \rangle$$

the environment is

$$st_1 \oplus st_2 \oplus \dots \oplus st_n$$

We can define the distributed override operator ‘ $\oplus/$ ’ which extracts the environment from a sequence of symbol tables by the following:

$$\left| \begin{array}{l} \oplus/ : \text{seq } ST \rightarrow ST \\ \hline \forall s : ST; ss, tt : \text{seq } ST \bullet \\ \oplus/\langle \rangle = \{\} \wedge \\ \oplus/\langle s \rangle = s \wedge \\ \oplus/(ss \frown tt) = (\oplus/ ss) \oplus (\oplus/ tt) \end{array} \right.$$

Initially no blocks have been entered; hence the block-structured symbol table is the empty sequence:

$$bst_0 == \langle \rangle$$

2.3.2 Operations

The operations on a block-structured symbol table transform a state before (bst) to a state after (bst'):

$$\Delta BST \triangleq [bst, bst' : BST]$$

Some operations leave the state unchanged:

$$\Xi BST \triangleq [\Delta BST \mid bst' = bst]$$

There are two operations that retrieve information about a symbol from a block-structured symbol table: $BLookUp$ and $BSearch$. $BLookUp$ looks in the most nested symbol table only; it is defined in terms of $STLookUp$. $BSearch$ searches for a symbol in the environment (i.e. the most nested occurrence of the symbol in the block-structured symbol table). The specification of $BSearch$ makes use of distributed override.

$$\left| \begin{array}{l} BSearch0 _ \\ \hline \Xi BST \\ s? : SYM \\ v! : VAL \\ \hline s? \in \text{dom}(\oplus/ bst) \wedge v! = (\oplus/ bst)(s?) \end{array} \right.$$

We follow the convention of using the operation name `BSearch` with a ‘0’ appended for the specification of the operation without any error handling. The robust operation `BSearch` is specified in Section 2.3.3.

When the start of a block is encountered a new (empty) symbol table is appended to the sequence.

$BStart0$
ΔBST
$bst' = bst \frown \langle st_0 \rangle$

When the end of a block is encountered the last symbol table in the sequence is deleted. This has a precondition that the sequence is non-empty.

$BEnd0$
ΔBST
$bst \neq \langle \rangle \wedge bst' = front(bst)$

We want to be able to perform the simple symbol table operations (*STAdd*, *STReplace*, *STLookUp* and *STDelete*) on the most nested (last) symbol table in the sequence. These operations can only be performed provided the sequence is non-empty, and they change only the last symbol table in the sequence. The relationship between the before and after values of the last symbol table in the sequence is determined by the simple symbol table operations.

A direct definition of the replace operation on the last symbol table in the sequence is given by *BReplace0*. The predicate part of this schema has been written to highlight its relationship to *STReplace*; the conditions at the end of the predicate are exactly those in *STReplace*.

$BReplace0$
ΔBST
$s? : SYM$
$v? : VAL$
$rep! : Report$
ΔST
$bst \neq \langle \rangle \wedge front(bst') = front(bst) \wedge$ $st = last(bst) \wedge st' = last(bst') \wedge$ $((s? \in dom(st) \wedge st' = st \oplus \{s? \mapsto v?\} \wedge rep! = OK)$ \vee $(s? \notin dom(st) \wedge st' = st \wedge rep! = Symbol_not_present))$

The other operations can be written in a similar manner, but here we would like to make use of a technique known as promotion to define the operations on the last symbol table in terms of the operations given earlier on a single symbol table. To do this we pull out the common part of the promoted operations in a framing schema ΦBST .

ΦBST
ΔBST
ΔST
$bst \neq \langle \rangle \wedge$ $front(bst') = front(bst) \wedge$ $st = last(bst) \wedge$ $st' = last(bst')$

The schema ΦBST does not specify the relationship between the last symbol table in the sequence before (st) and after (st') an operation; we have already described these relationships in our definitions of the simple symbol table operations. ΦBST just links a single symbol table to the most nested one in the sequence bst . We can now define the promoted symbol table operations in terms of the definitions of the simple symbol table operations given earlier.

The promoted operations follow:

$$\begin{aligned}
BLookUp0 &\hat{=} STLookUp \wedge \Phi BST \\
BAdd0 &\hat{=} STAdd \wedge \Phi BST \\
BReplace0 &\hat{=} STReplace \wedge \Phi BST \\
BDelete0 &\hat{=} STDelete \wedge \Phi BST
\end{aligned}$$

This definition of $BReplace0$ is equivalent to the one given earlier.

The state components st and st' are used in the above definition as the link between the simple symbol table operation schema and the framing schema. The state of the operation is really just bst and bst' . The st and st' components can be hidden.

A schema may have some of its components hidden by using existential quantification. The declarations of the hidden variables are removed from the declaration part of the schema and are existentially quantified in the predicate part. If the list of components being quantified is replaced by a schema then all the variables in the declaration part of the schema are hidden. As we wish to hide st and st' , we quantify with ΔST .

$$\begin{aligned}
BLookUp1 &\hat{=} (\exists \Delta ST \bullet BLookUp0) \\
BAdd1 &\hat{=} (\exists \Delta ST \bullet BAdd0) \\
BReplace1 &\hat{=} (\exists \Delta ST \bullet BReplace0) \\
BDelete1 &\hat{=} (\exists \Delta ST \bullet BDelete0)
\end{aligned}$$

The components of ΔST (st and st') are hidden in the above definitions because we wish to define the operations as working on before and after states which are of type BST ; the ΔST components are only used to make the link between the specifications of the operations on the simple symbol table and the part of the BST state that the simple operations are to be performed on. The expanded form of $BReplace1$ is the following:

<i>BReplace1</i>
ΔBST
$s? : SYM$
$v? : VAL$
$rep! : Report$
$\begin{aligned} &\exists \Delta ST \bullet bst \neq \langle \rangle \wedge front(bst') = front(bst) \wedge \\ &\quad st = last(bst) \wedge st' = last(bst') \wedge \\ &\quad ((s? \in dom(st) \wedge st' = st \oplus \{s? \mapsto v?\} \wedge rep! = OK) \\ &\quad \vee \\ &\quad (s? \notin dom(st) \wedge st' = st \wedge rep! = Symbol_not_present)) \end{aligned}$

This may be simplified by using the single point rule to eliminate the existential quantifier: as $st = last(bst)$ and $st' = last(bst')$ we can replace all occurrences of st and st' by $last(bst)$ and $last(bst')$, respectively.

ΔBST
$s? : SYM$
$v? : VAL$
$rep! : Report$
$\begin{aligned} &bst \neq \langle \rangle \wedge front(bst') = front(bst) \wedge \\ &((s? \in dom(last(bst)) \wedge last(bst') = last(bst) \oplus \{s? \mapsto v?\} \wedge \\ &\quad rep! = OK) \\ &\vee \\ &(s? \notin dom(last(bst)) \wedge last(bst') = last(bst) \wedge \\ &\quad rep! = Symbol_not_present)) \end{aligned}$

2.3.3 Errors

The upgraded operations and *BEnd* fail if the sequence is empty.

<i>Empty</i>
ΞBST
$rep! : Report$
$bst = \langle \rangle \wedge rep! = Not_within_any_block$

The *BSearch* operation fails if the symbol is not in the environment. If the sequence is empty we give preference to the *Empty* error. Hence, for this error, we require that the sequence is non-empty.

<i>NotFound</i>
ΞBST
$s? : SYM$
$rep! : Report$
$bst \neq \langle \rangle \wedge s? \notin dom(\oplus / bst) \wedge rep! = Symbol_not_found$

The final definitions of the operations follow:

$$\begin{aligned}
BSearch &\hat{=} (BSearch0 \wedge Success) \vee NotFound \vee Empty \\
BStart &\hat{=} BStart0 \wedge Success \\
BEnd &\hat{=} (BEnd0 \wedge Success) \vee Empty \\
BLookUp &\hat{=} BLookUp1 \vee Empty \\
BAdd &\hat{=} BAdd1 \vee Empty \\
BReplace &\hat{=} BReplace1 \vee Empty \\
BDelete &\hat{=} BDelete1 \vee Empty
\end{aligned}$$

An expanded and simplified definition of $BSearch$ follows:

$ \begin{aligned} &\exists BST \\ &s? : SYM \\ &v! : VAL \\ &rep! : Report \end{aligned} $
$ \begin{aligned} &(s? \in dom(\oplus / bst) \wedge v! = (\oplus / bst)(s?) \wedge rep! = OK) \\ &\vee \\ &(bst \neq \langle \rangle \wedge s? \notin dom(\oplus / bst) \wedge rep! = Symbol_not_found) \\ &\vee \\ &(bst = \langle \rangle \wedge rep! = Not_within_any_block) \end{aligned} $

Here is a different expansion and simplification of $BSearch$. It is logically equivalent to the one above, but achieves a different emphasis:

$ \begin{aligned} &\exists BST \\ &s? : SYM \\ &v! : VAL \\ &rep! : Report \end{aligned} $
$ \begin{aligned} &(bst = \langle \rangle \Rightarrow rep! = Not_within_any_block) \\ &\wedge \\ &(bst \neq \langle \rangle \Rightarrow \\ &\quad (s? \in dom(\oplus / bst) \Rightarrow v! = (\oplus / bst)(s?) \wedge rep! = OK) \\ &\quad \wedge \\ &\quad (s? \notin dom(\oplus / bst) \Rightarrow rep! = Symbol_not_found)) \end{aligned} $

Exercise 2.2 Give an expansion of $BDelete$.

Exercise 2.3 Define a search operation $BLocate$ that returns not only the value associated with a symbol but also the level of the innermost block in which it is declared.

2.4 Other approaches

The specification given above is just one approach to the specification of a block-structured symbol table. It is tailored to sequential processing of a block-structured language. An alternative approach that would allow for non-sequential processing is to allocate a block identifier from the set

$$[BlockId]$$

to each block. The state then becomes a mapping from block identifiers to simple symbol tables, along with a relation defining which blocks are contained in which.

$\begin{array}{l} \text{BST1} \\ \hline bst : BlockId \leftrightarrow ST \\ contained : BlockId \leftrightarrow BlockId \\ \hline contained^* \in \text{partial_order}[BlockId] \end{array}$

A quite different specification can be developed starting from this state.

Another alternative is to identify a block by the sequence of indices of blocks enclosing it. As there can be more than one block directly nested within a given enclosing block, we need to distinguish between the blocks by an index giving the position of the block within its enclosing block. Any block is uniquely identified by a sequence of such indices, which has length equal to the depth of nesting of the block:

$$PATH == \text{seq } \mathbb{N}_1$$

The block-structured symbol table is a mapping from paths to simple symbol tables. As the paths represent the nesting structure, any prefix of a path indexing a block must correspond to a block in the table.

$\begin{array}{l} \text{BST2} \\ \hline bst : PATH \leftrightarrow ST \\ \hline \forall p : \text{dom } bst \bullet \\ \quad \forall pr : PATH \bullet pr \subseteq p \Rightarrow pr \in \text{dom } bst \end{array}$

These alternative specifications of the state of a block-structured symbol table are just different views of the same abstract entity. None is any more valid than the others. However, a particular view may be more suitable for specifying particular properties of the system.

2.5 Solutions to exercises

Solution 2.1

<i>STReplace</i>
$st, st' : SYM \leftrightarrow VAL$ $s? : SYM$ $v? : VAL$ $rep! : Report$
$(s? \in \text{dom}(st) \wedge st' = st \oplus \{s? \mapsto v?\} \wedge rep! = OK)$ $\vee (s? \notin \text{dom}(st) \wedge st' = st \wedge rep! = \text{Symbol_not_present})$

<i>STDelete</i>
$st, st' : SYM \leftrightarrow VAL$ $s? : SYM$ $rep! : Report$
$(s? \in \text{dom}(st) \wedge st' = \{s?\} \triangleleft st \wedge rep! = OK)$ $\vee (s? \notin \text{dom}(st) \wedge st' = st \wedge rep! = \text{Symbol_not_present})$

Solution 2.2

<i>BDelete</i>
$bst, bst' : \text{seq } ST$ $s? : SYM$ $rep! : Report$
$(\exists st, st' : ST \bullet$ $bst \neq \langle \rangle \wedge$ $front(bst') = front(bst) \wedge$ $st = last(bst) \wedge$ $st' = last(bst') \wedge$ $((s? \in \text{dom}(st) \wedge st' = \{s?\} \triangleleft st \wedge rep! = OK)$ \vee $(s? \notin \text{dom}(st) \wedge st' = st \wedge$ $rep! = \text{Symbol_not_found})))$ \vee $(bst = \langle \rangle \wedge$ $bst' = bst \wedge$ $rep! = \text{Not_within_any_block})$

This is equivalent to the following:

<i>BDelete</i>
$bst, bst' : \text{seq } ST$ $s? : SYM$ $rep! : Report$
$(bst \neq \langle \rangle \wedge s? \in \text{dom}(\text{last}(bst)) \wedge$ $\text{front}(bst') = \text{front}(bst) \wedge$ $\text{last}(bst') = \{s?\} \triangleleft \text{last}(bst) \wedge$ $rep! = OK)$ \vee $(bst \neq \langle \rangle \wedge s? \notin \text{dom}(\text{last}(bst)) \wedge$ $bst' = bst \wedge$ $rep! = \text{Symbol_not_found})$ \vee $(bst = \langle \rangle \wedge$ $bst' = bst \wedge$ $rep! = \text{Not_within_any_block})$

Solution 2.3

<i>BLocate0</i>
$\exists BST$ $s? : SYM$ $v! : VAL$ $level! : \mathbb{N}$
$bst \neq \langle \rangle \wedge s? \in \text{dom}(\oplus / bst) \wedge$ $level! = \max\{i : \text{dom}(bst) \mid s? \in \text{dom}(bst(i))\} \wedge$ $v! = bst(level!)(s?)$

$$BLocate \hat{=} (BLocate0 \wedge Success) \vee NotFound \vee Empty$$

Chapter 3

Telephone network

Carroll Morgan

Abstract The specification of a simple telephone system is used to illustrate two general features of specifications in Z:

- how the use of *schemas* can drastically reduce the amount of rewriting required when developing specifications; and
- how the direct use of mathematics makes it possible to describe desired properties of an implementation without constraining the implementor's choice of algorithm.

Exercises and solutions to them are included. A similar but more comprehensive specification is given in [42].

3.1 Introduction

We choose as our example a simple *telephone network* in which connections may be established between pairs of telephones. A request may be made for the connection of a given phone to any other; if the request cannot be satisfied immediately, it will be stored by the network and satisfied, if possible, at some later time (for example, the other phone might be engaged). We describe only three network operations:

Call – request a connection between two telephones;

HangUp – terminate a connection; and

Engaged – indicate whether a given telephone is currently engaged, and if so, with which phone.

3.2 The specification

Let the set of telephones be *PHONE*:

[*PHONE*]

We define a connection to be a set of *PHONE*s:

$$CON == \mathbb{P} PHONE$$

This definition allows the possibility of *conference calls* (connections c such that $\#c > 2$), as well as *test calls* perhaps made by maintenance staff ($\#c = 1$). Even the *empty call* is possible (but perhaps pointless).

The state of the telephone network is described by two variables:

$$\begin{array}{ll} reqs : \mathbb{P} CON & \text{the set of connections requested but not yet terminated; and} \\ cons : \mathbb{P} CON & \text{the set of connections currently active.} \end{array}$$

These variables satisfy two invariants, i.e. properties of the system that are true initially and after every operation:

$$\begin{array}{ll} cons \subseteq reqs & \text{only requested connections are active; and} \\ cons \in disjoint & \text{no phone may engage in more than one connection at any time.} \end{array}$$

The ‘disjoint’ used here applies to a set of sets (in contrast to the one defined in the glossary). The definition of disjoint used here is the following:

$$\begin{array}{l} \boxed{\begin{array}{l} [X] \\ disjoint : \mathbb{P}(\mathbb{P} X) \\ \forall cons : \mathbb{P} X \bullet \\ cons \in disjoint \Leftrightarrow (\forall c1, c2 : cons \bullet c1 \neq c2 \Rightarrow c1 \cap c2 = \{\}) \end{array}} \end{array}$$

Our state is described by the schema *TN*.

$$\boxed{\begin{array}{l} \textit{TN} \\ reqs, cons : \mathbb{P} CON \\ cons \subseteq reqs \\ cons \in disjoint \end{array}}$$

We follow a commonly used Z convention that explicit conjunctions (\wedge symbols) are not required for a list of conjoined predicates written one per line.

An *efficient* network, however, would at any time have activated as many connections as possible. We describe it with the following schema, *efficientTN*:

$$\boxed{\begin{array}{l} \textit{efficientTN} \\ \textit{TN} \\ \neg (\exists cons0 : \mathbb{P} CON \bullet \\ cons \subset cons0 \wedge \\ \textit{TN}[cons0/cons]) \end{array}}$$

efficientTN requires the set of connections *cons* to be *maximal* with respect to *TN*, i.e. at any time it must not be possible to augment the set while continuing to satisfy the *TN* invariant. Notice that it is not necessary to select an algorithm for achieving maximality: it is sufficient simply to state that it is required. The definition of *efficientTN*, in full, is the following:

$\frac{\text{efficientTN}}{\text{reqs}, \text{cons} : \mathbb{P} \text{CON}}$
$\begin{array}{l} \text{cons} \subseteq \text{reqs} \\ \text{cons} \in \text{disjoint} \\ \neg (\exists \text{cons0} : \mathbb{P} \text{CON} \bullet \\ \quad \text{cons} \subset \text{cons0} \wedge \\ \quad \text{cons0} \subseteq \text{reqs} \wedge \\ \quad \text{cons0} \in \text{disjoint}) \end{array}$

Each of the three operations on the network are described in terms of the state before (*efficientTN*) and after (*efficientTN'*), and the phone from which it is initiated:

$ph? : \text{PHONE}$

We collect these in the schema ΔTN , but impose the additional constraint that a connection will never be terminated unless such termination is necessary to preserve the invariant.

$\frac{\Delta TN \quad \text{efficientTN} \quad \text{efficientTN}' \quad ph? : \text{PHONE}}{\neg (\exists \text{cons1} : \mathbb{P} \text{CON} \bullet \\ \quad (\text{cons} \setminus \text{cons1}) \subset (\text{cons} \setminus \text{cons}') \wedge \\ \quad \text{efficientTN}'[\text{cons1}/\text{cons}'])}$

ΔTN requires the set of connections terminated as an effect of the operation ($\text{cons} \setminus \text{cons}'$) to be *minimal* with respect to *efficientTN'*. That is, it must not be possible to diminish that set while maintaining the *efficientTN'* invariant.

Each of the three operations is expressed in terms of the ΔTN schema above, and any additional variables it requires individually.

3.2.1 Call

The *Call* operation requests a connection between the initiating phone $ph?$ and the phone $dialled?$. The request $\{ph?, dialled?\}$ is added to the set *reqs* of requests; the maximality constraint of *efficientTN'* ensures that if the request can be satisfied immediately, it will be; and the minimality constraint of ΔTN ensures that no other change will occur in *cons*.

$\frac{\text{Call} \quad \Delta TN \quad dialled? : \text{PHONE}}{\text{reqs}' = \text{reqs} \cup \{\{ph?, dialled?\}\}}$

3.2.2 HangUp

The *HangUp* operation terminates any connection in which the initiating phone is engaged; any such connection c is removed from the set of requests (which therefore forces it to be removed from the set of connections also).

$HangUp$	_____
ΔTN	
$reqs' = reqs \setminus \{c : cons \mid ph? \in c\}$	

3.2.3 Engaged

The *Engaged* operation indicates whether a phone is connected or not:

$Status ::= Yes \mid No$

$Engaged$	_____
ΔTN	
$engaged! : Status$	
$other! : PHONE$	
$\theta TN' = \theta TN$	
$(engaged! = Yes) \Rightarrow (\{ph?, other!\} \in cons)$	
$(engaged! = No) \Rightarrow ph? \notin (\bigcup cons)$	

3.3 Exercises

Exercise 3.1 ‘A connection is not terminated unless terminating it is necessary to preserve the invariant’. Justify the formulation of ΔTN by showing that

$$\forall \Delta TN \bullet (\forall c : cons \bullet c \in reqs' \Rightarrow c \in cons')$$

That is, any connection which (a) was active before an operation ($\in cons$); and (b) is still requested after the operation ($\in reqs'$) remains active ($\in cons'$).

Exercise 3.2 Show that it is possible for *HangUp* to activate a connection. That is, prove the following:

$$(\exists HangUp \bullet (cons' \setminus cons) \neq \{\})$$

Exercise 3.3 Add a variable

$avail : \mathbb{P} PHONE$

to the telephone network state; *avail* is to be, at any time, the set of phones that are *available* for connection (i.e. are in working order). Leave aside for the moment just what determines the value of *avail* itself; concentrate instead on what changes must be made to ensure that only available phones can be used (to initiate an operation) or engaged.

Exercise 3.4 Specify two operations for updating the variable *avail*. That is, define

AV
$avail : \mathbb{P} PHONE$

and two operations

$Break$
ΔAV
$phone? : PHONE$
????

and

Fix
ΔAV
$phone? : PHONE$
????

where *Break* makes a phone unavailable, and *Fix* makes it available again.

Exercise 3.5 Extend the operations *Break* and *Fix* (introduced in Exercise 3.4) so that they operate on the network state *efficientTN*, but do not change *reqs* (*cons* may *have* to change). Add to *Call*, *HangUp*, and *Engaged* the constraint that they do not change *avail*. Finally, present the entire specification as it now stands.

3.4 Solutions to exercises

Solution 3.1 Since from *efficientTN'* we have $cons' \subseteq reqs'$, it follows that

$$(cons \setminus reqs') \subseteq (cons \setminus cons')$$

But $(cons \setminus reqs') = cons \setminus (reqs' \cap cons)$, and so

$$cons \setminus (reqs' \cap cons) \subseteq cons \setminus cons' \quad (3.1)$$

Now if for some *c* it were not true that

$$(c \in cons \wedge c \in reqs') \Rightarrow c \in cons' \quad (3.2)$$

then we would have both of the following:

1. $c \notin (cons \setminus (reqs' \cap cons))$
2. $c \in (cons \setminus cons')$

which together would show that the inclusion (3.1) was strict:

$$cons \setminus (reqs' \cap cons) \subset cons \setminus cons'$$

Letting *cons1* be $reqs' \cap cons$, this contradicts ΔTN , and so we have established (3.2).

□

Solution 3.2 We assume that the set *PHONES* is big enough to contain at least three distinct elements; let them be *a*, *b*, and *c*. Then

$$\begin{aligned}
& \text{HangUp} \wedge \\
& \text{reqs} = \{\{a, b\}, \{a, c\}\} \wedge \\
& \text{cons} = \{\{a, b\}\} \wedge \\
& \text{ph?} = a \\
\Rightarrow & \\
& \text{reqs}' = \{\{a, c\}\} \wedge \text{cons}' = \{\{a, c\}\} \\
\Rightarrow & \\
& (\text{cons}' \setminus \text{cons}) \neq \{\} \quad \square
\end{aligned}$$

Solution 3.3 First extend the original *TN* (before *efficientTN*), specifying that only available phones can be engaged in a connection.

TN	TN $\text{avail} : \mathbb{P} \text{PHONE}$
$(\bigcup \text{cons}) \subseteq \text{avail}$	

The use of *TN* within its own definition is not recursive; the sub-schema name *TN* refers to its *previous* definition, and so the above is strictly an extension. We have redefined *TN* as follows:

TN	$\text{reqs}, \text{cons} : \mathbb{P} \text{CON}$ $\text{avail} : \mathbb{P} \text{PHONE}$
$\text{cons} \subseteq \text{reqs}$ $\text{cons} \in \text{disjoint}$ $(\bigcup \text{cons}) \subseteq \text{avail}$	

The schema *efficientTN* is textually the same as its previous definition, but now it includes the new definition of *TN*.

efficientTN	TN
$\neg (\exists \text{cons0} : \mathbb{P} \text{CON} \bullet$ $\quad \text{cons} \subset \text{cons0} \wedge \text{TN}[\text{cons0}/\text{cons}])$	

ΔTN has a similar definition to before, but it includes the new definition of *efficientTN* and the additional constraint that $\text{ph?} \in \text{avail}$.

ΔTN	efficientTN $\text{efficientTN}'$ $\text{ph?} : \text{PHONE}$
$\text{ph?} \in \text{avail} \wedge$ $\neg (\exists \text{cons1} : \mathbb{P} \text{CON} \bullet$ $\quad (\text{cons} \setminus \text{cons1}) \subset (\text{cons} \setminus \text{cons}') \wedge$ $\quad \text{efficientTN}'[\text{cons1}/\text{cons}'])$	

The definitions of *Call*, *HangUp* and *Engaged* are textually the same as before, but include the new definition of ΔTN . We repeat them here.

<i>Call</i>	_____
ΔTN	_____
$dialled? : PHONE$	_____
$reqs' = reqs \cup \{\{ph?, dialled?\}\}$	_____

<i>HangUp</i>	_____
ΔTN	_____
$reqs' = reqs \setminus \{c : cons \mid ph? \in c\}$	_____

<i>Engaged</i>	_____
ΔTN	_____
$engaged! : Status$	_____
$other! : PHONE$	_____
$\theta TN' = \theta TN$	_____
$(engaged! = Yes) \Rightarrow (\{ph?, other!\} \in cons)$	_____
$(engaged! = No) \Rightarrow ph? \notin (\bigcup cons)$	_____

Solution 3.4 First define the set of available phones:

<i>AV</i>	_____
$avail : \mathbb{P} PHONE$	_____

and the straightforward change of state schema:

ΔAV	_____
<i>AV</i>	_____
<i>AV'</i>	_____

The two operations are then

<i>Break</i>	_____
ΔAV	_____
$phone? : PHONE$	_____
$phone? \in avail$	_____
$avail' = avail \setminus \{phone?\}$	_____

and

<i>Fix</i>	_____
ΔAV	_____
$phone? : PHONE$	_____
$phone? \notin avail$	_____
$avail' = avail \cup \{phone?\}$	_____

Solution 3.5 We redefine ΔAV to include the complete state plus the constraint that the requests may not change.

ΔAV	
ΔTN	
$reqs' = reqs$	

The definitions of *Break* and *Fix* are textually the same as before, but now include the complete state. We do not repeat them here (see expansion below).

We redefine *Call*, *HangUp* and *Engaged* explicitly. None of these operations change the set of available phones.

$$\begin{aligned} Call &\hat{=} [Call \mid avail' = avail] \\ HangUp &\hat{=} [HangUp \mid avail' = avail] \\ Engaged &\hat{=} [Engaged \mid avail' = avail] \end{aligned}$$

The complete specification follows.

The state The state is as before.

TN	
$reqs, cons : \mathbb{P} CON$	
$avail : \mathbb{P} PHONE$	
$cons \subseteq reqs$	
$cons \in disjoint$	
$(\bigcup cons) \subseteq avail$	

The requirement that the set of connections cannot be increased is retained.

$efficientTN$	
TN	
$\neg (\exists cons0 : \mathbb{P} CON \bullet$	
$cons \subset cons0 \wedge$	
$TN[cons0/cons])$	

General properties of the operations

ΔTN	
$efficientTN$	
$efficientTN'$	
$ph? : PHONE$	
$ph? \in avail$	
$\neg (\exists cons1 : \mathbb{P} CON \bullet$	
$(cons \setminus cons1) \subset (cons \setminus cons') \wedge$	
$efficientTN'[cons1/cons'])$	

The operations Request a connection

$Call$ ΔTN $dialled? : PHONE$
$reqs' = reqs \cup \{\{ph?, dialled?\}\}$ $avail' = avail$

Terminate a connection

$HangUp$ ΔTN
$reqs' = reqs \setminus \{c : cons \mid ph? \in c\}$ $avail' = avail$

Determine telephone status

$Engaged$ ΔTN $engaged! : Status$ $other! : PHONE$
$\theta TN' = \theta TN$ $(engaged! = Yes) \Rightarrow (\{ph?, other!\} \in cons)$ $(engaged! = No) \Rightarrow ph? \notin (\bigcup cons)$

Break a telephone

$Break$ ΔAV $phone? : PHONE$
$phone? \in avail$ $avail' = avail \setminus \{phone?\}$ $reqs' = reqs$

Fix a telephone

Fix ΔAV $phone? : PHONE$
$phone? \notin avail$ $avail' = avail \cup \{phone?\}$ $reqs' = reqs$

3.5 Supplementary exercises

Exercise 3.6 Can a telephone call itself? If so, what is the effect of a subsequent *Engaged*?

Exercise 3.7 If an engaged telephone is suddenly broken, does the connection remain active? Is the request lost? What happens when the telephone is fixed?

Exercise 3.8 The system state is capable of describing conference calls, in which more than two telephones may be connected together, but some of the operations as now specified are inappropriate for this. Why? Modify those operations so that conference calls are supported.

Part II

SOFTWARE ENGINEERING

The software engineering project was the mainstay of the work on specification within the Programming Research Group. The work dates from 1979–80 when both Jean-Raymond Abrial and Cliff Jones were visiting the group at the invitation of Tony Hoare. In that year the seeds were sown for the work that is reported in this book.

The major part of the work done within the project has involved the application of techniques for mathematical specification to real systems, and a significant part of that work has involved industrial collaboration. The software engineering project spawned the project to study the formalisation of transaction processing reported in Part IV, and has worked closely with the distributed computing project reported in Part III.

The specification of the Unix filing system was the first published paper to use the schema notation as a mechanism for presenting mathematical specifications. It gives a gentle introduction to both the use of the notation and to the Unix filing system; as such it provides a more tutorial introduction to specification than the chapters that follow. The version presented here is slightly changed from that originally published in the March 1984 issue of the *IEEE Transactions on Software Engineering*.

The early work on specification included a project with STL on a Computer Aided Visitor Information And Retrieval (CAVIAR) system. The result of this work was an unpublished specification and an implementation. At the stage the work was done the schema notation had not been developed. The CAVIAR specification has since been reworked using the schema notation and has now been consolidated into Chapter 5. For the second edition this CAVIAR specification has been reworked yet again to make use of an experimental modularisation facility for Z.

A collaboration with ICL produced Chapter 6. It describes an existing ICL product – the ICL Data Dictionary. The chapter builds a mathematical model of the data dictionary, which provides the reader with an insight into the logical structure of the system. The precision of mathematics allows one to deduce properties of the system that are not made clear in the user manual for the system.

Chapter 7 describes a flexitime system; it provides a good example of the descriptive power of set theory. The specification makes use of a state which is richer than that necessary for an implementation, and this has as its reward an overall simplification. This inspiration for this example was provided by a contact in GEC.

Chapter 8 presents a specification of a simple assembler which was originally developed in response to another ‘specification’ that was in fact closer to an implementation design. The chapter shows how a specification should be based on fundamental properties of a proposed system rather than on a particular implementation strategy.

The design of a two-pass assembler and a justification that this design meets the specification are also given.

Chapter 4

Specification of the Unix filing system

Carroll Morgan and Bernard Sufrin

Abstract A specification of the Unix filing system is given using a notation based on elementary mathematical set theory. The notation used involves very few special constructs of its own.

The specification is detailed enough to capture the filing system's behaviour at the system call level, yet abstracts from issues of data representation, whether within programs or on the storage medium, and from the description of any algorithms which might be used to implement the system.

The presentation of the specification is in several stages, each new stage building on its predecessors; major concepts are introduced separately so that they may be easily understood. The notation used allows these separate stages to be joined together to give a complete description of each filing system operation – including its error conditions.

4.1 Introduction

The Unix [52] operating system is widely known, and its filing system is well understood. Why, then, do we present a formal specification of it here? It is because the idea of formalising the specification of computer-based systems has yet to receive widespread acceptance among computing practitioners, and in our view this is because very few realistic examples have been published. Publishing a *post hoc* specification of aspects of the Unix filestore offered us the possibility of showing how to use a mathematically based notation to capture important aspects of the behaviour of a system that is clearly not just a toy.

The use of natural language – *without* supporting mathematics – has serious limitations as a vehicle for the description of computer systems. As anyone who has ever

Copyright © 1984 IEEE. Reprinted, with permission, from *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 2, pp. 128–142.

used an operating system will confirm, the manuals cannot tell the whole story about the behaviour of a system. Indeed, almost every programmer who starts to use a new operating system sets up a number of *experiments*, by which she attempts to discover how it ‘really’ behaves. It is a commonplace observation that large computer systems, operating systems in particular, accumulate around themselves a body of folklore – necessary knowledge for anybody who wishes to use them effectively – and a number of ‘gurus’ – people who understand the hidden secrets of the system because they have read . . . the source code!

In our approach to the description of computer systems we use natural language *together with* the formal language of mathematics. And our particular style is simply a means of presenting the formal part of the description in a way that can be easily manipulated and understood. The formal descriptions themselves are given in elementary mathematical set theory, which is convenient for this purpose because programs are themselves mathematical objects [1, 20]. The difference between a mathematical specification and a program is only of degree: they are objects drawn from the same continuum. This uniformity allows, for example, the refinement of formal specifications into programs to be mathematically verified [33].

By using a mixture of natural language and elementary set theory we have enabled ourselves to give a description which is comprehensive enough to describe the essential aspects of the system’s behaviour, but is sufficiently abstract that it will not burden the reader with the kind of detail that appears in the source code. In particular, it has allowed us to avoid describing the representation of data on external media and within programs and to refrain from presenting details of the algorithms that are used to implement the filestore operations. Thus the specification here might occur midway along the path from a more abstract but informal specification – a description such as is given in [52] – to a more concrete one – the source code itself [38]. This intermediate level of abstraction is one which conveniently captures the behaviour of the system at the system call level, without being concerned with representational matters.

At each stage of presentation, the static (invariant) properties of the system are characterised by *naming* the observations that can be made of it, attributing a (set theoretical) *type* to each observation, and recording the invariant relationships between these observations as a collection of predicates.

The dynamic behaviour of the system is characterised by giving – for each of the operations under which the system evolves – the names of the observations that can be made *before* the operation, the names of those that can be made *after* the operation, and a collection of predicates that relate these two sets of observations. The operations in question in this case are just the Unix system calls, and the observations we are interested in may include components of the system state, and the ‘arguments’ and ‘results’ of system calls.

When providing a specification (such as this one) which is a ‘tutorial’ exercise rather than a reference manual, the concepts must be introduced gradually so as not to overwhelm the reader with immediate detail. The specification begins with the definition of a file alone, but ultimately includes channels (file descriptors), file identifiers (i-numbers), and even the abstract format of a directory file. Error conditions are treated last of all, so that they do not complicate the description of what usually happens with the problems of what might happen.

One novel aspect of the specification style is the use of a *homogeneous* framework – *schemas* – to characterise both dynamic and static properties. Schemas supplement the notation of set theory by providing notations for naming and combining groups

of observations and predicates, and methods of reasoning about the combinations; this is exactly what is needed to present the specification gradually. Moreover, since the tutorial style of the specification is based on mathematics, it is necessary when providing a reference manual only to collect its definitions into one unit – a summary, in effect – using the laws of combination of schemas.

The value of a specification such as this is that it defines the system in question, so that its properties may be determined by reasoning rather than by performing experiments on the system itself – these could be difficult (if the system is complex) and costly (if it has not yet been built). Since several specifications can be constructed for one system, each may take a point of view, or adopt a level of abstraction, which is appropriate to the questions it is required to answer. And if these specifications are presented within a formal framework, the question of their meaning and consistency is only a mathematical one, and so can be answered by mathematical means rather than by armwaving. But of course the *real* payoff is that when the system is built and in use, all those painful – and perplexing – visits to the guru can be avoided.

4.2 Scope of the specification

The system described is Unix Level 6. The operations covered include the system calls

read	write	create
seek	open	close
fstat	link	unlink

and the commands

ls	move
----	------

Some of the features not treated are

- special files;
- pipes; and
- file access permissions.

Some of the more practical considerations, such as storage device size, are examined in Appendix 4.5. The treatment of errors covers only a few examples, but illustrates the technique which would apply to them all.

4.3 The specification

4.3.1 Bytes and files

The ultimate constituent of the filing system is the *byte*; the set of all bytes is called *BYTE*:

$$BYTE == 0 \dots 255$$

A *file* is a finite *sequence* of bytes of any length¹ (including the null sequence $\langle \rangle$ of length 0):

$$FILE == \text{seq } BYTE$$

¹See Appendix 4.5.1.

In general, a sequence of X is a partial function from the natural numbers (\mathbb{N}) into X ; for any sequence s and natural number n , $s(n)$ is the n th element of s (if defined). Thus for any f of type *FILE* $f(1)$ is the first byte of the file. The function $\#$ gives the length of any sequence; hence $\#f$ is the size of the file, and $f(\#f)$ is its last byte.

4.3.2 Reading and writing

When a file is read the file itself is not changed; if $file'$ is the file's value after the operation, and $file$ is its value before, then

$$file' = file$$

The result of *reading* a file is a sequence $data!$ of bytes:

$$data! : \text{seq } BYTE$$

The value of $data!$ is determined by an offset into the file and a length to be read; both are natural numbers (i.e. non-negative integers):

$$offset?, length? : \mathbb{N}$$

and in fact

$$data! = (1 \dots length?) \triangleleft (file \text{ after } offset?)$$

The infix operator ‘after’ takes a sequence, in this case $file$, as its first argument and an offset as its second argument, and returns the subsequence of $file$ beginning after the offset. The first $length?$ bytes (if there are that many) are then selected from the resultant sequence to give the data returned by the read. The operator ‘after’ has the following definition:

$ \begin{array}{l} \text{[X]} \\ \text{-- after } _ : \text{seq } X \times \mathbb{N} \rightarrow \text{seq } X \\ \hline \forall s : \text{seq } X; \text{ offset} : \mathbb{N} \bullet \text{dom}(s \text{ after } offset) = (1 \dots \#s - offset) \wedge \\ (\forall n : \mathbb{N} \bullet \\ \quad (n + offset) \in \text{dom } s \Rightarrow \\ \quad \quad (s \text{ after } offset)(n) = s(n + offset)) \end{array} $
--

Therefore

$$(file \text{ after } offset?)$$

is a formalisation of

$$file \text{ after the first } offset? \text{ bytes}$$

This means that the first byte of the file has offset 0.

The domain restriction operator (\triangleleft) here excludes any element whose index is not in the set $1 \dots length?$; $data!$ is therefore

$$file, \text{ after } offset?, \text{ for no more than } length?$$

For example, if

$$\begin{aligned} file &= \langle X, A, N, F, R, E, D \rangle \\ offset? &= 2 \\ length? &= 3 \end{aligned}$$

then

$$(file \text{ after } 2)(n) = file(n + 2)$$

That is,

$$file \text{ after } offset? = \langle N, F, R, E, D \rangle$$

and therefore

$$data! = (1 \dots 3) \triangleleft \langle N, F, R, E, D \rangle = \langle N, F, R \rangle$$

All of these properties may be collected in a *schema* which defines the reading operation:

$\begin{aligned} file, file' &: FILE \\ offset?, length? &: \mathbb{N} \\ data! &: \text{seq } BYTE \end{aligned}$
$\begin{aligned} file' &= file \\ data! &= (1 \dots length?) \triangleleft (file \text{ after } offset?) \end{aligned}$

When a schema is used (as it is here) to characterise an operation, its *signature*

$$\begin{aligned} file, file' &: FILE \\ offset?, length? &: \mathbb{N} \\ data! &: \text{seq } BYTE \end{aligned}$$

gives names and types to the observations that can be made before and after the operation. The *predicate*

$$\begin{aligned} file' &= file \\ data! &= (1 \dots length?) \triangleleft (file \text{ after } offset?) \end{aligned}$$

relates these observations to one another.

Naming a schema allows it to be referred to within subsequent definitions; the name is written as part of the enclosing ‘box’.

$\begin{aligned} file, file' &: FILE \\ offset?, length? &: \mathbb{N} \\ data! &: \text{seq } BYTE \end{aligned}$
$\begin{aligned} file' &= file \\ data! &= (1 \dots length?) \triangleleft (file \text{ after } offset?) \end{aligned}$

The definition above can be read:

The *readFILE* operation does not change the file. It expects an offset and length as parameters, and returns as its result the data read. The value returned is the longest sequence of bytes, of length not greater than that requested, which begins at the given offset in the file.

To define the *writeFILE* operation, a similar schema is used; this time, however, the file *is* changed.

The byte *ZERO* is used in the definition of *writeFILE*; it is a distinguished element of *BYTE*:

$$ZERO == 0$$

And *zero(k)* is a sequence of length *k* containing only *ZERO* bytes:

$$\frac{}{\text{zero} : \mathbb{N} \rightarrow \text{seq } \text{BYTE}} \quad \frac{}{\forall n : \mathbb{N} \bullet \text{zero}(n) = (\lambda k : 1 \dots n \bullet ZERO)}$$

Writing with an offset greater than the file length leaves *ZERO* bytes between the previous end of the file and the newly written data.

$$\frac{\text{writeFILE} \quad \text{file}, \text{file}' : \text{FILE} \quad \text{offset?} : \mathbb{N} \quad \text{data?} : \text{seq } \text{BYTE}}{\text{file}' = \text{zero}(\text{offset?}) \oplus \text{file} \oplus (\text{data?} \text{ shift } \text{offset?})}$$

The infix operator ‘shift’ takes a sequence, in this case *data?* and an offset and shifts *data?* by the offset. It has the following definition:

$$\frac{[X] \quad \text{-- shift --} : \text{seq } X \times \mathbb{N} \rightarrow (\mathbb{N} \leftrightarrow X)}{\forall s : \text{seq } X; \text{offset} : \mathbb{N} \bullet \text{dom}(s \text{ shift } \text{offset}) = \{i : \text{dom } s \bullet i + \text{offset}\} \wedge (\forall n : \text{dom}(s \text{ shift } \text{offset}) \bullet (s \text{ shift } \text{offset})(n) = s(n - \text{offset}))}$$

‘ \oplus ’ is the function overriding operator: $f \oplus g$ behaves like g except where g is undefined, in which case it behaves like f . Thus the value of any byte in the file

$$\text{zero}(\text{offset?}) \oplus \text{file} \oplus (\text{data?} \text{ shift } \text{offset?})$$

is determined first by the written *data?*, then by the previous contents of the *file*, and finally is *ZERO* otherwise. The length of the new file is

$$\max(\# \text{file}, \text{offset?} + \# \text{data?})$$

Thus

$$\begin{aligned} \text{file} &= \langle X, A, N, F, R, E, D \rangle \wedge \\ \text{offset?} &= 8 \wedge \end{aligned}$$

$$\begin{aligned}
data? &= \langle N, U, N, I, B, A, D \rangle \\
\Rightarrow \\
file' &= \langle \sqcup, \sqcup, \sqcup, \sqcup, \sqcup, \sqcup, \sqcup, \sqcup \rangle \oplus \langle X, A, N, F, R, E, D \rangle \oplus \\
&\quad (\langle N, U, N, I, B, A, D \rangle \text{ shift } 8) \\
\Rightarrow \\
file' &= \langle X, A, N, F, R, E, D, \sqcup \rangle \oplus (\langle N, U, N, I, B, A, D \rangle \text{ shift } 8) \\
\Rightarrow \\
file' &= \langle X, A, N, F, R, E, D, \sqcup, N, U, N, I, B, A, D \rangle
\end{aligned}$$

(The byte *ZERO* is here represented by a ' \sqcup '.)

A consequence of this definition is that *writeFILE* is possible for *all* values of *file*, *offset?*, and *data?* (subject to any limitation on the maximum size of files in general); formally, this is shown by proving that there is always a value for *file'*, consistent with its type *FILE* (seq *BYTE*), such that the following predicate holds:

$$file' = zero(offset?) \oplus file \oplus (data? \text{ shift } offset?)$$

4.3.3 File storage

The *file storage* system allows files to be stored and retrieved using *file identifiers*; the set of all file identifiers is called *FID*:

$$[FID]$$

The storage system is characterised by a single observation: a partial function² from *FID* to *FILE*.

$$\begin{array}{c}
SS \\
\hline
fstore : FID \leftrightarrow FILE
\end{array}$$

An empty file may be *created* in the storage system by supplying its identifier as a parameter to an operation which changes an *old* storage system, *SS*, into a *new* one which contains the created file, *SS'*. *SS* is equivalent to

$$fstore : FID \leftrightarrow FILE$$

so *SS'* is equivalent to

$$fstore' : FID \leftrightarrow FILE$$

Thus, the effect of decorating a schema name is to decorate the names of its observation(s).

The operation that creates an empty file is defined by the schema

$$\begin{array}{c}
createSS \\
\hline
\begin{array}{c}
SS \\
SS' \\
fid : FID
\end{array} \\
\hline
fstore' = fstore \oplus \{fid \mapsto \langle \rangle\}
\end{array}$$

²See Appendices 4.5.2 and 4.5.3.

The new store $fstore'$ is identical to the old except that fid now refers to the empty file $\langle \rangle$ – *whether or not it referred to a file previously*. Thus creating an existing file empties it. We do not write ' $fid?$ ' because later it will be seen that these file identifiers are in fact not visible to the user.

Destroying a file is defined

$destroySS$	
SS	
SS'	
$fid : FID$	
<hr/>	
$fid \in \text{dom } fstore$	
$fstore' = \{fid\} \triangleleft fstore$	

Naturally, a file must exist ($\in \text{dom } fstore$) to be destroyed. The new $fstore'$ is identical to the old except that there is no file referred to by fid :

$$fid \notin \text{dom } fstore'$$

4.3.4 Reading and writing stored files – framing

Reading a *stored* file is defined by the following schema:

SS	
SS'	
$fid : FID$	
$offset?, length? : \mathbb{N}$	
$data! : \text{seq } BYTE$	
$file, file' : FILE$	
<hr/>	
$fid \in \text{dom } fstore$	
$file = fstore(fid)$	
$data! = (1 \dots length?) \triangleleft (file \text{ after } offset?)$	
$file' = file$	
$fstore' = fstore \oplus \{fid \mapsto file'\}$	

The file read is that referred to by fid , the data output is from $offset?$ for $length?$ (as before), and the file is not changed.

This long-winded definition of reading a stored file shows that it is in fact a combination of the definitions given above for

- reading a file ($readFILE$); and
- the storage system (SS).

This kind of combination is called *framing*, because it involves specifying

- *which* file is read or written; and
- that the *other* files are unaffected.

That is, a frame is supplied within which the operation occurs. The following schema states this framing combination generally:

$$\begin{array}{c}
 \Phi SS \\
 \hline
 SS \\
 SS' \\
 file, file' : FILE \\
 fid : FID \\
 \hline
 fid \in \text{dom } fstore \\
 file = fstore(fid) \\
 fstore' = fstore \oplus \{fid \mapsto file'\}
 \end{array}$$

fid denotes the file affected in $fstore$ – namely $(file, file')$ – and no other file is changed. Φ is conventionally used as the first letter of framing schemas (Φ for *frame*).

Although the definition given above of reading a stored file could have stated explicitly that the filestore is not changed – $fstore' = fstore$ – this is really a *consequence* of the fact that the file itself is not changed. And the framing schema ΦSS makes it much easier to write such definitions generally – for example, the operation above could be defined as follows:

$$\begin{array}{c}
 readSS \\
 \hline
 \Phi SS \\
 readFILE
 \end{array}$$

The signatures and predicates of the two schemas are combined separately and then joined to form the new schema. Where the two schemas *share* a named observation in their signatures, it appears only once in the new schema. Thus, although $file$ and $file'$ occur in both $readFILE$ and ΦSS , they appear only once in $readSS$.

Writing a stored file is defined similarly.

$$\begin{array}{c}
 writeSS \\
 \hline
 \Phi SS \\
 writeFILE
 \end{array}$$

Its definition may be expanded:

$$\begin{array}{c}
 SS \\
 SS' \\
 fid : FID \\
 offset? : \mathbb{N} \\
 data? : \text{seq } BYTE \\
 file, file' : FILE \\
 \hline
 fid \in \text{dom } fstore \\
 file = fstore(fid) \\
 file' = \text{zero}(offset?) \oplus file \oplus (data? \text{ shift } offset?) \\
 fstore' = fstore \oplus \{fid \mapsto file'\}
 \end{array}$$

As in $readSS$, $file$ and $file'$ appear only once in this combination.

4.3.5 Hiding and simplification

In the schema $readSS$ the observations $file$ and $file'$ are entirely determined in value by the other observations of the schema. Unless it is necessary to observe the *whole* $file$ involved in a read or write operation, these observations have become inessential to the specification. Observations such as these are called *auxiliary*.

Hiding auxiliary observations can allow simplification of the schema in which they occur. Components are hidden by removing them from the signature of the schema and by existentially quantifying them in the predicate part. $readSS$, with $file$ and $file'$ hidden, is written $readSS \setminus (file, file')$ and is in full

SS SS' $fid : FID$ $offset?, length? : \mathbb{N}$ $data! : seq\ BYTE$
$(\exists file, file' : FILE \bullet$ $\quad fid \in dom\ fstore$ $\quad file = fstore(fid)$ $\quad data! = (1 \dots length?) \triangleleft (file\ after\ offset?)$ $\quad file' = file$ $\quad fstore' = fstore \oplus \{fid \mapsto file'\})$

This schema can be simplified using basic predicate calculus:

SS SS' $fid : FID$ $offset?, length? : \mathbb{N}$ $data! : seq\ BYTE$
$fid \in dom\ fstore$ $fstore' = fstore$ $data! = (1 \dots length?) \triangleleft (fstore(fid)\ after\ offset?)$

Writing may be treated similarly.

4.3.6 Sequential access to files

The read and write operations described so far support random access; in order to allow easy sequential use of these operations, a *channel* is defined which remembers the current position in the file.

$CHAN$ $fid : FID$ $posn : \mathbb{N}$
--

A channel has a file identifier *fid* – which may refer to a file in *fstore* – and a position *posn* within the file. As usual, operations involving the channel take the form of a predicate relating the observations of

CHAN

to those of

CHAN'

They have the additional property that the *fid* of a channel is never changed. The schema $\Delta CHAN$ expresses the general properties of any operation on a channel (Δ for *change*).

$\Delta CHAN$	_____
<i>CHAN</i>	
<i>CHAN'</i>	
$fid' = fid$	

Sequential reading and writing using channels is easily characterised by combining the previous definitions.

<i>readCHAN</i>	_____
<i>readSS</i>	
$\Delta CHAN$	
$offset? = posn$	
$posn' = posn + \#data!$	

<i>writeCHAN</i>	_____
<i>writeSS</i>	
$\Delta CHAN$	
$offset? = posn$	
$posn' = posn + \#data?$	

In addition, there is an operation *seekCHAN* which changes only the position.³

<i>seekCHAN</i>	_____
<i>SS</i>	
<i>SS'</i>	
$\Delta CHAN$	
$newposn? : \mathbb{N}$	
$fstore' = fstore$	
$posn' = newposn?$	

The new position is not constrained to be within the file.⁴

³See Appendix 4.5.4.

⁴See Appendix 4.5.5.

4.3.7 Channel system

A *channel storage* system may be defined which is analogous to the file storage system; it allows channels to be stored and retrieved using channel identifiers taken from the set CID . A channel identifier is a Unix ‘file descriptor’:

$[CID]$

$$\frac{CS}{cstore : CID \leftrightarrow CHAN}$$

Operations on the channel system have the general form

$$\frac{\Delta CS}{\begin{array}{l} CS \\ CS' \end{array}}$$

These operations are defined below:

$$\frac{\begin{array}{l} openCS \\ \Delta CS \\ CHAN \\ cid! : CID \end{array}}{\begin{array}{l} cid! \notin \text{dom } cstore \\ posn = 0 \\ cstore' = cstore \oplus \{cid! \mapsto \theta CHAN\} \end{array}}$$

$openCS$ creates a new channel and returns a new identifier which refers to it; the new channel’s position is zero. $\theta CHAN$ stands for the ‘pair’ with components $posn$ and fid . In this case the component $posn$ is zero and the component fid is unconstrained (its value will be determined at a later stage).

$$\frac{\begin{array}{l} closeCS \\ \Delta CS \\ cid? : CID \end{array}}{\begin{array}{l} cid? \in \text{dom } cstore \\ cstore' = \{cid?\} \triangleleft cstore \end{array}}$$

$closeCS$ removes a channel from the channel system.

4.3.8 The access system

The storage and channel systems together form the *access* system.

$$\frac{\begin{array}{l} AS \\ SS \\ CS \end{array}}{\{chan : \text{ran } cstore \bullet chan.fid\} \subseteq \text{dom } fstore}$$

The predicate in the above schema requires that every channel must refer to an existing file. This property is an *invariant* of the access system and is preserved by all operations on it. The schema ΔAS automatically includes the invariant of both the initial (AS) and final (AS') state.

ΔAS
AS
AS'

Reading, writing and seeking in the access subsystem are defined with the assistance of a framing schema.

ΦAS
ΔAS
$\Delta CHAN$
$cid? : CID$
$cid? \in \text{dom } cstore$
$\theta CHAN = cstore(cid?)$
$cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\}$

$\theta CHAN$ in the predicate part is the channel with components fid and $posn$ as they appear in $\Delta CHAN$; $\theta CHAN'$ is similar but with components fid' and $posn'$.

Reading, writing and seeking in the access system are now defined by combination of previous definitions and the framing schema ΦAS ; as usual, some auxiliary variables will be hidden.

The operator \wedge when applied to two schemas is shorthand for writing the two together; that is,

$$\Phi AS \wedge readCHAN$$

is just

ΦAS
$readCHAN$

The definitions are

$$\begin{aligned} readAS &\hat{=} (\Phi AS \wedge readCHAN) \setminus (offset?, fid', posn', file') \\ writeAS &\hat{=} (\Phi AS \wedge writeCHAN) \setminus (offset?, fid', posn') \\ seekAS &\hat{=} (\Phi AS \wedge seekCHAN) \setminus (fid, fid', posn, posn') \end{aligned}$$

which when expanded and simplified give

<i>readAS</i>	_____
ΔAS	
$cid? : CID$	
$length? : \mathbb{N}$	
$data! : \text{seq } BYTE$	
$CHAN$	
$file : FILE$	

$cid? \in \text{dom } cstore$	
$\theta CHAN = cstore(cid?)$	
$file = fstore(fid)$	
$fstore' = fstore$	
$(\exists CHAN' \bullet posn' = posn + \#data! \wedge$	
$fid' = fid \wedge$	
$cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\} \wedge$	
$data! = (1 \dots length?) \triangleleft (file \text{ after } posn)$	

and

<i>writeAS</i>	_____
ΔAS	
$cid? : CID$	
$data? : \text{seq } BYTE$	
$CHAN$	
$file, file' : FILE$	

$cid? \in \text{dom } cstore$	
$\theta CHAN = cstore(cid?)$	
$file = fstore(fid)$	
$file' = zero(posn) \oplus file \oplus (data? \text{ shift } posn)$	
$fstore' = fstore \oplus \{fid \mapsto file'\}$	
$(\exists CHAN' \bullet posn' = posn + \#data? \wedge$	
$fid' = fid \wedge$	
$cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\}$	

and

<i>seekAS</i>	_____
ΔAS	
$cid? : CID$	
$newposn? : \mathbb{N}$	

$cid? \in \text{dom } cstore$	
$fstore' = fstore$	
$(\exists CHAN' \bullet posn' = newPosn? \wedge$	
$fid' = (cstore cid?).fid \wedge$	
$cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\}$	

In addition to the three operations above, the *fstat* operation, which returns the size of the file accessed with a given *CID*, can be defined by

$fstat$	
ΔAS	
$cid? : CID$	
$size! : \mathbb{N}$	
$cid? \in \text{dom } cstore$	
$fstore' = fstore$	
$cstore' = cstore$	
$size! = \#(fstore((cstore \ cid?).fid))$	

4.3.9 A file naming system

The naming system associates file names from the set $NAME$ with file identifiers FID ; these file names will normally be chosen by the users of the file system.

$NS0$	
$nstore : NAME \leftrightarrow FID$	

To create an association in the naming system, a *name* and *fid* are supplied; the new association *overrides any existing association for that name*.

$createNS$	
$\Delta NS0$	
$name? : NAME$	
$fid : FID$	
$nstore' = nstore \oplus \{name? \mapsto fid\}$	

Given a *name*, its *fid* may be discovered.

$lookupNS$	
$\Xi NS0$	
$name? : NAME$	
$fid' : FID$	
$name? \in \text{dom } nstore$	
$fid' = nstore(name?)$	

The schema $\Xi NS0$ expresses the observation that the naming system is unaffected; its definition is

$\Xi NS0$	
$NS0$	
$NS0'$	
$nstore' = nstore$	

ΞCS and ΞSS are defined similarly.

Finally, given a *name?*, any association it has may be destroyed (this is the *unlink* operation).

$destroyNS$
$\Delta NS0$
$name? : NAME$
$name? \in \text{dom } nstore$
$nstore' = \{name?\} \triangleleft nstore$

4.3.10 Pathnames and directories

By further revealing file names to be sequences of *syllables*

$$[SYL] \\ NAME == \text{seq } SYL$$

it is possible to provide more structure in the name space as a whole (the name space is $\text{dom } nstore$). The naming system is augmented by a set of *directory* names $dnames$:

NS
$NS0$
$dnames : \mathbb{P} NAME$
$front(\langle dnames \cup \text{dom } nstore \rangle) \subseteq dnames$

\mathbb{P} is the *powerset* constructor. The fat brackets ($\langle \rangle$) denote application of the function (*front* in this case) to a *set* of arguments to yield a *set* of results. That is,

$$front(\langle S \rangle) = \{s : S \mid s \in \text{dom } front \bullet front(s)\}$$

The *front* of a sequence is obtained by removing its last element; only the empty sequence ('root') has no *front*. The predicate states that the *front* of every (file or directory) name must itself be a directory name (i.e. every file or directory – except root – must appear in some directory). For example, if $\text{dom } nstore$ included

/Carroll/Unix/paper
/dev/sanders
/Bernard/IEEE/Unixpaper
/Bernard/Mumble

(where syllables are preceded by */*) then $dnames$ would necessarily include

/
/Carroll
/dev
/Bernard
/Carroll/Unix
/Bernard/IEEE

Given a directory name $dir?$, the operation $lsNS$ reveals its 'contents'.

$lsNS$
ΞNS
$dir? : NAME$
$contents! : \mathbb{P} SYL$
$dir? \in dnames$
$contents! = last(\langle \{n : \text{dom } nstore \mid n \neq \langle \rangle \wedge front\ n = dir?\} \rangle)$

The *last* of a sequence is its final element.

4.3.11 Directories are files

An additional constraint on the Unix system is that directories are in fact stored as files; they can be read by users. That is,

$$dnames \subseteq \text{dom } nstore$$

dirformat is a function that maps a *FILE* to the directory structure it represents:

$$\left| \begin{array}{l} \text{dirformat} : \text{FILE} \leftrightarrow (\text{SYL} \leftrightarrow \text{FID}) \\ \text{RootFid} : \text{FID} \end{array} \right.$$

The mathematical definition of *dirformat* would be the definition of the format of a directory file – but such a definition need not be given here. *RootFid* is the *FID* of the root directory $\langle \rangle$. The content of each directory file is determined by the system in accordance with the following requirement:

$\begin{array}{l} \text{dirstored} \\ \text{SS} \\ \text{NS} \end{array}$ <hr style="border: 0.5px solid black;"/> $\begin{array}{l} \text{ran } nstore \subseteq \text{dom } fstore \\ nstore = (\lambda n : \text{NAME} \mid n \neq \langle \rangle \wedge n \in \text{dom } nstore \bullet \\ \quad (\text{dirformat}(fstore(nstore(\text{front } n))))(\text{last } n)) \\ \cup \{ \langle \rangle \mapsto \text{RootFid} \}. \end{array}$
--

The constraint above may be paraphrased as follows:

The association of names and file identifiers (*nstore*) is found by taking for any name ($\lambda n : \text{NAME} \dots$) all of its syllables except the last (*front* *n*); finding the file identifier so referred to (*nstore* \dots); finding the contents of that file (*fstore* \dots); interpreting those contents as a directory (*dirformat* \dots); and finally using the last syllable of the original name (*last* *n*) to obtain a file identifier from that directory – unless the original name is empty, in which case its file identifier is *RootFid*.

4.3.12 The complete filing system

The complete filing system is described by combining the descriptions of the three separate systems above: the storage systems *SS*, the channel system *CS*, and the name system *NS*.

$\begin{array}{l} \text{FS} \\ \text{SS} \\ \text{CS} \\ \text{NS} \end{array}$ <hr style="border: 0.5px solid black;"/> $\text{usedfids} : \mathbb{P} \text{FID}$ <hr style="border: 0.5px solid black;"/> $\begin{array}{l} \text{usedfids} = \text{ran } nstore \cup \{ \text{chan} : \text{ran } cstore \bullet \text{chan.fid} \} \\ \text{usedfids} \subseteq \text{dom } fstore \end{array}$

The auxiliary observation *usedfids* is introduced; it is the set of file identifiers in use at any time, either in the channel store or the name store. The predicate states that all file identifiers in use must refer to an existing file in the file store; members of $(\text{dom } fstore \setminus \text{usedfids})$ are the *fids* of files which may be destroyed (since they are not referred to).

The filing system operations can be specified by combining the definitions of their effects on each separate subsystem. The *createFS* operation, for example, makes an empty file in the storage system, a new channel referring to it in the channel system, and associates a name with it in the naming system.

<i>createFS0</i>	_____
ΔFS	
<i>createSS</i>	
<i>openCS</i>	
<i>createNS</i>	
<hr/>	
$name? \in \text{dom } nstore \Rightarrow fid = nstore(name?)$	
$name? \notin \text{dom } nstore \Rightarrow fid \notin \text{usedfids}$	

If an *existing* name is created, the file it refers to is emptied – i.e. it is simply replaced with an empty file, and its previous contents are lost. If the name does not exist in the naming system, a new *fid* is chosen which is not currently in use.

The channel identifier of a channel referring to the new (or newly truncated) file is returned (*cid!* is an observation of *openCS*).

The definition of *createFS* above is not sufficient. Remember that the name store is encoded in the file store as directory files. In the case where a new name is added to the name store, it also needs to be added to the (encoded) directory in the file store. We define the following schema, which updates the directory files in the file store without changing the non-directory files or the name store. It makes use of the schema *dirstored* on the final state to ensure the name store is correctly encoded into the file store.

<i>direncode</i>	_____
ΔFS	
ΞCS	
ΞNS	
<i>dirstored'</i>	
<hr/>	
$\exists dfids : \mathbb{P} FID \bullet dfids = nstore(\downarrow dnames \downarrow) \wedge$	
$dfids \triangleleft fstore' = dfids \triangleleft fstore$	

The only difference between the file store before encoding and the file store after is the contents of directory files. Before encoding they may not accurately represent the name store but afterwards they must.

The definition of *createFS* can now be completed. It is the schema composition (\circ) of *createFS0* and *direncode*. The definition of schema composition is given in Section 4.3.14.

$$createFS \triangleq createFS0 \circ direncode$$

open returns the channel identifier of an existing file.

<i>openFS</i>	_____
ΔFS	
ΞSS	
<i>openCS</i>	
<i>lookupNS</i>	
<i>fid</i> = <i>fid'</i>	

The *fid'* returned by *lookupNS* is equal to the *fid* supplied to *openCS* (and both *fid'* and *fid* are good candidates for hiding).

read and *write* do not change the name store.

<i>readFS</i>	_____
ΔFS	
<i>readAS</i>	
ΞNS	

<i>writeFS</i>	_____
ΔFS	
<i>writeAS</i>	
ΞNS	

close removes the association between a channel name and the channel it refers to.

<i>closeFS</i>	_____
ΔFS	
ΞSS	
<i>closeCS</i>	
ΞNS	

unlink removes a name from the naming system, but it does *not* destroy the associated file.

<i>unlinkFS0</i>	_____
ΔFS	
ΞSS	
ΞCS	
<i>destroyNS</i>	

As with *createFS*, this operation updates the name store. Hence the encoded version of the name store in the file store also needs to be updated.

$$unlinkFS \hat{=} unlinkFS0 \circ direncode$$

Destroy removes a file from the filing system.

<i>destroyFS</i>	_____
ΔFS	
<i>destroySS</i>	
ΞCS	
ΞNS	

But can a file be destroyed while it is in use? The FS' invariant requires that

$$usedfids' \subseteq \text{dom } fstore' \quad (4.1)$$

and from ΞCS and ΞNS it follows that

$$usedfids = usedfids' \quad (4.2)$$

and so, from (4.1) and (4.2),

$$usedfids \subseteq \text{dom } fstore' \quad (4.3)$$

But

$$destroySS \Rightarrow fid \notin \text{dom } fstore' \quad (4.4)$$

and (4.3) and (4.4) give

$$destroySS \Rightarrow fid \notin usedfids \quad (4.5)$$

That is, a file cannot be destroyed while it is in use.

4.3.13 Honesty of definitions

The constraint on the destroy operation

$$fid \notin usedfids$$

is not immediately obvious from its definition above. Because the constraint is implicit, the above definition could be said to be dishonest.

An honest definition is one for which the conditions of applicability are explicit. In general, a schema which describes an operation can be expanded to have the form

<i>operation</i>	_____
<i>STATE</i>	
<i>STATE'</i>	
<i>IN?</i>	
<i>OUT!</i>	
<i>inv</i> (<i>STATE</i>)	
<i>pre</i> (<i>STATE</i> , <i>IN?</i>)	
<i>trans</i> (<i>STATE</i> , <i>IN?</i> , <i>OUT!</i> , <i>STATE'</i>)	
<i>post</i> (<i>STATE'</i> , <i>OUT!</i>)	
<i>inv</i> (<i>STATE'</i>)	

where $P(S)$ denotes a predicate in which the observations of S may occur free.

$STATE$, $STATE'$, $IN?$, and $OUT!$ are schemas with no predicates – they are just signatures.

inv is the state invariant, *pre* and *post* are the pre- and post-conditions respectively, and *trans* is the predicate expressing the relationship between the initial state, inputs, outputs, and final state. The conjunction of the five predicates forms the definition of the operation, but the definition is said to be *honest* only if

$$inv \wedge pre \Rightarrow (\exists OUT!; STATE' \bullet trans \wedge post \wedge inv)$$

If the invariant holds, and the input satisfies its precondition, then the operation should have at least one defined result. Thus, in an honest definition, applicability can be determined by considering the precondition alone (if all operations preserve the invariant). This is an honest definition of destroy:

destroyFS	_____
ΔFS	
destroySS	
ΞCS	
ΞNS	
$fid \notin \text{usedfids}$	

It is, however, *mathematically* equivalent to its original definition above.

For any schema describing an operation, a suitably honest precondition can be discovered by hiding the *OUT!* and *STATE'* observations, and simplifying the resulting predicate.

4.3.14 Observation renaming and schema composition

It may be necessary at times to rename the observations of a schema to avoid name clashes with other schemas. Writing

$$\text{schema}[\text{name2}/\text{name1}]$$

denotes the result of systematically substituting *name2* for *name1* throughout *schema* (with suitable renaming of bound variables if necessary). For example:

$$\text{createNS}[\text{newname?}/\text{name?}] =$$

ΔNS	_____
$\text{newname?} : NAME$	
$fid : FID$	
$nstore' = nstore \oplus \{\text{newname?} \mapsto fid\}$	

and

$$\text{lookupNS}[\text{oldname?}/\text{name?}] =$$

ΔNS	_____
$\text{oldname?} : NAME$	
$fid' : FID$	
$\text{oldname?} \in \text{dom } nstore$	
$fid' = nstore(\text{oldname?})$	
$nstore' = nstore$	

The *composition* of two schemas, written

$$schema1 \circledast schema2$$

is intended to capture the effect of ‘*schema1* then *schema2*’. It is formed by

1. Determining all of the dashed observations of *schema1* that correspond with undashed observations of *schema2* (*name'* corresponds with *name*).
2. Renaming each corresponding pair to a single new name

$$\begin{array}{l} schema1[name''/name'] \\ schema2[name''/name] \end{array}$$

3. Combining the schemas, and hiding the new observations

$$\begin{array}{l} schema1 \circledast schema2 \hat{=} \\ (schema1[name''/name'] \wedge \\ schema2[name''/name]) \setminus (name''). \end{array}$$

This operation allows schemas to be combined in a way suggestive of forward functional composition: the final state of *schema1* becomes the initial state of *schema2*. For example:

$$linkNS \hat{=} lookupNS[oldname?/name?]\circledast createNS[newname?/name?]$$

gives in full:

$$\begin{array}{l} \frac{linkNS}{\Delta NS} \\ \frac{oldname?, newname? : NAME}{oldname? \in \text{dom } nstore} \\ nstore' = nstore \oplus \{newname? \mapsto nstore(oldname?)\} \end{array}$$

The hidden observations are *nstore* and *fid*. *linkNS* makes the filename *newname?* refer to the same file as does *oldname?*.

A similar construction defines *moveNS*:

$$moveNS \hat{=} linkNS \circledast destroyNS[oldname?/name?]$$

That is,

$$\begin{array}{l} \frac{moveNS}{\Delta NS} \\ \frac{oldname?, newname? : NAME}{oldname? \in \text{dom } nstore} \\ nstore' = (\{oldname?\} \triangleleft nstore) \oplus \{newname? \mapsto nstore(oldname?)\} \end{array}$$

moveNS renames a file from *oldname?* to *newname?*. It is important that the two occurrences of *oldname?* – in *linkNS* and *destroyNS[oldname?/name?]* – are merged,

and so only one file is referred to. However, *oldname?* appears only once in the signature of *moveNS*.

Combining the definitions of *linkNS* and *moveNS* above, with ΞSS and ΞCS , gives their definitions in the complete file system FS.

$$\begin{aligned} linkFS &\hat{=} (\Delta FS \wedge \Xi SS \wedge \Xi CS \wedge linkNS) \circ direncode \\ moveFS &\hat{=} (\Delta FS \wedge \Xi SS \wedge \Xi CS \wedge moveNS) \circ direncode \end{aligned}$$

Because both these operations update the name store, we need to update the encoded version of the name store in the file store.

4.3.15 Definition of error conditions

The definitions given so far describe only *successful* operations. For example, the schema

$$\begin{array}{|l} \hline lookupNS \\ \hline \Xi NS \\ name? : NAME \\ fid' : FID \\ \hline name? \in \text{dom } nstore \\ fid' = nstore(name?) \\ \hline \end{array}$$

gives no indication of the result of looking up a name that is *not* in the name store. In fact, the definition explicitly states that the name must be there

$$name? \in \text{dom } nstore.$$

It is to that extent unrealistic.

To describe unsuccessful as well as successful operations, a schema is introduced below which includes an *error report* observation. The following error reports are used:

$$REPORT ::= Ok \mid NoSuchCid \mid NoSuchName \mid NoFreeCids$$

$$\begin{array}{|l} \hline \Delta FS \\ FS \\ FS' \\ report! : REPORT \\ \hline report! \neq Ok \Rightarrow (\theta FS' = \theta FS) \\ \hline \end{array}$$

The predicate states that in the event of an unsuccessful report

$$report! \neq Ok$$

the system's state is unaltered ($\theta FS' = \theta FS$). Successful operations are described by the schema below:

$$\begin{array}{|l} \hline success \\ \hline \Delta FS \\ \hline report! = Ok \\ \hline \end{array}$$

The following schemas define typical failures:

$CidErr$	_____
ΔFS	
$cid? : CID$	
$cid? \notin \text{dom } cstore$	
$report! = NoSuchCid$	

$CidErr$ describes an attempt to use a non-existent channel identifier. Two other common errors are

$NameErr$	_____
ΔFS	
$name? : NAME$	
$name? \notin \text{dom } nstore$	
$report! = NoSuchName$	

and

$ChanErr$	_____
ΔFS	
$\text{dom } cstore = CID$	
$report! = NoFreeCids$	

$NameErr$ describes an attempt to use a non-existent file name; $ChanErr$ describes an unsuccessful attempt to obtain a new channel identifier.

These error descriptions should be associated with the operations that can give rise to them; this is accomplished by schema *disjunction*:

$$schema1 \vee schema2$$

This is the schema formed by merging the two schemas' signatures (as for conjunction \wedge) and forming the disjunction of their predicate parts (where, in contrast, \wedge forms their conjunction).

Thus the schemas *read* and *open*, for example, can be redefined to include the error cases:

$$\begin{aligned} read &\hat{=} (readFS \wedge success) \vee CidErr \\ open &\hat{=} (openFS \wedge success) \vee NameErr \vee ChanErr \end{aligned}$$

The other operations may be similarly treated once their error conditions have been defined.

Figures 4.1 and 4.2 give the expansions of *read* and *open*, respectively.

4.4 Summary

The schema approach to the incremental presentation of large system specifications has been illustrated by using it to describe the Unix filestore. This technique has

<i>read</i>	
<i>FS</i>	
<i>FS'</i>	
<i>cid?</i> : <i>CID</i>	
<i>length?</i> : \mathbb{N}	
<i>data!</i> : seq <i>BYTE</i>	
<i>report!</i> : <i>REPORT</i>	
<i>CHAN</i>	
<i>file</i> : <i>FILE</i>	
$(report! = Ok \wedge$	
$cid? \in \text{dom } cstore \wedge$	
$\theta CHAN = cstore(cid?) \wedge$	
$file = fstore(fid) \wedge$	
$fstore' = fstore \wedge$	
$(\exists CHAN' \bullet posn' = posn + \#data! \wedge$	
$fid' = fid \wedge$	
$cstore' = cstore \oplus \{cid? \mapsto \theta CHAN'\} \wedge$	
$nstore' = nstore \wedge$	
$data! = (1 \dots length?) \triangleleft (file \text{ after } posn))$	
$\vee (report! = NoSuchCid \wedge$	
$cid? \notin \text{dom } cstore \wedge$	
$\theta FS' = \theta FS)$	

Figure 4.1: Expansion of *read*

<i>open</i>	
<i>FS</i>	
<i>FS'</i>	
<i>name?</i> : <i>NAME</i>	
<i>cid!</i> : <i>CID</i>	
<i>report!</i> : <i>REPORT</i>	
<i>fid, fid'</i> : <i>FID</i>	
	$(report! = Ok \wedge$ $\quad name? \in \text{dom } nstore \wedge$ $\quad fid = fid' = nstore(name?) \wedge$ $\quad fstore' = fstore \wedge$ $\quad (\exists CHAN' \bullet posn' = 0 \wedge$ $\quad \quad fid' = fid \wedge$ $\quad \quad cstore' = cstore \oplus \{cid! \mapsto \theta CHAN'\}) \wedge$ $\quad nstore' = nstore \wedge$ $\quad cid! \notin \text{dom } cstore)$ $\vee (report! = NoSuchName \wedge$ $\quad name? \notin \text{dom } nstore \wedge$ $\quad \theta FS' = \theta FS)$ $\vee (report! = NoFreeCids \wedge$ $\quad \text{dom } cstore = CID \wedge$ $\quad \theta FS' = \theta FS)$

Figure 4.2: Expansion of *open*

been used elsewhere to present, and reason about, specifications of other large-scale systems [40, 41, 60, 62]. It has also proved useful in presenting the behaviour of systems from a *variety* of points of view, drawing these together by showing how they are related from an ‘Olympian’ point of view.

However, because of the generality of the underlying theory (set theory), and in particular because of the unrestricted nature of the predicates which can be written to characterise operations, there is no *a priori* guarantee that a system specified in this style is implementable, nor is there any ‘automatic’ way of checking even its internal consistency. The best that can be done is to demonstrate a constructive model at a suitably high level of abstraction. Fortunately, the provision of such a model is usually the first step to be taken in the development of an implementation.

This specification technique is not yet a *development method*; it is simply a step on the way to one. In particular, the usual criteria for deciding on correctness of representations and of algorithms have yet to be adapted to this style of presentation.

Once suitable mathematical types have been discovered for the *observations* to be made of a system (i.e. once suitable mathematical theories have been found and decided upon), the *narrative* part of the top-level views of a system is relatively easy to formulate.

Acknowledgements The use of set theory to specify the behaviour of computer systems was first explained to us by Jean-Raymond Abrial. This specification has been developed from the original attempt by Richard Miller. We have benefited from collaboration with many of our colleagues, especially Ib Sørensen, Steve Schumann, Tony Hoare, Ian Hayes, Roger Gimson, and Tim Clement. The continuing financial support of the UK Science and Engineering Research Council is gratefully appreciated.

4.5 Appendix: differences from Unix

4.5.1 File size

There is an upper bound on the size of files; if a file could contain no more than *FileSizeLimit* bytes

$$\mid \quad FileSizeLimit : \mathbb{N}_1$$

then *FILE* would be defined

$$FILE == \{f : \text{seq } BYTE \mid \#f \leq FileSizeLimit\}$$

4.5.2 Directory size

There is an upper bound on the number of files in the storage system (i.e. the number of ‘inodes’ is limited):

$$\mid \quad FileNumberLimit : \mathbb{N}_1$$

SS
$fstore : FID \leftrightarrow FILE$
$\#fstore \leq FileNumberLimit$

4.5.3 Storage medium capacity

The storage medium used to implement the filing system has finite capacity:

$$\mid \quad DeviceCapacity : \mathbb{N}_1$$

We assume *minbytes*

$$\mid \quad minbytes : FILE \rightarrow \mathbb{N}$$

maps a file into the minimum number of bytes required to represent it in the storage system.

$\begin{array}{l} \text{\textit{SS}} \\ \hline fstore : FID \leftrightarrow FILE \\ \hline \#fstore \leq FileNumberLimit \\ DeviceCapacity \geq \sum \llbracket fid : \text{dom } fstore \bullet minbytes(fstore \text{ } fid) \rrbracket \end{array}$
--

Because in the storage system it is possible to represent a file in more than one way (small, large, huge – also, totally zero blocks may or may not be allocated), all that can be said about the system’s capacity is that it must be at least as large as the minimum required to represent the files within it. Similarly, all that can be said of the device-full condition is that it *cannot* occur while the capacity is sufficient for the *maximum* required. We assume *maxbytes*

$$\mid \quad maxbytes : FILE \rightarrow \mathbb{N}$$

maps a file into the maximum number of bytes required to represent it in the storage system. The condition

$$\sum \llbracket fid : \text{dom } fstore'' \bullet maxbytes(fstore'' \text{ } fid) \rrbracket > DeviceCapacity$$

is *necessary* for a device full error (where *fstore''* is the storage system which would have resulted from the attempted operation).

4.5.4 Seek

seek as defined in Unix has several options, which automatically calculate the desired new offset depending, for example, on the file’s current length. These may be described separately.

<i>seekoffset</i>	
<i>SS</i>	
<i>CHAN</i>	
$n? : \mathbb{N}$	
$p? : 0 \dots 5$	
<i>offset?</i> ,	
<i>size</i> : \mathbb{N}	
$size = \#(fstore(fid))$	
$p? = 0 \Rightarrow offset? = n?$	
$p? = 1 \Rightarrow offset? = posn + n?$	
$p? = 2 \Rightarrow offset? = size + n?$	
$p? = 3 \Rightarrow offset? = 512 * n?$	
$p? = 4 \Rightarrow offset? = posn + 512 * n?$	
$p? = 5 \Rightarrow offset? = size + 512 * n?$	

offset? and *size* are now auxiliary components.

The above schema could be combined with the schema for *seek* to give the full definition of the seek system call.

4.5.5 Representation of numbers

The new position of the file is in fact limited by the ability of the computer to represent numbers.

In this and other cases this limitation could be expressed, for example, as:

$$n24bit == 0 \dots 2^{24} - 1$$

Such sets would then be used, where appropriate, instead of \mathbb{N} :

<i>CHAN</i>	
<i>fid</i> : <i>FID</i>	
<i>posn</i> : <i>n24bit</i>	

Chapter 5

CAVIAR: a case study in specification

Bill Flinn and Ib Holm Sørensen

Abstract This chapter describes the specification of a reasonably complex software system. Important features of the Z approach which are highlighted in this chapter include the interleaving of mathematical text with informal prose, the creation of parameterised specifications, and use of the schema calculus to construct descriptions of large systems from simpler components.

This presentation of the CAVIAR specification also makes use of an experimental modularisation facility.

5.1 Introduction

We view a specification as having a twofold purpose: firstly, to give a formal (mathematical) system description which provides a basis from which to construct a design. Such a mathematical description is essential if we are to prove formally that a design meets its specification. Secondly, to give an informal statement of the system's properties, in order that the specification can be validated against the (usually informal) statement of requirements. Thus the Z approach is to construct a specification document which consists of a judicious mix of informal prose with precise mathematical statements. The two parts of the document are complementary in that the informal text can be viewed as commentary for the formal text. It can be consulted to find out what aspects of the real world are being described and how they relate to the informally stated requirements. The formal text on the other hand provides the precise definition of the system and hence can be used to resolve any ambiguities present in the informal text. A beneficial side effect for practitioners writing such documents is

that their understanding of the system in question is helped greatly by the process of constructing both the formal and the informal descriptions.

It is often the case that the process of abstraction used to construct a specification results in structures that are more general than those actually required for the system being considered. It is part of the Z approach to identify and describe such general structures. These descriptions can be placed in a specification library. Particular cases of these general components can then be used later, either as part of the current system or in subsequent projects.

This specification case study develops a number of general systems which are subsequently constrained and combined to form the complete system description.

5.2 The case study

The **C**omputer **A**ided **V**isitor **I**nformation **A**nd **R**etrieval (CAVIAR) system specification resulted from the analysis of a manual system concerned with the recording and retrieval of data about arrangements for visitors and meetings at a large industrial site. Standard Telecommunications Laboratories (UK) sponsored the study in order to investigate the feasibility of converting to a computer-based solution. Of particular concern were the interrelationships of the stored information, the quality of the user interface, and the volume of data that needed to be processed. The customer provided as input to the study an informal requirements document. We attempt to provide in this chapter an outline of the steps involved in development of the eventual formal specification. It is important to stress at the outset that we view the task of constructing such a specification to be an iterative process, involving several attempts at construction of a model for the system interspersed with frequent dialogues with the customer to clarify details that are ambiguous or undefined in the initial requirements document, and frequent redrafting to clarify the structure of the document.

At an early stage in the analysis it became clear that the CAVIAR system consisted of several largely independent subsystems. Each subsystem records important relationships within the complete system and these separate subsystems are themselves related according to some simple rules. Most of the operations to be provided in the user interface can be explained as functions which transform one particular subsystem only, leaving the others invariant. These observations led to the decision first to define the subsystems in isolation and then to describe the complete system by combining the definitions of the subsystems. Once this decision had been taken, it also became clear that each of the individual subsystems, when viewed at an appropriate level of abstraction, was a particular instance of a general structure. From this vantage point it was natural to specify each of the subsystems by ‘refining’ a specification which describes the underlying general system.

The process of analysis as presented here begins with an identification of the sets which appear to be important from the customer’s point of view. Next the relationships between these sets are investigated and a preliminary classification of the subsystems follows. The third phase consists of developing an appropriate general mathematical structure in which to place these subsystems. Various ways of *specialising* (restricting) the general structure are then investigated and particular subsystems are modelled by *instantiation*. Finally the subsystem models are combined.

5.3 Identification of the basic sets

We now present a brief account of the existing system, emphasising the important concepts in *italics*. *Visitors* come to the site to attend *meetings* and/or consult company employees. A visitor may require a *hotel reservation* and/or *transport reservation*. Each meeting is also required to take place in a designated *conference room*, at a certain *time*. A meeting may require the use of a *dining room* for lunch, on a particular date. Booking a dining room requires *lunch information*, including the number of places needed. Each conference room booking requires *session information* about resources required for use in the meeting, e.g. viewgraphs, projectors. The main operations required at the user interface can briefly be described as facilities for booking, changing, and cancelling the use of resources. We list below the sets together with the names that we shall adopt for referring to them.

Set	Name
Meetings	<i>M</i>
Visitors	<i>V</i>
Conference Rooms	<i>CR</i>
Dining Rooms	<i>DR</i>
Lunch Information	<i>LI</i>
Session Information	<i>SI</i>
Hotel Reservation	<i>HR</i>
Transport Reservation	<i>TR</i>

These form our basic sets:

$$[M, V, CR, DR, LI, SI, HR, TR]$$

The informal interpretation of these sets is straightforward, and for the purpose of this specification no further detail is necessary. Note that the question of modelling time remains to be resolved; at this point we simply observe that hotel reservations are made for particular *dates*, transport reservations are made for certain *times* on particular dates, and conference room bookings are made for *sessions* on particular dates. We shall not specify the term *session* further, apart from noting that a date is always associated with a session; it could, for example, denote complete mornings or afternoons, or hourly or half-hourly intervals, depending on the way conference rooms are allocated.

The notion of time and the relationship between the different units of time used within the system can be formalised by asserting the existence of three sets as follows:

$$[Date, Session, Time]$$

together with two total functions:

$$\left| \begin{array}{l} date_of_session : Session \rightarrow Date \\ date_of_time : Time \rightarrow Date \end{array} \right.$$

5.4 The subsystems of CAVIAR

The first approach to a mathematical model stems from the realisation that several of the sets listed above can be viewed as *resources* and other sets can be viewed as

users of those resources. We can identify the following subsystems of CAVIAR in this framework (i.e. Resource–User systems). Observe that in different subsystems the same set may appear in different roles.

System	Resources	Users
<i>CR_M</i>	Conference Rooms	Meetings
<i>DR_M</i>	Dining Rooms	Meetings
<i>M_V</i>	Meetings	Visitors
<i>HR_V</i>	Hotel Reservations	Visitors
<i>TR_V</i>	Transport Reservations	Visitors

Once we have made this mathematical abstraction it seems worthwhile, for the following reasons, to develop a general theory of such resource–user systems:

1. A specification of such a general system would be more useful as part of a *specification library* than a specific instance of such a system. Reusability is much more likely to be achieved by having *generic* specifications available that can be *instantiated* to provide particular systems.
2. Particular subsystems of the general system can be constructed as *special cases* of the general specification in various ways. This will amply repay care and time spent on the general case. Furthermore, such instantiation may well result in a more compact implementation.

5.5 Modules

To facilitate the production of a specification library, we add a simple parameterised module construct to Z. The modularisation feature has been introduced as an experimental extension to Z to help structure the CAVIAR specification, it is not part of ‘standard’ Z. The parameterisation mechanism is similar to that used for parameterised schemas in Z. For more details on the module construct see [16].

Modules appear as (sub-)sections of this chapter with titles consisting of ‘**Module:**’ followed by the name of the module, optionally followed by a list of generic set formal parameters in square brackets. For example, the header of Section 5.6 below introduces module *Resource_User* with formal parameter sets *T*, *R*, and *U*.

Within a module, a formal parameter set acts like a basic type in a normal Z specification. The scope of variables and schemas defined within a module is limited to the module. However, a specification (or another module) may gain access to the definitions within a module by instantiating the module.

When a module is instantiated, actual parameter sets are specified. For example, the *Resource_User* module of Section 5.6 can be instantiated with the sets *Date*, *HR*, and *V* by the declaration

$$\textit{Resource_User}[\textit{Date}, \textit{HR}, \textit{V}]$$

All the definitions of the module *Resource_User* are available within the specification that included the instantiation. However, every occurrence of a formal parameter within the module definition is replaced by the corresponding actual parameter set within the instantiation.

When a module is instantiated its definitions may also be decorated to distinguish definitions with the same name in different modules. This is discussed in Section 5.11.

5.6 Module: *Resource_User*[*T*, *R*, *U*]

We consider a system parameterised over three sets: *T*, *R*, *U*. Informally, *T* is to be thought of as a set of *time slots*, *R* is a set of *resources*, and *U* is a set of *users*. We describe a general resource–user system as a function from *T* to the set of relations between *R* and *U*. Thus we have a rather general framework: for each time slot $t \in T$, some users are occupying or using some resources. The set *T* will later be instantiated with different sets in the various applications. Notice that considering *relations* between *R* and *U* allows us the possibility of a user occupying several different resources simultaneously, as is shown informally in Figure 5.1.

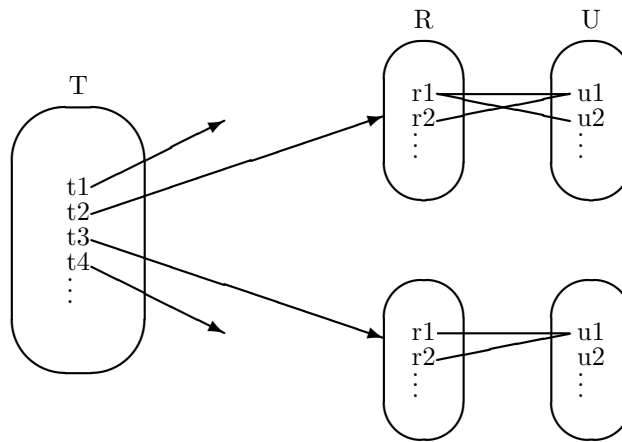


Figure 5.1: General resource–user system

Formally, the structure we are describing is captured by a function *ru* of type

$$T \rightarrow (R \leftrightarrow U)$$

We shall now incorporate this into a *schema definition*. This schema contains some ancillary components in addition to the function *ru* above which are useful in later analysis. In Z specifications it is common to introduce such derived components: as specifiers of software we are neither in the position of a pure mathematician looking for a particularly sparse set of definitions and axioms with which to define a mathematical structure, nor are we in the position of an implementor trying to minimise storage. The component *in_use*, which gives the set of resources in use at any instant, is useful in contexts where we are not concerned with the user component of the system state. The relation *users*, which gives the users occupying resources at any instant, is used in situations where we do not require the information about resources. We also note that there may be occasions when we wish to consider the set of *inverse* relations generated by *ru*; we call this function *ur*.

R_U	$ru : T \rightarrow (R \leftrightarrow U)$ $in_use : T \leftrightarrow R$ $users : T \leftrightarrow U$ $ur : T \rightarrow (U \leftrightarrow R)$
$\forall t : T \bullet$	$in_use(\{t\}) = \text{dom}(ru(t)) \wedge$ $users(\{t\}) = \text{ran}(ru(t)) \wedge$ $ur(t) = (ru(t))^\sim$

The *initial state* of this system is defined by making $ru(t)$ the empty relation for each t :

$$R_U_Init \triangleq [R_U \mid \text{ran}(ru) = \{\{\}\}]$$

Our first theorem asserts that such an initial state is reasonable and assures us of the consistency of the definition of R_U .

Theorem 5.6.1

$$\vdash \exists R_U \bullet R_U_Init$$

The proof of this theorem is straightforward because the components in_use , $users$ and ur are derived from ru and well defined for any value of ru . In particular, if the range of ru contains only the empty set, then the relations in_use and $users$ are empty and the range of ur contains only the empty set. In the interests of readability we have not given proofs of theorems stated in this chapter.

We continue by defining the appropriate operations for this structure. The first step is to identify *commonalities*. For our purposes, the operations that we wish to consider on this structure are concerned with making a new booking, i.e. adding a new pair $(r \mapsto u)$ to an existing relation at some time t , cancelling an existing booking, i.e. removing such an $(r \mapsto u)$ pair, or modifying in some other way the relation that exists at some particular time. In fact we shall be a little more general and define a class of operations on R_U which allows the image of a *set* of time values to be altered. This is because we anticipate such operations as booking a conference room for a meeting which lasts for several time slots. Of course a booking which involves only a single time slot is a special case.

Thus we may summarise the common part of all the operations as follows. Their description involves: a state before, R_U , which introduces ru , in_use , $users$ and ur ; a state after, R_U' , which introduces ru' , in_use' , $users'$, and ur' ; a set of time values, $t?$, which denotes an input. The operations always leave the function ru unchanged except for times in $t?$. Formally this is captured by the following:

ΔR_U	R_U R_U' $t? : \mathbb{P} T$
	$t? \triangleleft ru' = t? \triangleleft ru$

We now have a successful *booking* operation defined as follows:

$ \begin{array}{l} R_U_Book \\ \hline \Delta R_U \\ r? : R \\ u? : U \\ \hline \forall t : t? \bullet \\ (r? \mapsto u?) \notin ru(t) \wedge \\ ru'(t) = ru(t) \cup \{r? \mapsto u?\} \end{array} $	
--	--

Thus R_U_Book inherits all the properties of ΔR_U . Furthermore, it takes two additional (input) parameters $r? : R$ and $u? : U$, and is constrained by a predicate which imposes a requirement on the input parameters and also further relates the before and after states.

Notice that we are making the predicate

$$\forall t : t? \bullet (r? \mapsto u?) \notin ru(t)$$

a precondition for a successful booking. In fact, we can show that this condition is sufficient for performing a successful booking; that is, if we are in a valid system state with the required input parameters of the correct type available and if furthermore the above condition holds, then there exists a resulting valid system state that is related to the starting state according to the R_U_Book schema. Formally, this is the content of the following result.

Theorem 5.6.2

$$\begin{array}{l}
[R_U; t? : \mathbb{P} T; r? : R; u? : U \mid \forall t : t? \bullet (r? \mapsto u?) \notin ru(t)] \\
\vdash \\
\exists R_U' \bullet R_U_Book
\end{array}$$

A successful *cancellation* operation may be defined as follows:

$ \begin{array}{l} R_U_Cancel \\ \hline \Delta R_U \\ r? : R \\ u? : U \\ \hline \forall t : t? \bullet \\ (r? \mapsto u?) \in ru(t) \wedge \\ ru'(t) = ru(t) \setminus \{r? \mapsto u?\} \end{array} $	
--	--

The precondition for successful cancellation is that the pair $(r? \mapsto u?)$ is related by $ru(t)$ for all time values t in $t?$, i.e. the following theorem holds.

Theorem 5.6.3

$$\begin{array}{l}
[R_U; t? : \mathbb{P} T; r? : R; u? : U \mid \forall t : t? \bullet (r? \mapsto u?) \in ru(t)] \\
\vdash \\
\exists R_U' \bullet R_U_Cancel
\end{array}$$

So far we have only specified successful operations; thus these descriptions are incomplete. We could at this stage define robust operations by introducing appropriate

error recovery machinery. In the interests of simplicity we shall not give a general treatment of errors; however, we shall indicate in Section 5.11.2 how the descriptions of the operations at the user interface may be completed.

We shall define two further operations on this structure. The first involves deleting a resource and all use of that resource at a particular set of times. This is an operation to be treated with caution: see Theorem 5.6.7 below.

$$\begin{array}{c}
 \frac{R_U_Del_Res}{\Delta R_U} \\
 \frac{r? : R}{\forall t : t? \bullet} \\
 \frac{}{t \text{ in_use } r? \wedge} \\
 \frac{}{ru'(t) = \{r?\} \triangleleft ru(t)}
 \end{array}$$

Informally, this operation may be described as follows. Consider, in turn, each element t in $t?$ and the corresponding relation $ru(t)$. All elements $(r? \mapsto u)$ are to be removed from $ru(t)$.

Theorem 5.6.4

$$\begin{array}{c}
 [R_U; t? : \mathbb{P} T; r? : R \mid \forall t : t? \bullet t \text{ in_use } r?] \\
 \vdash \\
 \exists R_U' \bullet R_U_Del_Res
 \end{array}$$

Corresponding to the deletion of a resource there is an operation which, given a user value $u?$, deletes all pairs $(r \mapsto u?)$ from the relations associated with time values in $t?$. This is defined as follows:

$$\begin{array}{c}
 \frac{R_U_Del_User}{\Delta R_U} \\
 \frac{u? : U}{\forall t : t? \bullet} \\
 \frac{}{t \text{ users } u? \wedge} \\
 \frac{}{ru'(t) = ru(t) \triangleright \{u?\}}
 \end{array}$$

Theorem 5.6.5

$$\begin{array}{c}
 [R_U; t? : \mathbb{P} T; u? : U \mid \forall t : t? \bullet t \text{ users } u?] \\
 \vdash \\
 \exists R_U' \bullet R_U_Del_User
 \end{array}$$

So far we have listed theorems that a specifier is obliged to prove, namely the result that the initial state satisfies the required definition (and therefore that the specification of the state is consistent), and in addition the theorems that explicitly give the preconditions for each operation. For the specifications that we shall develop from now on such theorems have been omitted in the interests of brevity.

In addition to these obligatory results, there are other ‘optional’ theorems that are satisfied by the specification, and which often give insight into the structure being developed. Two such results for our system follow.

Theorem 5.6.6

$$R_U_Book \circ R_U_Cancel \vdash ru' = ru$$

Informally, this theorem states that if we make a booking and follow it immediately by a cancellation using the same input parameters, then the state of the system does not change.

Theorem 5.6.7

$$\begin{aligned} & R_U_Del_Res \\ & \vdash \\ & in_use' = in_use \setminus \{t : t? \bullet t \mapsto r?\} \wedge \\ & users' = users \setminus \{t : t?; u : U \mid (ur\ t)(\{u\}) = \{r?\}\} \end{aligned}$$

This theorem makes precise the informal comment made earlier about the need for caution with the $R_U_Del_Res$ operation. It shows that, in addition to modifying the in_use relation, the operation may also affect the $users$ relation for times in $t?$. For t in $t?$, an element $t \mapsto u$ is removed from $users$ exactly when the user u is using only the resource $r?$ at time t . There is a similar result concerning the $R_U_Del_User$ operation.

5.7 Specialisations of the resource–user system

We shall now *specialise* the general resource–user system into particular classes of the system. These specialisations are motivated by the observation that for some of the instances listed earlier, a resource may at any given time be related to only one user, or a user may occupy only one resource, or both.

5.7.1 Module: *Exclusive_Resource*[T, R, U]

The first case we define is the class where each resource may be utilised by at most one user, but each user may occupy several resources. This module is built on top of module *Resource_User* and like *Resource_User* is parameterised with the sets T, R and U . We instantiate *Resource_User* with the parameter sets $[T, R, U]$ of *Exclusive_Resource* passed on:

$$Resource_User[T, R, U]$$

All the definitions within *Resource_User* are now available within module *Exclusive_Resource*.

We denote the state of this system by $R \gg U$ and define it formally by

$$R \gg U \triangleq [R_U \mid \text{ran}(ru) \subseteq R \leftrightarrow U]$$

This constrains the resource–user relation to be a function, i.e. each resource is utilised by at most one user. The symbol ‘ \gg ’ is just a character in the name $R \gg U$; it has been chosen to emphasise that, for exclusive resources, the resource–user relations are functions.

The initial state of this system is given by the same condition as for R_U_Init ; thus we have

$$R \gg U_Init \triangleq R \gg U \wedge R_U_Init$$

All operations are described in terms of the following change of state schema:

$$\begin{aligned}\Delta R \gg U &\triangleq \Delta R_U \wedge R \gg U \wedge R \gg U' \\ \Xi R \gg U &\triangleq [\Delta R \gg U \mid \theta R \gg U' = \theta R \gg U]\end{aligned}$$

The operations on this system may be defined as special cases of the general operations for R_U . We first consider the booking operation:

$$R \gg U_Book \triangleq \Delta R \gg U \wedge [R_U_Book \mid \forall t : t? \bullet \neg (t \text{ in_use } r?)]$$

The qualifying predicate is included to indicate that there is a further precondition for booking a resource in a $R \gg U$ system. We now have two parts to the precondition for this operation; firstly this qualifying predicate, and secondly the precondition arising from R_U_Book . In fact the former implies the latter, as is easily checked.

The cancellation operation is defined as follows:

$$R \gg U_Cancel \triangleq \Delta R \gg U \wedge R_U_Cancel$$

On considering the two deletion operations defined for R_U , we observe that $R_U_Del_Res$ is almost equivalent to a cancellation in our present context, because the resource is associated with only one user. However, it is convenient to retain the operation $R_U_Del_Res$ because it does not require the user as an input: it determines the user from the resource.

$$\begin{aligned}R \gg U_Del_Res &\triangleq \Delta R \gg U \wedge R_U_Del_Res \\ R \gg U_Del_User &\triangleq \Delta R \gg U \wedge R_U_Del_User\end{aligned}$$

5.7.2 Module: *Sole_Resource*[T, R, U]

The second case we define is the class where each user may occupy at most one resource but resources may be shared amongst users. This system is also built from the general resource–user system:

$$Resource_User[T, R, U]$$

We denote the state of this system by $R \ll U$ and define it formally by

$$R \ll U \triangleq [R_U \mid \text{ran}(ur) \subseteq U \leftrightarrow R]$$

The initial state of this system is given by the predicate for R_U_Init :

$$R \ll U_Init \triangleq R \ll U \wedge R_U_Init$$

The operations are described in terms of the following schema:

$$\begin{aligned}\Delta R \ll U &\triangleq \Delta R_U \wedge R \ll U \wedge R \ll U' \\ \Xi R \ll U &\triangleq [\Delta R \ll U \mid \theta R \ll U' = \theta R \ll U]\end{aligned}$$

We now define the booking operation for the system:

$$R \ll U_Book \triangleq \Delta R \ll U \wedge [R_U_Book \mid \forall t : t? \bullet \neg (t \text{ users } u?)]$$

As before, a qualifying predicate is included and again the constraint given here implies the earlier precondition for the general R_U_Book operation.

The cancellation operation is defined as follows:

$$R \ll U_Cancel \triangleq \Delta R \ll U \wedge R_U_Cancel$$

On considering the two deletion operations defined for R_U , we observe that this time $R_U_Del_User$ is almost equivalent to a cancellation in our present context, because a user may be associated with only one resource. As before, we retain it as a separate operation because it does not require the resource as input:

$$\begin{aligned} R \ll U_Del_Res &\triangleq \Delta R \ll U \wedge R_U_Del_Res \\ R \ll U_Del_User &\triangleq \Delta R \ll U \wedge R_U_Del_User \end{aligned}$$

5.7.3 Module: *Sole_Exclusive_Resource*[T, R, U]

The third and last specialisation we define shares all of the properties of the systems defined in the preceding two sections. We instantiate the previous two modules and use their definitions to build the sole exclusive resource module:

$$\begin{aligned} &Exclusive_Resource[T, R, U] \\ &Sole_Resource[T, R, U] \end{aligned}$$

The state of this system is therefore defined as the *conjunction* of the two states above. In this system each user may occupy at most one resource and each resource may be occupied by at most one user. Formally we have

$$R \equiv U \triangleq R \gg U \wedge R \ll U$$

The initial state of this system is defined by

$$R \equiv U_Init \triangleq R \equiv U \wedge R_U_Init$$

The operations of this system are given by the conjunction of the operations defined for each of the two earlier systems. For this system we require only the booking and cancellation operations.

$$\begin{aligned} R \equiv U_Book &\triangleq R \gg U_Book \wedge R \ll U_Book \\ R \equiv U_Cancel &\triangleq R \gg U_Cancel \wedge R \ll U_Cancel \end{aligned}$$

5.7.4 The specification library

We have now constructed four specifications which might be considered to form the nucleus of a specification library for resource-user systems. We may summarise the relationships between the four classes of system schematically in Figure 5.2.

5.8 Classification and instantiation

5.8.1 Some laws for CAVIAR

In this section, in order to illustrate the clarification process that took place during requirements analysis, we list some observations about the CAVIAR system which emerged during dialogue with the customer. We formalise the important constraints as *laws* which need to be taken into account in the development that follows:

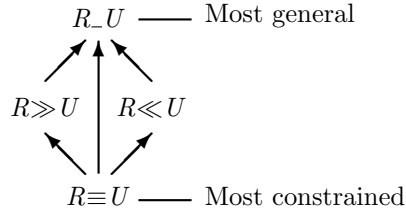


Figure 5.2: Relationships between resource-user systems.

1. At any time a conference room is associated with only one meeting.
2. At any time a meeting may be associated with *more than one* conference room.

Law 1 is reasonably obvious: it would be difficult to hold more than one meeting in a given room. Law 2 is not obvious: it was unclear from the informal description whether or not a meeting could occupy more than one room. In fact the customer believed initially that a meeting could only take up one room, but a counter-example was found amongst the supporting documentation.

3. At any time a meeting is associated with only one dining room.
4. At any time participants from several meetings can occupy the same dining room.

These laws followed from the informal information provided that all visitors in a particular meeting would go to lunch in the same dining room. It was further established that all seats in a dining room were treated as indistinguishable, so further meetings could be accommodated if enough seats were available. Further clarification was necessary regarding lunch times: it transpired that there were 'early' and 'late' lunches; however, this was handled by 'doubling up' each dining room. For example, a booking would be made for 'DR 1, early' and this was a different dining room from 'DR 1, late'.

5. At any time a visitor is associated with only one meeting.
6. At any time a meeting may involve several visitors.

Law 5 had to be checked out with the customer.

7. At any time a hotel room is associated with only one visitor and vice versa.
8. At any time a transport reservation is associated with only one visitor and vice versa.

Law 7 was natural, but law 8 was less so. It was established that even if the transport department decided to use a minibus, a separate transport reservation would be issued to each visitor.

5.8.2 Matching systems with models

In this section we first consider each CAVIAR subsystem in turn and match it to the appropriate model. In fact we have enough structure available to define two subsystems directly and we do this in the remainder of this section.

1. We first consider the conference room/meeting system CR_M .

From laws 1 and 2 we see that CR_M is an instance of the $R \gg U$ subsystem.

2. The dining room/meeting subsystem DR_M .

Applying laws 3 and 4 we find that DR_M is an instance of $R \ll U$.

However, this system does not contain any information about numbers of seats or the lunch details, so we will need to extend this system later.

3. The meeting/visitor subsystem M_V .

From laws 5 and 6 M_V is an instance of $R \ll U$.

However, we have not documented the fact that meetings have to be created before visitors can be attached to them; this will also be done later.

4. The hotel reservation/visitor subsystem HR_V , and the transport reservation/visitor subsystem TR_V , both have the property that each resource is occupied by only one user and vice versa. Therefore both these systems are instances of $R \equiv U$.

In fact this model is sufficient to define HR_V and TR_V completely, by *instantiation*, as we now show.

5.8.3 Module: *HotelReservation*

The hotel reservation system is an instance of a sole, exclusive resource allocator for booking hotel rooms (HR) for visitors (V) in time units of days ($Date$). The hotel reservation module is not itself parameterised, but instantiates the parameterised module for sole, exclusive resources with the sets $Date$, HR , and V introduced in Section 5.3:

$$Sole_Exclusive_Resource[Date, HR, V]$$

This instantiation provides us with all the definitions made in the that module. However, the sets T , R , and U used in the definitions there have been replaced by $Date$, HR , and V , respectively, in the above instantiation. Instantiating the parameters once on the inclusion of the module, rather than having the definitions parametrised individually, guarantees that its definitions are instantiated consistently.

The state of this system is an instance of the $R \equiv U$ schema:

$$Hotel_State \hat{=} R \equiv U$$

Here is an expansion of $Hotel_State$ in which each occurrence of the parameterised sets is instantiated accordingly.

<i>Hotel_State</i>	_____
$ru : Date \rightarrow (HR \leftrightarrow V)$	
$in_use : Date \leftrightarrow HR$	
$users : Date \leftrightarrow V$	
$ur : Date \rightarrow (V \leftrightarrow HR)$	
$\text{ran}(ru) \subseteq HR \leftrightarrow V \wedge$	
$\text{ran}(ur) \subseteq V \leftrightarrow HR \wedge$	
$(\forall t : Date \bullet$	
$in_use(\{t\}) = \text{dom}(ru(t)) \wedge$	
$users(\{t\}) = \text{ran}(ru(t)) \wedge$	
$ur(t) = (ru(t))^\sim$	

The initial state of the hotel reservation subsystem is given by

$$Hotel_Init \hat{=} R \equiv U_Init$$

and the operations are given by

$$\begin{aligned} Book_Hotel_Room_0 &\hat{=} R \equiv U_Book \\ Cancel_Hotel_Room_0 &\hat{=} R \equiv U_Cancel \end{aligned}$$

5.8.4 Module: *Transport_Reservation*

This subsystem is essentially the same as the *HR_V* subsystem except for the parameters. The instances of the parameters are denoted respectively *Time*, *TR*, and *V*, where once again the sets *TR* and *V* are as in Section 5.3. We shall not specify the set *Time* further, except to repeat that each time is associated with a unique *Date* (see Section 5.3):

$$Sole_Exclusive_Resource[Time, TR, V]$$

The state of the transport reservation subsystem includes component *users_D*, which is similar to *users* except that it gives the users for a particular date rather than a time. As every time is associated with a unique date this component is easily derived from *users*.

<i>Transport_State</i>	_____
$R \equiv U$	
$users_D : Date \leftrightarrow V$	
$users_D = date_of_time^\sim \circ users$	

The initial state is given by

$$Transport_Init \hat{=} Transport_State \wedge R \equiv U_Init$$

and operations given by

$$\begin{aligned} Book_Transport_0 &\hat{=} \Delta Transport_State \wedge R \equiv U_Book \\ Cancel_Transport_0 &\hat{=} \Delta Transport_State \wedge R \equiv U_Cancel \end{aligned}$$

5.9 The meeting attendance subsystem

We now turn our attention to what is necessary in order to complete a model for the meeting/visitor (M_V) subsystem. Booking and cancelling operations have been defined already but so far we have not taken account of the fact that before bookings can be made the meeting itself has to have been ‘created’. The question of exactly which objects are ‘currently defined’ at any particular time is important because in several cases only those objects known to the system (i.e. those objects that have been created but not yet destroyed) can book resources, etc.

5.9.1 Module: *Resource_Pool*[T, X]

We can model this situation with a simple structure, which we term a resource pool. This system is parameterised over the set T and an arbitrary set X . There are only two operations to be defined; namely those that add an object to, and delete an object from, the pool, over a specified time period. Formally we have:

$$\boxed{\begin{array}{l} \textit{Pool} \\ \textit{exists} : T \leftrightarrow X \end{array}}$$

with initial state given by

$$\textit{Pool_Init} \triangleq [\textit{Pool} \mid \textit{exists} = \{\}]$$

For later use we define

$$\Delta\textit{Pool} \triangleq \textit{Pool} \wedge \textit{Pool}'$$

and

$$\Xi\textit{Pool} \triangleq [\Delta\textit{Pool} \mid \theta\textit{Pool}' = \theta\textit{Pool}]$$

The operations to add and delete objects follow:

$$\boxed{\begin{array}{l} \textit{Create} \\ \Delta\textit{Pool} \\ t? : \mathbb{P} T \\ x? : X \\ \hline \textit{exists}' = \textit{exists} \cup \{t : t? \bullet t \mapsto x?\} \end{array}}$$

$$\boxed{\begin{array}{l} \textit{Destroy} \\ \Delta\textit{Pool} \\ t? : \mathbb{P} T \\ x? : X \\ \hline \textit{exists}' = \textit{exists} \setminus \{t : t? \bullet t \mapsto x?\} \end{array}}$$

We could have included in the *Create* operation the precondition that the object $x?$ must not already exist for any of the times in $t?$. However, we make a deliberate decision here to omit this – having in mind the situation where an object may already exist for some of the times in $t?$ and its existence needs to be extended to all of $t?$. A similar remark applies to the *Destroy* operation.

5.9.2 Module: *Meeting_Visitor*

To construct the model for the M_V system we combine the resource pool and sole resource structures (with the parameter sets as shown):

$$\begin{aligned} &Sole_Resource[Session, M, V] \\ &Resource_Pool[Session, M] \end{aligned}$$

The state of the meeting/visitor subsystem is a combination of an instance of a sole resource state and a resource pool state. We add two additional components *users_D* and *exists_D*, which are similar to *users* and *exists* except that they are based on dates rather than sessions.

$\begin{aligned} &Meeting_State \\ &R \ll U \\ &Pool \\ &users_D : Date \leftrightarrow V \\ &exists_D : Date \leftrightarrow M \end{aligned}$
$\begin{aligned} &in_use \subseteq exists \\ &users_D = date_of_session \sim \circ users \\ &exists_D = date_of_session \sim \circ exists \end{aligned}$

The first predicate ensures that visitors can only attend existing meetings.
The initial state is given by

$$Meeting_Init \triangleq Meeting_State \wedge R \ll U_Init \wedge Pool_Init$$

We define the operations on *State* in terms of

$$\Delta Meeting_State \triangleq Meeting_State \wedge Meeting_State'$$

The first operation is concerned with adding a visitor to a meeting:

$$\begin{aligned} Add_Visitor_to_Meeting_0 &\triangleq \\ &\Delta Meeting_State \wedge \exists Pool \wedge R \ll U_Book \end{aligned}$$

When an operation is ‘promoted’ in this way, its new precondition is determined as follows: the ‘old’ precondition (i.e. that arising from its definition) must be conjoined with a further predicate which arises from the new invariant of the larger state. Here, for example, the precondition for the earlier booking operation is given in Section 5.7.2, namely

$$\forall t : t? \bullet \neg (t \text{ users } u?)$$

and this must be conjoined with

$$\forall t : t? \bullet t \text{ exists } r?$$

This second predicate is a consequence of the state invariant, which requires all resources that are in use to exist.

Thus the complete precondition for the *Add_Visitor_to_Meeting* operation is given by

$$\forall t : t? \bullet \neg (t \text{ users } u?) \wedge t \text{ exists } r?$$

which states that the visitor ($u?$) is not already attending a meeting at that time and that the meeting he is going to attend actually exists.

The second operation removes a visitor from a meeting:

$$\text{Remove_Visitor_from_Meeting}_0 \hat{=} \Delta\text{Meeting_State} \wedge \Xi\text{Pool} \wedge R \ll U_Cancel$$

It is easy to check that the precondition for this operation is simply inherited from the initial R_U_Cancel operation, namely

$$\forall t : t? \bullet (r? \mapsto u?) \in ru(t)$$

We now define the operations that create and cancel meetings as follows:

$$\text{Create_Meeting}_0 \hat{=} \Delta\text{Meeting_State} \wedge \Xi R \ll U \wedge \text{Create}$$

For the creation there is no precondition.

$$\boxed{\begin{array}{l} \text{Cancel_Meeting}_0 \\ \hline \Delta\text{Meeting_State} \\ R \ll U_Del_Res \\ \text{Destroy}[r?/x?] \end{array}}$$

We identify the resource being deleted ($r?$) with the pool entry being destroyed ($x?$) by renaming $x?$ to $r?$ within the inclusion of Destroy in the cancel meeting operation.

The precondition for cancelling a meeting arises from the $R_U_Del_Res$ operation, i.e. that

$$\forall t : t? \bullet t \text{ in_use } r?$$

5.10 The meeting resource subsystems

We are left with the task of defining the systems CR_M and DR_M . We observe that both of these have further information associated with the resource–user relationship; so in order to capture this facet in our model, we introduce the concept of a *diary* system.

5.10.1 Module: $\text{Diary_System}[T, X, IX]$

The diary is required to record information over time, T , about some elements of a set, X . The associated information is from the set IX . The function recording the information is named *info*. We also define a derived function *info_1*, which is used to simplify later predicates. A relation *recorded* is defined to specify elements of X for which information is recorded.

$$\boxed{\begin{array}{l} \text{Diary} \\ \hline \text{info} : T \rightarrow (X \leftrightarrow IX) \\ \text{info}_1 : (T \times X) \leftrightarrow IX \\ \text{recorded} : T \leftrightarrow X \\ \hline \forall t : T; x : X; i : IX \bullet \\ \quad (t, x) \mapsto i \in \text{info}_1 \Leftrightarrow x \mapsto i \in \text{info}(t) \wedge \\ \quad \text{recorded} = \text{dom}(\text{info}_1) \end{array}}$$

The initial state is given by

$$Diary_Init \triangleq [Diary \mid \text{ran}(info) = \{\{\}\}]$$

The two operations to be defined both involve a change over a particular time period. Note that we are motivated to make this definition in order to maintain compatibility with existing systems. Formally we define the following:

$\frac{\Delta Diary \quad \text{Diary} \quad \text{Diary}' \quad t? : \mathbb{P} T}{t? \triangleleft info' = t? \triangleleft info}$
--

$\frac{\text{Add} \quad \Delta Diary \quad x? : X \quad i? : IX}{\forall t : t? \bullet \neg (t \text{ recorded } x?) \wedge info'(t) = info(t) \cup \{x? \mapsto i?\}}$
--

The complementary erasure operation should remove one element (and the information associated with it) from $info(t)$. However, we note that this is a special case of the following more powerful operation:

$\frac{\text{Erase} \quad \Delta Diary \quad x? : T \leftrightarrow X}{t? = \text{dom}(x?) \wedge x? \subseteq \text{recorded} \wedge info_1' = x? \triangleleft info_1}$

5.10.2 Module: *Conference_Room_Booking*

We are now in a position to specify fully the conference room booking subsystem, by instantiation, as follows:

$$\begin{aligned} & Exclusive_Resource[Session, CR, M] \\ & Diary_System[Session, CR, SI] \end{aligned}$$

The state includes derived components in_use_D and $users_D$ similar to those introduced earlier.

$Conference_State$ $R \gg U$ $Diary$ $in_use_D : Date \leftrightarrow CR$ $users_D : Date \leftrightarrow M$
$in_use = recorded$ $in_use_D = date_of_session \sim \circ in_use$ $users_D = date_of_session \sim \circ users$

The initial state is given by

$$Conference_Init \hat{=} Conference_State \wedge R \gg U_Init \wedge Diary_Init$$

It would be more correct to regard the session information SI as being related to a meeting rather than a conference room. The reason for associating SI with conference rooms is that it contains information that is issued to the department supplying equipment for meetings, and they are concerned with the *venue* rather than what is to take place there.

The operations use the following state change schema:

$$\Delta Conference_State \hat{=} Conference_State \wedge Conference_State'$$

The operations that we require for conference room booking are given below. Information is recorded about each resource when it is booked, and must be erased when a cancellation takes place.

$Book_Conf_Room_0$ $\Delta Conference_State$ $R \gg U_Book$ $Add[r?/x?]$
--

Information is added to the diary for the conference room. Hence, the input to Add , $x?$, is renamed to be the same as the input to the resource booking operation, $r?$.

$Cancel_Conf_Rooms_0$ $\Delta Conference_State$ $R \gg U_Del_User$ $Erase$
$x? = \{t : t?; r : CR \mid u? \mapsto r \in (ur\ t)\}$

The cancellation operation here deletes all conference rooms associated with a particular meeting over the specified time period. This is the operation which is most compatible with the *Cancel_Meeting* operation defined for M_V . However, if required, we could also define the operation that cancels just one conference room and meeting pairing.

The input $x?$ to $Cancel_Conf_Rooms_0$ is auxiliary in the sense that it is determined from $t?$ and the state, so we hide it:

$$Cancel_Conf_Rooms_1 \hat{=} Cancel_Conf_Rooms_0 \setminus (x?)$$

5.10.3 Module: *Dining_Room_Booking*

The final resource subsystem that we need to consider is DR_M . A meeting has a *sole* dining room allocated to it for a given date, but multiple meetings may share a common dining room. Hence, we use an instance of module *Sole_Resource* to keep track of dining room bookings:

$$Sole_Resource[Date, DR, M]$$

In addition, luncheon information is recorded in a diary for each meeting:

$$Diary_System[Date, M, LI]$$

The analysis so far does not take account of the fact that dining rooms have a finite capacity, so we need to extend our model. We suppose that we have been given a bag

$$| \quad max_cap : \text{bag } DR$$

that records the maximum capacity of each dining room and we record the number of seats in each dining room which have been reserved already. (See Appendix A.11 for more details about operations on bags.)

The DR_M system is defined formally as follows:

$ \begin{array}{l} \textit{Dining_State} \\ R \ll U \\ \textit{Diary} \\ rsvd : Date \rightarrow (\text{bag } DR) \\ \hline users = recorded \wedge \\ (\forall t : Date \bullet rsvd(t) \sqsubseteq max_cap \wedge \\ (\forall r : DR \bullet (t \textit{ in_use } r) \Rightarrow (rsvd\ t) \# r \neq 0)) \end{array} $
--

Observe that in this case information is associated with each *user*, and therefore the diary system takes M as its main parameter. Dining rooms that are in use have a number of seats reserved, and this number has to be within the dining room's capacity: the dining seats in use at a particular time t , $rsvd(t)$, must be a sub-bag of the maximum capacity of the dining rooms, max_cap . If a dining room r is in use at a particular time t , then it should have some seats reserved.

The initial state of the dining room booking subsystem is given by

$$Dining_Init \hat{=} Dining_State \wedge R \ll U_Init \wedge Diary_Init$$

The two operations that we require for this structure are *booking* a (number of seats in a) dining room and *cancelling* a lunch booking for a particular meeting. In normal circumstances, a resource (dining room) will not be subject to being taken out of service (although clearly this occurrence is easy to model if required).

Both of these operations leave $rsvd$ unchanged for time values outside the period in question; we make this part of the operation invariant.

$\Delta Dining_State$ $Dining_State$ $Dining_State'$ $\Delta R \ll U$ $\Delta Diary$ $amount? : Date \leftrightarrow \mathbb{N}_1$
$dom(amount?) = t? \wedge$ $t? \triangleleft rsvd' = t? \triangleleft rsvd$

To book a dining room there should be sufficient capacity at all the times requested. The requested number of places is reserved for the each of the dates. ($B \uplus C$ is the bag in which the frequency of each element is the sum of its frequencies in B and C .) Luncheon information is added to the diary for the meeting.

$Book_Dining_Room_0$ $\Delta Dining_State$ $R \ll U_Book$ $Add[u?/x?]$
$(\forall t : t? \bullet$ $\quad rsvd(t) \# r? + amount?(t) \leq max_cap \# r? \wedge$ $\quad rsvd'(t) = rsvd(t) \uplus \{r? \mapsto amount?(t)\})$

All the dining rooms for a meeting can be cancelled by the following operation. The dining room, dr , in use at a given time by the meeting is uniquely determined.

$Cancel_Dining_Room_0$ $\Delta Dining_State$ $R \ll U_Del_User$ $Erase$
$x? = \{t : t? \bullet t \mapsto u?\} \wedge$ $(\forall t : t? \bullet$ $\quad \mathbf{let} \ dr == ur(t)(u?) \bullet$ $\quad rsvd(t) \# dr \geq amount?(t) \wedge$ $\quad rsvd'(t) = rsvd(t) \ominus \{dr \mapsto amount?(t)\})$

The input $x?$ is auxiliary in the above definition, so we hide it:

$$Cancel_Dining_Room_1 \hat{=} Cancel_Dining_Room_0 \setminus (x?)$$

5.10.4 Module: *Visitor_Pool*

From the informal requirements we find that visitors must be registered before they are allowed to attend meetings or have resources booked on their behalf. This requirement is easily met by introducing a visitor *Pool* structure with actual parameters *Date* and *V*:

$$Resource_Pool[Date, V]$$

$$Visitor_State \hat{=} Pool$$

The initial state given by

$$Visitor_Init \hat{=} Pool_Init$$

The operations that we require for this system are simply those of creation and destruction of visitors. Formally we have

$$\begin{aligned} Create_Visitor_0 &\hat{=} Create \\ Destroy_Visitor_0 &\hat{=} Destroy \end{aligned}$$

5.10.5 The construction process

In this section we summarise the constructions we have used to build the individual CAVIAR components. In Sections 5.9 and 5.10 we added *resource pool* and *diary* components to our basic library of Section 5.7. We now have a library that consists of the six components R_U , $R \gg U$, $R \ll U$, $R \equiv U$, *Resource_Pool* and *Diary_System*. We indicate in Figure 5.3 how each subsystem has been constructed using components from the library.

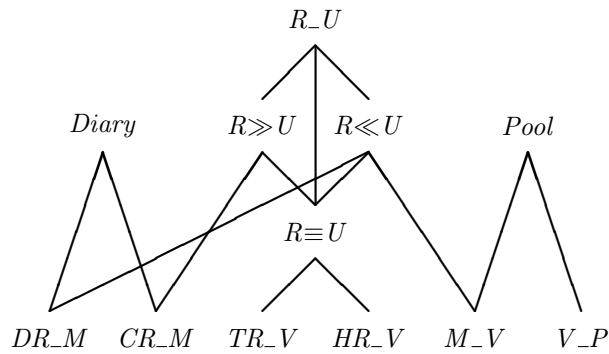


Figure 5.3: Construction from components

5.11 Module: *CAVIAR*

We have now achieved our first goal of specifying all constituent subsystems of CAVIAR. As many of the component subsystems are built from the same library modules, we need to distinguish different uses of the same names by decorating the subsystems as we include them. The decoration consists of an abbreviation indicating the subsystem followed by ‘::’.

HR_V::Hotel_Reservation
TR_V::Transport_Reservation
M_V::Meeting_Visitor
CR_M::Conference_Room_Booking
DR_M::Dining_Room_Booking
V_P::Visitor_Pool

The decoration is applied to all names defined in a subsystem, and in particular when applied to schemas acts as a decoration of the components of the schema. This is necessary to distinguish the same component name occurring in two logically different ways in two different schemas that are to be combined.

We have yet to combine the subsystems into a coherent whole. This is now a comparatively easy task, once we have observed a few extra constraints.

5.11.1 Combining subsystems to form the state

We define the visitor part of the system as follows:

V_SYS
$V_P::Visitor_State$
$HR_V::Hotel_State$
$TR_V::Transport_State$
$HR_V::users \subseteq V_P::exists \wedge$
$TR_V::users_D \subseteq V_P::exists$

The invariant states that visitors that have hotel or transport reservations must be known.

The meeting part of the system is defined by M_SYS .

M_SYS
$M_V::Meeting_State$
$CR_M::Conference_State$
$DR_M::Dining_State$
$CR_M::users \subseteq M_V::exists \wedge$
$DR_M::users \subseteq M_V::exists_D$

The invariant states that meetings which are occupying conference rooms or dining rooms must be known to the system at that time.

These two subsystems are now combined to form the CAVIAR system.

$CAVIAR$
V_SYS
M_SYS
$M_V::users_D \subseteq V_P::exists$

Informally, the invariant states that all visitors who are attending meetings must be known to the system. The initial state of the system is given by the conjunction of all the initialisations:

$$CAVIAR_Init \triangleq HR_V::Hotel_Init \wedge TR_V::Transport_Init \wedge \\ M_V::Meeting_Init \wedge CR_M::Conference_Init \wedge \\ DR_M::Dining_Init \wedge V_P::Visitor_Init$$

It is easy to verify that this conjunction satisfies the invariant.

The operations on CAVIAR may be divided naturally into three groups: those involving meetings only; those involving visitors only; and a general visitor removal operation. They all involve the complete state of CAVIAR:

$$\Delta \text{CAVIAR} \triangleq \text{CAVIAR} \wedge \text{CAVIAR}'$$

5.11.2 Operations that involve meetings only

These operations are concerned with M_SYS only and leave V_SYS unchanged. We denote this by

$$\begin{aligned} \Xi V_SYS &\triangleq [V_SYS; V_SYS' \mid \theta V_SYS = \theta V_SYS'] \\ M_OP &\triangleq \Delta \text{CAVIAR} \wedge \Xi V_SYS \end{aligned}$$

Similar definitions for $CR_M::\Xi \text{Conference_State}$, $DR_M::\Xi \text{Dining_State}$, etc., are assumed in what follows.

The first operation is to construct a meeting:

$$\begin{aligned} \text{Create_Meeting} &\triangleq M_OP \wedge M_V::\text{Create_Meeting_0} \wedge \\ &\quad CR_M::\Xi \text{Conference_State} \wedge DR_M::\Xi \text{Dining_State} \end{aligned}$$

This operation has no precondition, so it is total (there is no precondition for Create_Meeting_0).

The next operation is to cancel a meeting:

$$\begin{aligned} \text{Cancel_Meeting_1} &\triangleq M_OP \wedge M_V::\text{Cancel_Meeting_0} \wedge \\ &\quad CR_M::\Xi \text{Conference_State} \wedge DR_M::\Xi \text{Dining_State} \end{aligned}$$

We can determine the precondition for this operation as follows: first we establish the constraint arising from the system invariant. The operation removes an element from $M_V::\text{exists}$ so this element cannot be a user in CR_M or DR_M during the period $M_V::t?$. Formally, we require that

$$\begin{aligned} \forall t : M_V::t? \bullet \\ (t \mapsto M_V::r?) \notin CR_M::\text{users} \wedge \\ (\text{date_of_session}(t) \mapsto M_V::r?) \notin DR_M::\text{users} \end{aligned}$$

The second part of the precondition is from the precondition for Cancel_Meeting_0 . This is precisely

$$(\forall t : M_V::t? \bullet (t \mapsto M_V::r?) \in M_V::\text{in_use})$$

We shall at this point fulfil the promise made in Section 5.6, namely indicating how to define the corresponding total operation. This is formed by the *disjunction* of the successful operation with the schema which takes as its qualifying predicate the *negation* of the precondition established above.

$\begin{aligned} &\text{Cancel_Meeting_Fail} \\ &\Xi \text{CAVIAR} \\ &M_V::t? : \mathbb{P} \text{Session} \\ &M_V::r? : M \end{aligned}$
$\begin{aligned} &(\exists t : M_V::t? \bullet \\ &\quad (t \mapsto M_V::r?) \in CR_M::\text{users} \vee \\ &\quad (\text{date_of_session}(t) \mapsto M_V::r?) \in DR_M::\text{users} \vee \\ &\quad (t \mapsto M_V::r?) \notin M_V::\text{in_use}) \end{aligned}$

$$\text{Cancel_Meeting} \hat{=} \text{Cancel_Meeting_1} \vee \text{Cancel_Meeting_Fail}$$

Informally, if the required precondition for the meeting cancellation is not satisfied, the system is unchanged. In practice, we would require an appropriate error message to be output.

For the sake of brevity, we shall present the remainder of the operations without going through this process.

The next two operations add visitors to, and delete visitors from, a meeting:

$$\begin{aligned} \text{Add_Visitor_to_Meeting} &\hat{=} M_OP \wedge \\ &M_V::\text{Add_Visitor_to_Meeting_0} \wedge \\ &CR_M::\exists \text{Conference_State} \wedge DR_M::\exists \text{Dining_State} \\ \text{Remove_Visitor_from_Meeting} &\hat{=} M_OP \wedge \\ &M_V::\text{Remove_Visitor_from_Meeting_0} \wedge \\ &CR_M::\exists \text{Conference_State} \wedge DR_M::\exists \text{Dining_State} \end{aligned}$$

The preconditions for these operations are straightforward to determine in the usual way, and we shall omit them and those for the remaining operations also.

The next two operations deal with conference rooms:

$$\begin{aligned} \text{Book_Conf_Room} &\hat{=} M_OP \wedge M_V::\exists \text{Meeting_State} \wedge \\ &CR_M::\text{Book_Conf_Room_0} \wedge DR_M::\exists \text{Dining_State} \\ \text{Cancel_Conf_Rooms} &\hat{=} M_OP \wedge M_V::\exists \text{Meeting_State} \wedge \\ &CR_M::\text{Cancel_Conf_Rooms_1} \wedge DR_M::\exists \text{Dining_State} \end{aligned}$$

We now have the two operations concerning dining rooms:

$$\begin{aligned} \text{Book_Dining_Room} &\hat{=} M_OP \wedge M_V::\exists \text{Meeting_State} \wedge \\ &CR_M::\exists \text{Conference_State} \wedge \\ &DR_M::\text{Book_Dining_Room_0} \\ \text{Cancel_Dining_Room} &\hat{=} M_OP \wedge M_V::\exists \text{Meeting_State} \wedge \\ &CR_M::\exists \text{Conference_State} \wedge \\ &DR_M::\text{Cancel_Dining_Room_1} \end{aligned}$$

There is one final operation to be defined in this section: namely the cancellation of both dining room and conference room(s) associated with a particular meeting. This is *not* the conjunct of the two cancellation operations already given because each of these leaves the components it is not acting on *fixed*. Hence we need a different operation defined by

$\begin{aligned} &\text{Cancel_Meeting_Arrangements_0} \\ &M_OP \\ &\text{meeting?} : M \\ &M_V::\exists \text{Meeting_State} \\ &CR_M::\text{Cancel_Conf_Rooms_1} \\ &DR_M::\text{Cancel_Dining_Room_1} \end{aligned}$
$\begin{aligned} &\text{meeting?} = CR_M::u? = DR_M::u? \\ &CR_M::t? = \{s : \text{Session} \mid (s \mapsto \text{meeting?}) \in CR_M::\text{users}\} \\ &DR_M::t? = \{d : \text{Date} \mid (DR_M::r? \mapsto \text{meeting?}) \in DR_M::ru(d)\} \end{aligned}$

The components $CR_M::u?$, $CR_M::t?$, $DR_M::u?$, $DR_M::r?$, and $DR_M::t?$ are auxiliary within $Cancel_Meeting_Arrangements_0$, so we hide them:

$$\begin{aligned} Cancel_Meeting_Arrangements_1 &\hat{=} \\ &Cancel_Meeting_Arrangements_0 \setminus (CR_M::u?, CR_M::t?, \\ &DR_M::u?, DR_M::r?, DR_M::t?) \end{aligned}$$

5.11.3 Operations that involve visitors only

This section contains operations which involve V_SYS only and leave M_SYS unchanged. We denote this group by

$$V_OP \hat{=} \Delta CAVIAR \wedge \Xi M_SYS$$

The first pair of operations introduce visitors to and remove visitors from the visitor system:

$$\begin{aligned} Create_Visitor &\hat{=} V_OP \wedge V_P::Create_Visitor_0 \wedge \\ &HR_V::\Xi Hotel_State \wedge TR_V::\Xi Transport_State \\ Destroy_Visitor &\hat{=} V_OP \wedge V_P::Destroy_Visitor_0 \wedge \\ &HR_V::\Xi Hotel_State \wedge TR_V::\Xi Transport_State \end{aligned}$$

The CAVIAR invariant induces a precondition for the $Destroy_Visitor$ operation:

$$\begin{aligned} \forall t : V_P::t? \bullet \\ (t \mapsto V_P::x?) \notin \\ (HR_V::users \cup TR_V::users_D \cup M_V::users_D) \end{aligned}$$

The two operations concerned with hotel rooms are as follows:

$$\begin{aligned} Book_Hotel_Room &\hat{=} V_OP \wedge V_P::\Xi Visitor_State \wedge \\ &HR_V::Book_Hotel_Room_0 \wedge TR_V::\Xi Transport_State \\ Cancel_Hotel_Room &\hat{=} V_OP \wedge V_P::\Xi Visitor_State \wedge \\ &HR_V::Cancel_Hotel_Room_0 \wedge TR_V::\Xi Transport_State \end{aligned}$$

The two operations concerned with transport reservations follow:

$$\begin{aligned} Book_Transport &\hat{=} V_OP \wedge V_P::\Xi Visitor_State \wedge \\ &HR_V::\Xi Hotel_State \wedge Book_Transport_0 \\ Cancel_Transport &\hat{=} V_OP \wedge V_P::\Xi Visitor_State \wedge \\ &HR_V::\Xi Hotel_State \wedge Cancel_Transport_0 \end{aligned}$$

5.11.4 A general visitor removal operation

Finally we define an operation that removes a visitor entirely from the system for a particular set of dates.

$\Delta \text{Delete_Visitor_0}$ $v? : V$ $d? : \mathbb{P} \text{ Date}$ $CR_M :: \exists \text{Conference_State}$ $DR_M :: \exists \text{Dining_State}$ $HR_V :: \text{Cancel_Hotel_Room_0}$ $TR_V :: \text{Cancel_Transport_0}$ $M_V :: \text{Remove_Visitor_from_Meeting_0}$ $V_P :: \text{Destroy_Visitor_0}$
$v? = HR_V::u? = TR_V::u? = M_V::u? = V_P::x? \wedge$ $HR_V::t? = \{d : d? \mid d \mapsto v? \in HR_V::users\} \wedge$ $TR_V::t? = \{t : \text{Time} \mid t \mapsto v? \in TR_V::users \wedge$ $\quad \text{date_of_time}(t) \in d?\} \wedge$ $M_V::t? = \{s : \text{Session} \mid s \mapsto v? \in M_V::users \wedge$ $\quad \text{date_of_session}(s) \in d?\} \wedge$ $V_P::t? = \{d : d? \mid d \mapsto v? \in V_P::exists\}$

Within *Delete_Visitor_0* schema the components $V_P::x?$, $HR_V::u?$, $TR_V::u?$, $M_V::u?$, $HR_V::t?$, $TR_V::t?$, $M_V::t?$, and $V_P::t?$ are auxiliary, so we hide them:

$$\text{Delete_Visitor} \hat{=} \text{Delete_Visitor_0} \setminus \\
(V_P::x?, HR_V::u?, TR_V::u?, M_V::u?, \\
HR_V::t?, TR_V::t?, M_V::t?, V_P::t?)$$

5.12 Conclusion

This specification has created a conceptual model for the CAVIAR system which provides a precise description of the system state and its external interface, together with an exact functional specification of every operation. The subtle inter-relationships between constituent subsystems are described in the predicates that constrain the combination of these subsystems, and these have been taken into account in the specification of the operations. The system designer can now concentrate on the important parts of the design task: namely selecting appropriate data structures and algorithms, without having to be simultaneously concerned with the complexity of subsystem interactions. This reflects the classical principle of *separation of concerns*.

It may be argued that a specification such as we have given above is far from being an actual product. Experience shows, however, that such specifications reduce substantially the effort required to develop executable software. In the case of CAVIAR, a Pascal implementation was constructed directly and quickly, working from the earlier specification.

Acknowledgements A formal specification of CAVIAR was given in 1981 by J.-R. Abrial. This work was carried out at the Programming Research Group at Oxford University in collaboration with Bernard Sufrin, Tim Clement and Ib Holm Sørensen. Tim Clement implemented a prototype version of the specification on an ITT-2020 computer in UCSD Pascal. J.-R. Abrial's original specification document listed most

of the properties of the system that appear in this document, though the style of the presentation, the notation, and the conventions used in this chapter have since been developed by members of the Programming Research Group.

We would like to thank J.-R. Abrial for his original contribution, and Ian Hayes for editing this chapter and developing the experimental modularisation facility. Thanks are also due to all those involved with the project, particularly the personnel in the Visitor Services Department of STL, who willingly provided the team with information about the current manual system in operation at that time.

We would also like to thank Bernie Cohen, Tim Denvir, and Tom Cox for their initial effort in setting up this collaborative effort between STL and the Programming Research Group and their continuing interest.

We would like to thank Brendan Mahony for his assistance in producing the special fonts used within this chapter.

Chapter 6

Towards a formal specification of the ICL Data Dictionary

Bernard Sufrin

Abstract We present a formal specification of the ICL Data Dictionary system, paying particular attention to the facilities it provides for controlling the retrieval and updating of dictionary elements. We conclude by suggesting some modifications to the design which would render the system simpler to understand and to implement whilst retaining its full power. The specification notation Z [46, 61], which is based on set theory, is used, and familiarity with the mathematical notions of predicate, set, relation and function is assumed throughout.

Background The potential benefits of applying formal, or at least mathematically rigorous, methods to the design and production of software are currently a topic of much discussion and have been eloquently expounded elsewhere, for example in [21] and in [33]. In common with many others, we believe that the time is ripe, perhaps even overripe, for the application of these methods in an industrial context. This report arose out of a challenge from ICL to work with a group of their practising programmers to investigate the applicability of the methods to a real commercial product, the ICL Data Dictionary system (henceforward DDS). The company sponsored a ten-day pilot project, during which we used mathematical techniques to investigate two areas of DDS that its designers believe to be difficult to understand and explain, namely the means provided for controlling access to dictionary elements, and the support provided for multiple versions. Here we report on our investigation of access control.

Copyright © ICL Technical Journal, 1984. This paper was first published in the *ICL Technical Journal*, November 1984.

6.1 Introduction

It is by now a matter of common knowledge that a substantial proportion of the total costs of ‘bugs’ discovered during the lifetime of a computer-based system can be attributed to mistakes made during the earliest stages of its development. It is for this reason that we believe that the most appropriate time to construct – and to use – a formal specification for a large system is before the system is built. Why, then, attempt a mathematical specification of parts of a software product that is already several years old? Firstly, the standard DDS documentation of access control is rather difficult to understand, partly because the answers to many questions can only be discovered by reading the complete product documentation in its entirety. A concise formal description of the dictionary system from which the answers to questions about access can *easily* be deduced will be useful in its own right, and perhaps provide a basis for better product documentation. Secondly, a by-product of the specification activity will be the construction of a conceptual framework within which the consequences of simplifying the design of the system can be investigated. Thirdly, even though we hardly expect the majority of DDS users to be able to read a mathematical specification, in order to produce it at all we are forced to ask questions of the implementers of the system which the product documentation presently fails to answer adequately. These questions and their answers will certainly be of use to the authors of subsequent editions of the product documentation. Finally, if used in the spirit that it was made, the specification should also help to ensure compatibility of successive versions and new implementations of the Dictionary.

6.2 Overview of the Data Dictionary System

The potential application areas of the DDS are outlined in detail in the first chapter of its reference manual [31]. In essence it is a database system specially adapted to the needs of supporting the construction, documentation and maintenance of collections of programs which must be kept mutually consistent. Such collections are to be found at computer installations everywhere, and without computer-based support they can quickly become very difficult to manage.

For the purposes of this report it is enough for us to note that the system provides a means of naming, storing, manipulating and enquiring about any number of *elements*, each of which may have several named *properties* which possess *values*. Some properties are possessed only by certain types of element, for example the *TITLE-PAGE property of REPORT-PROGRAM elements. Other properties may be possessed by any or all elements but are never acted upon by DDS, for example the *DESCRIPTION and *NOTE properties, whose values are uninterpreted text. Finally there is a class of properties which are administrative in nature and may be possessed by any or all elements and which *are* interpreted (acted upon) by the DDS, for example the *PRIVACY and *AUTHORITY properties which are possessed by almost all elements. It is by setting administrative properties such as these that an administrator may control aspects of the behaviour of a Data Dictionary at a particular installation, and users may control how the elements they define can be manipulated by others.

In order to begin constructing a mathematical model we must first introduce some nomenclature. Let P denote the set of all possible names for *properties* possessed by elements in the database, and let V denote the set of all possible *values* that these

properties may take. For the moment we need not investigate the internal structure of the sets P and V , though we will do so later.

In the manual, the term ‘element’ means an object consisting of a collection of named properties which have values. Such an object may be modelled as a finite mapping from property names to values. Let E denote the set of all possible elements storable in a DDS database; we define it formally by

$$E == P \multimap V$$

Notice that this definition merely explains the essence of ‘elementhood’, but gives us no clues about how to represent elements using the data structures that are available in a conventional programming language.

If we take some liberties with the way in which we write values, and suppose that *authority*, *description*, *note* and *privacy* are among the possible property names, then we can give a couple of (possibly untypical) examples of elements:

$$\begin{aligned} &\{ \textit{authority} \mapsto \textit{Bernard}, \textit{description} \mapsto \text{‘Formal Documentation’} \} \\ &\{ \textit{authority} \mapsto \textit{Bernard}, \textit{privacy} \mapsto 99, \textit{note} \mapsto \text{‘What’s a note?’} \} \end{aligned}$$

The term ‘element identifier’ means the *name* by which an element is known to the DDS. If we let EI denote the set of *all possible* element identifiers, then the current state of a DDS may be modelled as an object from the set $EI \multimap E$, which can be expanded to $EI \multimap (P \multimap V)$. That is to say, a mapping from element identifiers to elements, which are themselves mappings from property names to storable values. Notice that in order to describe the state of a DDS at this level of abstraction it is not necessary for us to give details of the internal structure of element identifiers, though we will be forced to reveal this structure later.

6.3 Access control

In this section we present a sequence of successively more accurate descriptions of the abstract information structures of a DDS which support access control. We do so without making explicit the fact that the dictionaries are self-describing – in the sense that these information structures are completely described by elements present in the dictionaries themselves. In Section 6.4 we consider the state of a running DDS, and show how some simple commands issued by users affect that state. In Section 6.5 we demonstrate in detail the relationship between our abstract description and the stored elements with which the dictionary implements the structures introduced in the first. Finally, in Section 6.6 we describe several more complex commands, whose effects depend on the details just demonstrated.

6.3.1 Abstract information structures of DDS

Our first approximation is rather simple: we simply observe the set of element names known to the system and the correspondence between these names and their stored values. More formally, we define a schema $DD0$ which characterises the possible states of a DDS.

<i>DD0</i>
$elements : \mathbb{F} EI$ $store : EI \leftrightarrow E$
$elements = \text{dom } store$

The predicate below the bar records the invariant relationship we expect to hold between the two observations: the element names known to the system are exactly those which correspond to elements in the store.

A formalisation at this level of abstraction could serve as the basis of a model for the data stored in *any* entity-attribute database! Since it fails to take into account the *specific* characteristics of DDS that we are trying to explain, we shall discard it.

In our second approximation we record the fact that some elements have owners. These are called *authority elements* in the product documentation, and form the basis for one of the methods by which access to elements is controlled. When an ordinary user runs one of the DDS programs, he or she may do so under the aegis of an authority: access to elements in the dictionary depends, among other things, upon this authority.

We begin to formalise this situation by introducing additional observations, namely the finite set of authority element identifiers which have been introduced by the dictionary administrator into the system, and a mapping from the identifiers of stored elements to those of their owners.

<i>DD1</i>
$store : EI \leftrightarrow E$ $elements : \mathbb{F} EI$ $auth : \mathbb{F} EI$ $owner : EI \leftrightarrow EI$
$elements = \text{dom } store$ $\text{ran } owner \subseteq auth \subseteq elements$ $\text{dom } owner \subseteq elements \setminus auth$

The new predicates below the bar record the additional invariant relationships between our observations: all owners of elements must be authority elements, all authority elements must be present in the store, but no authority element has an owner.

Notice that the last predicate is such that not all elements need an owner; indeed it is consistent with a state in which *no* elements have owners. It turns out that it is easier to explain the system if we invent a special ‘mythical’ authority – the nil authority – and insist that every non-authority element has an owner (which might be the nil authority). We formalise this in two steps: first we introduce a constant element identifier, *nil*, to stand for the identifier of the mythical nil authority.

| $nil : EI$

Next we give a new description of a DDS state, which incorporates the second approximation but strengthens the invariant with two additional predicates: the first states that *nil* is an authority element, and the second that *all* non-authority elements must now have owners.

<i>DD2</i>
<i>DD1</i>
$nil \in auth$ $dom\ owner = elements \setminus auth$

We now engage in a little speculation (with the best of pedagogical motives): if the DDS designers had had an authoritarian or individualistic cast of mind, they might have stopped their design activity at this point and insisted that only the owner of an element can retrieve or update it. Under these circumstances we would have been able to end our modelling activity by making just one more observation of the state of a dictionary: the relation *canaccess*, which can be completely determined by the values of the remaining observations of *DD2*, holds between an authority and an element exactly when that authority would allow access to the element.

<i>AuthoritarianDD</i>
<i>DD2</i>
$canaccess : EI \leftrightarrow EI$
$\forall user : auth; elt : elements \bullet$ $user\ canaccess\ elt \Leftrightarrow owner\ elt = user$

A marginally more libertarian group of designers might have interpreted ownership by the *nil* authority somewhat differently and allowed anybody to retrieve or update such elements:

<i>NotQuiteSoAuthoritarianDD</i>
<i>DD2</i>
$canaccess : EI \leftrightarrow EI$
$\forall user : auth; elt : elements \bullet$ $user\ canaccess\ elt \Leftrightarrow owner\ elt \in \{user, nil\}$

As might be expected the ICL designers wanted to make their system a little more flexible than either of these descriptions indicate and we find both that they have included a number of ways in which elements may be shared between authorities and that they distinguish between retrieval and updating.

An authority may delegate rights to retrieve an element to one or more other authorities. We formalise this by introducing the relation *delegates*, which holds between an authority *a* and an authority *a'* if and only if *a* has taken steps to permit *a'* to retrieve *all* the elements which *a* owns.

<i>DD3</i>
<i>DD2</i>
$delegates : EI \leftrightarrow EI$
$delegates \in (auth \leftrightarrow auth)$

Note that *delegates* cannot be declared to be of type $auth \leftrightarrow auth$ because the scope of *auth* is only the predicate part of the *DD3*.

Rights to a *single* element may be given by its owner to another authority, so we introduce another relation, *mayretrieve*, which holds between an authority *a* and an element *e* only if *e*'s owner has explicitly taken steps to allow *a* to retrieve it.

DD4
DD3
$mayretrieve : EI \leftrightarrow EI$
$dom\ mayretrieve \subseteq auth$
$ran\ mayretrieve \subseteq elements \setminus auth$

As we shall see in the next section, part of the stored description of an authority element is a description of the types of element which it may not update. Since our description is not yet at a level of detail that includes types, we can discuss the right of an authority to *update* an element in the database only in rather general terms. In order to begin the discussion we add a relation *maynotupdate* to our observations. This relation holds between a declared authority and the names of elements which it has explicitly been forbidden to update; these can include elements that are not yet in the store.

DD5
DD4
$maynotupdate : EI \leftrightarrow EI$
$dom\ maynotupdate \subseteq auth$

Note: Readers familiar with DDS will recognise that *delegates* is closely related to the *RETRIEVE property of authority elements, that *mayretrieve* is related to the *RETRIEVE property of non-authority elements, and that *maynotupdate* is related to the *INHIBIT properties of authority elements.

If the designers had stopped here, we would characterise an authority's rights to retrieve and update elements by beginning to define the relation *canretrieve*, which holds between an authority and the elements it is permitted to retrieve, and the relation *canupdate*, which holds between an authority and the elements that the system will allow it to update. At this stage the only thing we can say about the updating is negative: namely that if an authority has explicitly been forbidden to update an element then the system will prevent it from doing so.

DD6
DD5
$canretrieve : EI \leftrightarrow EI$
$canupdate : EI \leftrightarrow EI$
$\forall user : auth; elt : elements \bullet$ $(owner\ elt \in \{user, nil\} \vee$ $(owner\ elt)\ delegates\ user \vee$ $user\ mayretrieve\ elt) \Rightarrow user\ canretrieve\ elt$
$\forall user : auth; elt : elements \bullet$ $((user, elt) \in maynotupdate) \Rightarrow$ $(user\ maynotupdate\ elt) \Rightarrow \neg (user\ canupdate\ elt)$

The above specification does not completely determine the values of *canretrieve* and *canupdate*. These definitions are strengthened in Figure 6.1, where the implications in the above definition are strengthened to equivalences.

Although the system we have described above might have satisfied many designers, it turns out that orthogonal to the system of ownership the DDS has a notion of *levels of privacy*. Stored elements have a privacy level, which is a number between 0 and 99.

$$Level == 0 \dots 99$$

Irrespective of the possibilities for retrieval afforded by the ownership system, a user running under the aegis of an authority may retrieve any element whose privacy level is less than or equal to that of the authority. (Incidentally, the product documentation indicates that the privacy level of an element may be higher than that of its owner; this allows a user to create a private object that the user may access, but none of the user's peers may access.)

The *nil* authority is given a privacy level of 0, which reflects its role as the 'owner' of elements intended to be universally retrievable. More formally

DD7	_____
DD6	
$priv : EI \leftrightarrow Level$	
$dom\ priv = elements$	
$priv\ nil = 0$	
$\forall user : auth; elt : elements \bullet$	
$priv\ elt \leq priv\ user \Rightarrow user\ canretrieve\ elt$	

Again this definition does not completely define *canretrieve*; it is strengthened in Figure 6.1 below.

A dictionary administrator may decide for operational reasons to nominate one authority as the master authority. This authority (if one has been nominated) may retrieve and update any element in the dictionary, irrespective of ownership. A master authority should not be confused with the dictionary administrator: although the two roles might be played by the same person in many organisations, their functions are entirely different.

The only element that does not have a privacy level in the range 0 .. 99 is the master authority (if there is one). For our purposes it will simplify matters if we attribute privacy level 99 to a master element if one exists: whilst this is not strictly in accordance with the choice of representation made by the ICL designers, its consequences are precisely the same. Note that the master authority (if there is one) may not be forbidden to update elements. Note also that, by virtue of its privacy level, the master authority has the right to retrieve any element at all.

Technical note: $\mathbb{F}^1 EI$ means a finite set of *EI* with at most one element.

$$\mathbb{F}^1 X == \{S : \mathbb{F} X \mid \#S \leq 1\}$$

<i>DD8</i>
<i>DD7</i>
$master : \mathbb{F}^1 EI$
$master \subseteq auth$
$priv(\downarrow master) \subseteq \{99\}$
$master \cap (\text{dom } maynotupdate) = \{\}$

This almost concludes the first part of our description of the information structures which characterise a DDS and the invariant relations which hold between them. In Figure 6.1 we summarise these information structures, formally simplifying some of the predicates and recording two more things. The first of these is that the conditions hitherto outlined are the *only* conditions under which retrieval can take place: this is signified by strengthening the implications (\Rightarrow) of *DD6* and *DD7* to equivalences (\Leftrightarrow). The second characterises updating more positively: an element may be updated by its owner or by the master authority. Notice that it is possible for a non-master authority that owns an element to be prevented from updating it.

Since we have not yet given any details of the structure of values, we have no way yet of recording the fact that the owner of a stored element is stored as its *authority* property. Nor can we record the fact that the set of authorities to whom an authority has delegated all its access rights is stored as the *retrieval* property of that authority element, and that the set of authorities which have been given the right of access to an element by its owner is stored as that element's *retrieval* property. We will only be able to go into these details after revealing more of the internal structure of stored values, element identifiers and property names in Section 6.5.

6.4 DDS dynamics: Part 1

In this section and its companion we describe the way in which certain commands affect DDS information structures. In fact there are two distinct kinds of DDS run: administrative runs, during which administrative commands may be issued; and ordinary runs, during which they may not. For the purposes of our investigation the main difference is that certain kinds of element (in particular *AUTHORITY elements) may be inserted only during administrative runs, but otherwise may not.

Although one might expect the administrator to be the same as the *master* authority, this turns out not to be so. For simplicity, therefore, we have avoided consideration of administrative commands and concentrated on the most important non administrative commands. Later we suggest a small simplification of the design which would avoid the distinction between administrative and ordinary runs.

6.4.1 The state of a running DDS

In characterising the state of a running DDS we must account for the fact that users 'log in' under the aegis of an authority, which is deemed in our model to be the *nil* authority for those users who run under 'no authority'. In fact a password-based scheme is used to check that an individual has the right to log in as a particular authority, but the details of logging in are beyond the scope of this report.

Once the running authority is established, elements are processed by establishing the 'element context', i.e. the identifier of the element to which subsequent commands

<i>DD9</i>
$store : EI \multimap E$ $elements, auth : \mathbb{F} EI$ $owner : EI \multimap EI$ $delegates, mayretrieve, maynotupdate : EI \leftrightarrow EI$ $priv : EI \multimap Level$ $master : \mathbb{F}^1 EI$ $canretrieve, canupdate : EI \leftrightarrow EI$
$elements = \text{dom } store$ $\text{ran } owner \subseteq auth \subseteq elements$ $\text{dom } owner \subseteq elements \setminus auth$ $nil \in auth$ $\text{dom } owner = elements \setminus auth$ $delegates \in (auth \leftrightarrow auth)$ $\text{dom } mayretrieve \subseteq auth$ $\text{ran } mayretrieve \subseteq elements \setminus auth$ $\text{dom } maynotupdate \subseteq auth$ $\text{dom } priv = elements$ $priv \ nil = 0$ $master \subseteq auth$ $priv \setminus master \subseteq \{99\}$ $master \cap (\text{dom } maynotupdate) = \{\}$ $(\forall user : auth; elt : elements \bullet user \ \underline{canretrieve} \ elt \Leftrightarrow$ $\quad user = owner \ elt \vee$ $\quad (owner \ elt) \ \underline{delegates} \ user \vee$ $\quad user \ \underline{mayretrieve} \ elt \vee$ $\quad priv \ elt \leq priv \ user)$ $(\forall user : auth; elt : EI \bullet user \ \underline{canupdate} \ elt \Leftrightarrow$ $\quad user \in \{owner \ elt\} \cup master \wedge$ $\quad \neg (user \ \underline{maynotupdate} \ elt))$

Figure 6.1: Summary of the information structures of a DDS

will refer. One thing that the product documentation does not make quite clear is whether or not the element context identified by the user must always refer to an already-defined element. This seems plausible, however, since there is a special element whose identifier is *null*; the *null* element context prevails at the beginning of a run, after certain classes of errors, and after the stop command has been issued. We therefore introduce a constant

$$\mid \text{ null : EI}$$

and characterise the state of a running DDS by the following:

DDS
$DD9$
$user : EI$
$elementcontext : EI$
$user \in auth$
$null \in elements$
$(\forall a : auth \bullet a \text{ mayretrieve } null)$
$user \text{ canretrieve } elementcontext$

It is worth noting that the final constraint is a requirement *imposed* on the user, namely that he or she should be *able* to retrieve the element to which commands will subsequently refer.

In order to capture the effect of a command on a DDS, we relate the observations made before the command is performed (denoted by the undashed names) to those that can be made afterwards (denoted by dashed names). As is customary we factorise our description of the commands into those properties that all the commands have in common and those that are peculiar to particular commands.

The schema ΔDDS summarises the common characteristics of the effects of non-administrative commands on a DDS: the set of declared authorities may not change, nor may the master authority, nor may the authority of the running user.

ΔDDS
DDS
DDS'
$auth' = auth$
$master' = master$
$user' = user$

Some commands change the element context, or just display parts of the stored dictionary, but do not change the information structures concerned with access control; their additional common properties are summarised in the schema $\square DDS$.

$\square DDS$
ΔDDS
$owner' = owner$
$delegates' = delegates$
$priv' = priv$
$mayretrieve' = mayretrieve$
$maynotupdate' = maynotupdate$

6.4.2 The display command

The simplest command to describe is the DISPLAY command, which displays parts of the current element in a readable form. Below, we simply indicate that the user must supply some property names, and that on completion of the command he or she is presented (output *readable!*) with the values of the required properties as stored for the element currently in context. We do not concern ourselves with the precise form in which the display is presented.

<i>Display</i>
$properties? : \mathbb{F} P$
$\square DDS$
$readable! : P \leftrightarrow V$
$\#properties? = 1 \vee properties? = P$
$readable! = properties? \triangleleft (store\ elementcontext)$
$store' = store$
$elementcontext' = elementcontext$

Those familiar with DDS should note that the ALLPROPERTIES variant of the DISPLAY command corresponds to $properties? = P$, whilst the **property-keyword* variant corresponds to $\#properties? = 1$. In order to simplify our description we have not formalised either the PROMPTLIST or the EXPLOSION variants of this command. To do either would require a more detailed model than we have so far presented, though the detail is not complex.

6.4.3 Setting the element context

One command which plays at least three roles in the system is the FOR command: all three variants of which the author is aware set ‘context’ in one way or another. They are:

FOR VERSION *< version element identifier >*
 FOR AUTHORITY *< authority element identifier >*
 FOR *< element identifier >*

Since we do not treat version control here, we shall not consider the first of these. The second corresponds to the beginning of a DDS run: it just sets the user authority after checking a password and we shall not consider it here either. The third command is used to set the current element context: it sets the null context if presented with an element name that the current authority is not allowed to retrieve, or one that has not yet been defined.

FOR $eid? : EI$ $\square DDS$
$store' = store$ $user \text{ canretrieve } eid? \wedge eid? \in elements \wedge elementcontext' = eid?$ \vee $eid? \notin elements \wedge elementcontext' = null$ \vee $\neg (user \text{ canretrieve } eid?) \wedge elementcontext' = null$

This concludes our account of the commands that have no effect on the stored information. Before we can describe the remaining commands it will be necessary for us to take a small detour and explain the relationship between the stored elements and the access-control information.

6.5 Access-control information

In this section we formalise the fact that data dictionaries are self-describing. To put this another way, the abstract information structures that we introduced in order to explain the control of access to elements are actually represented by means of elements and properties stored in the dictionary.

The properties named **AUTHORITY*, **PRIVACY* and **RETRIEVAL* are defined for most elements, so we introduce distinct constants into our specification which will henceforth stand for these property names:

$authority, privacy, retrieval : P$
$authority \neq privacy \neq retrieval \neq authority$

Technical note: ‘ $a \neq b \neq c \neq d$ ’ abbreviates ‘ $a \neq b \wedge b \neq c \wedge c \neq d$ ’. The above declaration introduces three variables of type P ; we require the values of these variables to be distinct.

Properties may take values from a variety of types, but in the first part of this section the only types we shall be concerned with are the numbers, \mathbb{N} , single element identifiers, EI , and finite sets of element identifiers, $\mathbb{F} EI$. We can formalise the idea that elements of these types can all be represented as values by introducing the three injective functions:

$Number : \mathbb{N} \rightarrow V$ $Element : EI \rightarrow V$ $Elements : (\mathbb{F} EI) \rightarrow V$
$\text{disjoint}(\text{ran } Number, \text{ran } Element, \text{ran } Elements)$

Technical note: The definition above signifies that V contains exactly one value for each number $n : \mathbb{N}$, one value for each single element identifier $ei : EI$, and one value for each set of element identifiers $eis : \mathbb{F} EI$. These values are distinct and denoted respectively by the terms $Number(n)$, $Element(ei)$ and $Elements(eis)$.

All that now needs doing is to strengthen the invariant of the existing DDS description. In Figure 6.2 we summarise the qualities inherent in our use of the term ‘self-describing.’

<i>DD10</i>	
<i>DD9</i>	
<i>has</i> : <i>EI</i> \leftrightarrow <i>P</i>	
$\forall ei : EI; p : P \bullet$	$ei \text{ has } p \Leftrightarrow ei \in elements \wedge p \in \text{dom}(store\ ei)$
$\forall ei : elements \bullet$	$ei \text{ has privacy} \wedge$
	$Number(priv\ ei) = (store\ ei)(privacy) \vee$
	$\neg (ei \text{ has privacy}) \wedge priv\ ei = 0$
$\forall a : auth \bullet$	$a \text{ has retrieval} \wedge$
	$Elements(delegates(\{a\})) = (store\ a)(retrieval) \vee$
	$\neg (a \text{ has retrieval}) \wedge delegates(\{a\}) = \{\}$
$\forall ei : (elements \setminus auth) \bullet$	$(ei \text{ has retrieval} \wedge$
	$Elements(mayretrieve^{\sim}(\{ei\})) = (store\ ei)(retrieval))$
	\vee
	$\neg (ei \text{ has retrieval}) \wedge mayretrieve^{\sim}(\{ei\}) = \{\}$
$\forall ei : (elements \setminus auth) \bullet$	$ei \text{ has authority} \wedge$
	$Element(owner\ ei) = (store\ ei)(authority) \vee$
	$\neg (ei \text{ has authority}) \wedge owner\ ei = nil$

Figure 6.2: Data Dictionaries are self-describing

In order to simplify our account of the default values provided by the system, we have introduced an additional observation: the relation *has* which holds between an element identifier *ei* and a property name *p* exactly when *ei* has been stored with a property named *p*. The first additional predicate records the fact that the privacy level of each stored element is represented by the numeric value of its privacy property. The second predicate records that the retrieval property of each authority element represents the set of authorities who stand in the relation *delegates* to it. The third predicate records that the value of the retrieval property of each non-authority element represents the set of authorities which may retrieve it, and the fourth predicate that its authority property represents its owner. Notice the different interpretations given to the *retrieval* property of an authority element and the same property of a non-authority element.

Hitherto we have given a rather abstract characterisation of the information structure which prevents certain authorities updating certain types of element, but we are now in a position to give a fuller account. In order to do so we need to examine the structure of the space of element identifiers (*EI*) in a little more detail. In fact this space is two dimensional; an element is identified by an element-type-identifier (known in the product documentation as an element keyword) and a within-type identifier. If we let *K* denote the set of element type identifiers, and *I* denote the set of within-type identifiers, then we can define

$$\begin{array}{l} [K, I] \\ EI == K \times I \end{array}$$

An authority is prevented from updating certain types of element if the element which describes the authority has a property called *INHIBIT. The value of this property is the set of element type identifiers to which the authority is denied update rights – despite any retrieval rights it may have. First we introduce another constant

$$\begin{array}{|l} inhibit : P \\ \hline inhibit \notin \{authority, retrieval, privacy\} \end{array}$$

and indicate that sets of element type identifiers may also be stored:

$$\begin{array}{|l} Types : \mathbb{F} K \rightarrow V \\ \hline disjoint(\text{ran } Types, \text{ran } Number, \text{ran } Element, \text{ran } Elements) \end{array}$$

We then strengthen the *DD10* invariant a little further, to indicate that the type keyword of the element identifier that may not be updated by an authority *a* is one of the set of keywords stored as the *INHIBIT property of the authority *a*.

$$\begin{array}{|l} DD11 \\ DD10 \\ \hline \forall a : auth; et : K; i : I \bullet \\ \quad a \text{ maynotupdate } (et, i) \Leftrightarrow \\ \quad \quad a \text{ has } inhibit \wedge et \in Types \sim ((store \ a)(inhibit)) \end{array}$$

Our account of self-description is now as detailed as it needs to be for the purposes of this report. Interested readers may care to take the account further and formalise

the fact that details of the properties possessed by elements of each type are recorded in the database, as are details of the representation of each property. As a hint, we will show how the authority elements in the dictionary are identified. First we introduce the constant

| *AUTHORITY* : *K*

to denote the *AUTHORITY* element keyword.

Note: In fact all elements with the same type have property names drawn from the same set of names; these are stored as the **SYSTEM-PROPERTIES* and **USER-PROPERTIES* properties of the element which describes the type. A complete formalisation of this is possible within the framework we have already established, but goes beyond the scope of this report.

All that remains is to strengthen the state invariant yet again.

<i>DD12</i>	
<i>DD11</i>	
$auth = \{ei : elements \mid first\ ei = AUTHORITY\}$	

In other words, the authority elements are exactly those whose keyword is *AUTHORITY*.

6.6 DDS dynamics: Part 2

In this section we complete our description of the DDS commands with a formalisation of the behaviour of the *INSERT* and *DELETE* commands. The *REPLACE* command is simply a combination of *DELETE* followed by *INSERT*, so we leave its formalisation as an exercise for interested readers. Our formalisation is partial, in the sense that we account only for the behaviour of successful commands. Whilst behaviour in the case of erroneous commands can easily be described within the present framework, doing so would not be particularly useful, especially in view of the simplifications we have made (see Appendix 6.8). The change of state schema is updated with the additional components and constraints.

$$\Delta DDS1 \hat{=} DD12 \wedge DD12' \wedge \Delta DDS$$

6.6.1 Inserting elements

From its description in the product documentation, the *INSERT* command appears to have two variants. The first takes a new element identifier and a set of property-name, property-value pairs – in other words, an element – and stores the element as the value of the identifier. If no *authority* property is given, then the owner of the new element will be the current user; if no *privacy* property is given, then the element will be given the privacy level of the current user. The command sets the current element context to be the new element.

$ \begin{array}{l} \text{InsertNewElement0} \\ \hline eid? : EI \\ \text{newelement?} : P \multimap V \\ \Delta DDS1 \\ \hline eid? \notin \text{dom store} \wedge \\ (\text{let } \text{newelement}' == \\ \quad \{ \text{authority} \mapsto \text{Element } user, \\ \quad \quad \text{privacy} \mapsto \text{Number}(\text{priv } user) \} \oplus \text{newelement?} \bullet \\ \quad \text{store}' = \text{store} \oplus \{ eid? \mapsto \text{newelement}' \}) \\ \quad \text{elementcontext}' = eid? \end{array} $
--

This description is such that if the *privacy* and *authority* properties are specified in such a way that *elementcontext'* is no longer accessible, then the insert operation will fail (the last invariant of *DDS* will not be satisfied). It seems to indicate that a user running under one authority can add an element to the database but give ownership rights to another authority. Whilst we found this rather difficult to rationalise, we could not discover anything in the documentation that forbids it. On asking the designers what really happens – an option that might not be open to the average *DDS* user – we discovered that only the master authority can give ownership rights to another authority when creating a new element. When the current user is not the master and *authority* and *privacy* properties are supplied, then they must be the ones which the system would provide by default anyway. Formalised concisely we have:

$ \begin{array}{l} \text{InsertNewElement} \\ \hline \text{InsertNewElement0} \\ \hline user \notin \text{master} \Rightarrow \\ \quad \text{authority} \in \text{dom newelement?} \Rightarrow \\ \quad \quad \text{newelement?}(\text{authority}) = \text{Element}(user) \wedge \\ \quad \text{privacy} \in \text{dom newelement?} \Rightarrow \\ \quad \quad \text{newelement?}(\text{privacy}) = \text{Number}(\text{priv } user) \end{array} $
--

The second variant of the insert command takes a set of property-name plus property-value pairs and incorporates them into the current element, providing that it has no existing property with any of the names supplied.

$ \begin{array}{l} \text{InsertNewProperties0} \\ \hline \text{newprops?} : P \multimap V \\ \Delta DDS1 \\ \hline user \text{ canupdate } \text{elementcontext} \\ \neg (\exists p : (\text{dom newprops?}) \bullet \text{elementcontext } \underline{\text{has}} p) \\ \text{store}' = \text{store} \oplus \\ \quad \{ \text{elementcontext} \mapsto ((\text{store } \text{elementcontext}) \cup \text{newprops?}) \} \\ \text{elementcontext}' = \text{elementcontext} \end{array} $

The documentation does not make it clear whether or not administrative properties may be added to an element by authorities other than its owner once it has been inserted. On asking the designers, we discovered that the only administrative property for which the description given above fails to account is the *authority* property:

the master authority can give ownership of an unowned element to any authority, but non-master authorities can only take the ownership of such elements for themselves. More formally:

<i>InsertNewProperties</i>	_____
<i>InsertNewProperties0</i>	
$authority \in \text{dom } \text{newprops?} \Rightarrow$	
$user \in \text{master} \vee$	
$\text{newprops? } authority = \text{Element } user$	

6.6.2 Deleting elements

DELETE appears in two variants: in the first, the user explicitly mentions an element for the command to delete.

<i>DeleteElement</i>	_____
$eid? : EI$	
ΔDDS1	
$user \text{ canupdate } eid?$	
$store' = \{eid?\} \triangleleft store$	
$elementcontext' = null$	

In its second form, the user mentions some properties to be removed from the current element. Unfortunately, the documentation does not make it clear whether or not users may delete administrative properties from elements to which they have update rights, nor is it quite clear what happens if the last remaining property of an element is deleted. At first we assumed, albeit uneasily, that administrative properties can be deleted, and that elements with no properties can remain in the store, so to that extent our formalisation was inaccurate. Discussions with the implementation team proved our unease to be well founded; we learned that if administrative properties are deleted from an element by the user, then they revert to the default values which the system would have provided if the element had just been inserted.

<i>DeleteProperties</i>	_____
$props? : \mathbb{F} P$	
ΔDDS1	
$user \text{ canupdate } elementcontext$	
$(\text{let } element' ==$	
$\{authority \mapsto \text{Element } user, privacy \mapsto \text{Number}(priv \text{ user})\} \oplus$	
$props? \triangleleft (store \text{ elementcontext}) \bullet$	
$store' = store \oplus \{elementcontext \mapsto element'\}$	
$elementcontext' = elementcontext$	

6.7 Prospects

Although we would have liked to go on to describe the control of multiple versions in DDS, the present design proved too hard for us to formalise simply. The specification,

therefore, has its limitations and it would be an unwise user who relied upon formal deductions from it to discover the consequences of actions he might take whilst running the system itself. It nevertheless remains useful as a pedagogical tool because it provides a discursive introduction to the concepts that underlie access control.

In our view the principal benefit of constructing the formal specification is the fact that we have developed a framework within which designs of future dictionaries can easily be investigated. Whilst it has been an interesting challenge to build a mathematical model of a software system such as the Data Dictionary System, the enterprise would remain simply an academic exercise if we were to stop at this point, so we have tried to indicate how to use the framework by using it to make a tentative proposal for simplifying the system. This is presented in Appendix 6.8.

Acknowledgements ICL sponsored the ten-day pilot experiment in technology transfer which led (*inter alia*) to the production of this report. It is a pleasure to acknowledge the help of Roger Stokes of ICL, who despite the multiplicity of demands imposed on his time and talents, always remained interested enough in the experiment to convince me that it was worthwhile. Jean-Raymond Abrial first showed me how to apply mathematics to software specification and remains a continuing source of inspiration.

6.8 Appendix: potential simplifications

Those familiar with DDS will have noticed that we have made an important simplification already, by ignoring the ‘facility’ to refer to as-yet-undefined authorities when adding or modifying properties. Although we have no definite knowledge about the operational consequences of this facility, we hazard a guess that it causes more aggravation than it saves: readers who have been victims of implicit declarations in FORTRAN may care to comment on this.

The most obvious additional simplification would be to drop the independent notion of privacy level, which seems to be orthogonal to authorities and ownership. We are tempted to wonder if there are any DDS installations where both privacy and authority are employed within the same dictionary. A further simplification would

be to remove the distinction between the system administrator and other authorities. This might well pay dividends in terms of enhancing the functionality of the system and reducing the complexity of its documentation and implementation. Our design goal is based on a new interpretation of the meaning of an authority element, which we prefer to think of as a role, or locus of responsibility, rather than a particular person. Indeed it is often the case that one individual plays several distinct roles in an organisation.

In the design outlined below we make every authority subordinate to ('owned by') some other authority; the root of this tree of authorities is the system administration authority (which owns itself). Power to alter properties of elements reposes ultimately in the administrator, which is able to delegate them to subordinate authorities, which in turn can delegate them further if need be. Any element that several authorities need to retrieve or to update should be owned by an authority which is higher in the tree than all of them, and which delegates its retrieval or update rights to them all.

In order to formalise this design, we first need to introduce the idea of a 'loop-free' function, sometimes called a 'tree' or 'forest'. Consider a homogeneous function

$$\boxed{\begin{array}{l} [X] \\ f : X \rightarrow X \end{array}}$$

We say that an element $x' : X$ is *reachable via f* from an element $x : X$ if there is at least one non-zero number, $n : \mathbb{N}_1$ for which $x \underline{f}^n x'$. When this is the case, we write

$$x \underline{f}^+ x'.$$

More formally, we can define:

$$\boxed{\begin{array}{l} [X] \\ _+ : (X \rightarrow X) \rightarrow (X \rightarrow X) \\ \hline \forall f : X \rightarrow X; x, x' : X \bullet \\ (x, x') \in (f^+) \Leftrightarrow (\exists n : \mathbb{N}_1 \bullet (x, x') \in (f^n)) \end{array}}$$

Note that this definition is a specialisation to functions of the definition of $_+$ given for relations in the glossary (Appendix A.7).

A function $f : X \rightarrow X$ is said to be *loop-free*, or a *forest*, if there is no $x : X$ which is reachable from itself via f . More formally:

$$\text{Forest}[X] == \{f : X \rightarrow X \mid \neg (\exists x : X \bullet (x, x) \in (f^+))\}$$

Our first approximation to a description of the design outlined above recalls our description of the standard data dictionary: the main difference is that all elements (including authority elements) are owned, and that if we confine our attention to authorities other than the administrator, the ownership function is a tree.

The administrator is reachable from every element via the ownership function, i.e. the administrator is ultimately responsible for everything in the dictionary.

DD13

$$\begin{aligned} store &: EI \multimap E \\ elements &: \mathbb{F} EI \\ auth &: \mathbb{F} EI \\ owner &: EI \multimap EI \\ admin &: EI \end{aligned}$$

$$\begin{aligned} elements &= \text{dom } store \\ \text{dom } owner &= elements \\ \text{ran } owner \subseteq auth \subseteq elements \\ admin &\in auth \\ \{admin\} \triangleleft owner &\in \text{Forest}[EI] \\ \forall ei : elements \bullet (ei, admin) &\in (owner^+) \end{aligned}$$

The information structures from which the relations *canretrieve* and *canupdate* can be derived are similar to those in the original design, except that there is no longer a role for privacy levels, the *nil* authority no longer exists, and update permission is characterised positively rather than negatively.

DD14

DD13

$$\begin{aligned} delegates &: EI \leftrightarrow EI \\ mayretrieve &: EI \leftrightarrow EI \\ mayupdate &: EI \leftrightarrow EI \end{aligned}$$

$$\begin{aligned} delegates &\in auth \leftrightarrow auth \\ \text{dom } mayretrieve &\subseteq auth \\ \text{dom } mayupdate &\subseteq auth \\ delegates &\subseteq owner^\sim \end{aligned}$$

The last predicate states that an authority may only delegate its rights to authorities for which it is responsible.

An authority can retrieve an element if it or any of its subordinates own the element, or if it has been given explicit permission to retrieve it, or if it has been delegated rights to retrieve the element. An authority can update an element if it or any of its subordinates owns the element, or if it has been given permission to update the element.

DD15	
DD14	
$canretrieve : EI \leftrightarrow EI$	
$canupdate : EI \leftrightarrow EI$	
$\text{dom } canretrieve \subseteq auth$	
$\text{dom } canupdate \subseteq auth$	
$\forall user : auth; elt : elements \bullet$	
$user \underline{canretrieve} elt \Leftrightarrow$	
$((elt, user) \in (owner^+) \vee$	
$(owner \ elt) \underline{delegates} user \vee$	
$user \underline{mayretrieve} elt)$	
$\forall user : auth; elt : elements \bullet$	
$user \underline{canupdate} elt \Leftrightarrow$	
$(elt, user) \in (owner^+)$	
\vee	
$user \underline{mayupdate} elt$	

It might be nice if no authority possessed any capabilities that its owner does not also possess. In other words, if within a dictionary:

$$\begin{aligned} canretrieve &\subseteq owner \circ canretrieve \wedge \\ canupdate &\subseteq owner \circ canupdate \end{aligned}$$

Interested readers may care to check whether or not this is the case, and if not, to modify our formalisation so that it is.

Finally we propose a small project for the interested reader. Devise representations (along the lines suggested by Figure 6.2) for *mayretrieve*, *mayupdate*, and *delegates*. These should make the relations *canretrieve* and *canupdate* simple to compute, and also make the system invariant simple to check.

Chapter 7

Flexitime specification

Ian Hayes

Abstract This paper gives a simplified specification of an actual flexitime system. It is interesting for a number of reasons. It is brief and not too complicated, and gives some good examples of the power of set theory in specification. A state is used which is far richer than that necessary for an implementation, and this approach has as its reward an overall simplification of the specification. It is simplified also by using an absolute time frame rather than one using times only within the current pay period.

7.1 Introduction

Flexitime is a compromise between the rigidity of fixed working hours (e.g. ‘9 a.m. to 5 p.m.’) and the relative freedom having only to work a certain *number* of hours (e.g. 35 hours per week). The flexitime system developed here requires a certain number of hours to be worked in each pay period, and in addition requires that all the hours should be within certain limits (e.g. between 6 a.m. and 8 p.m.).

Keeping track of the time worked for each employee can be computerised, by having employees ‘clock in’ whenever they start work and ‘clock out’ whenever they stop.

7.2 State

We only record working time to the nearest minute:

$$Time == \mathbb{N}$$

A period of time can be represented by a set of (not necessarily contiguous) minutes:

$$Period == \mathbb{P} Time$$

We can represent the standard working times for a pay period by a set containing all the minutes between 9 a.m. and 5 p.m., excluding the lunch break from 12 noon

to 1 p.m., for all the days in the pay period. In a similar way we can represent the range of permissible flexitime working hours by a set of times.

The function *Standard_Hours* takes a time as argument, and gives the set of standard working times for the pay period encompassing the time given as its argument. For example, if pay periods were weekly, *Standard_Hours* (12:00 14 February 1986) might return the set of all minutes between 9 a.m. and 12 noon, and 2 p.m. and 5 p.m. on each day of the week from 10 to 14 February 1986.

Similarly, the function *Flexitime_Hours* gives the set of minutes that *could* be worked (and credited) in the period encompassing the time given as an argument.

Our model of the system records the times *worked* for all the employees, plus the time at which people currently working clocked *in*. Each employee is assigned a unique identifier from the set *Ident*.

[*Ident*]

$ \begin{array}{l} \textit{Flexi} \\ \textit{Standard_Hours}, \\ \textit{Flexitime_Hours} : \textit{Time} \rightarrow \textit{Period} \\ \textit{worked} : \textit{Ident} \leftrightarrow \textit{Period} \\ \textit{in} : \textit{Ident} \leftrightarrow \textit{Time} \end{array} $
$\text{dom}(\textit{in}) \subseteq \text{dom}(\textit{worked})$

7.3 Operations

Each operation transforms a state before (*Flexi*) to a state after (*Flexi'*):

$$\Delta \textit{Flexi} \hat{=} \textit{Flexi} \wedge \textit{Flexi}'$$

Some operations do not change the state:

$$\Xi \textit{Flexi} \hat{=} [\Delta \textit{Flexi} \mid \theta \textit{Flexi}' = \theta \textit{Flexi}]$$

Clocking in and out operations performed by employees involve them inserting their unique (card) key into a special terminal, which transmits the employee's identifier and the current time to the system. The system responds with an indicator of the operation performed taken from the following set:

$$\textit{Response} ::= \textit{In} \mid \textit{Out} \mid \textit{Balance} \mid \textit{IdUnknown}$$

The common part of the clocking operations is given by $\Delta \textit{Clocking}$.

$ \begin{array}{l} \Delta \textit{Clocking} \\ \Delta \textit{Flexi} \\ \textit{ident?} : \textit{Ident} \\ \textit{t?} : \textit{Time} \\ \textit{ind!} : \textit{Response} \end{array} $
$ \begin{array}{l} \textit{ident?} \in \text{dom}(\textit{worked}) \wedge \\ \textit{Standard_Hours}' = \textit{Standard_Hours} \wedge \\ \textit{Flexitime_Hours}' = \textit{Flexitime_Hours} \end{array} $

The identity of the employee must be known. Clocking operations do not affect *Standard_Hours* or *Flexitime_Hours*.

The operation of clocking in is given by the following schema:

<i>ClockIn_0</i>
$\Delta \textit{Clocking}$
$ident? \notin \text{dom}(in) \wedge$ $t? \in \textit{Flexitime_Hours}(t?) \wedge$ $in' = in \cup \{ident? \mapsto t?\} \wedge$ $worked' = worked \wedge$ $ind! = In$

The employee must not have clocked in already and the current time must be in the bounds of the flexitime working hours for the current pay period. The employee is clocked in at the given time.

The operation of clocking out is given by the following:

<i>ClockOut_0</i>
$\Delta \textit{Clocking}$
$ident? \in \text{dom}(in) \wedge$ $worked' = worked \oplus$ $\quad \{ident? \mapsto (worked(ident?) \cup (in(ident?) .. (t? - 1)))\} \wedge$ $in' = \{ident?\} \triangleleft in \wedge$ $ind! = Out$

The employee must have clocked in. The minutes worked since clocking in are credited to the employee's time worked. Only the period that lies within flexitime hours really counts towards flexitime, but we have chosen to record the total working time in this specification in order to simplify it and allow extensions to keep track of overtime worked, etc. The minutes worked are all those minutes from the time the employees clock in (though they may not have worked the whole of that minute) upto but not including the minute in which they clock out (even though they have worked part of that minute). On average, partial minutes not worked at clock in should cancel out partial minutes worked at clock out.

On each transaction the system responds with the current credit or debit of time worked by the employee within the current pay period, relative to the standard times. We use the set

$$\textit{RelMinutes} == \mathbb{Z}$$

where a positive value indicates a credit and a negative value indicates a debit.

<i>Worked</i>
$\Delta \textit{Clocking}$
$cr! : \textit{RelMinutes}$
$cr! = \#(worked'(ident?) \cap \textit{Flexitime_Hours}(t?)) -$ $\quad \#\{t : \textit{Standard_Hours}(t?) \mid t < t?\}$

Only the period of time worked within the flexitime hours for the current pay period counts.

The clocking operations in full are given by the following:

$$ClockIn \hat{=} ClockIn_0 \wedge Worked$$

$$ClockOut \hat{=} ClockOut_0 \wedge Worked$$

If employees not currently working insert a key outside flexitime hours they will not be clocked in. However, they will receive an indication of their current time credit.

$\frac{ReadOut}{Worked}$
$\begin{aligned} & ident? \notin \text{dom}(in) \wedge \\ & t? \notin Flexitime_Hours(t?) \wedge \\ & ind! = Balance \wedge \\ & \theta Flexi' = \theta Flexi \end{aligned}$

If an unknown key is inserted an error response is given.

$\frac{Unknown}{\Xi Flexi}$
$\begin{aligned} & ident? : Ident \\ & ind! : Response \end{aligned}$
$\begin{aligned} & ident? \notin \text{dom}(worked) \wedge \\ & ind! = IdUnknown \end{aligned}$

The operations *ClockIn*, *ClockOut* and *ReadOut* have disjoint domains of applicability. Outside flexitime hours only a *ReadOut* can occur. Inside flexitime hours a *ClockIn* occurs if the worker is not already clocked in, otherwise a *ClockOut* occurs. The only other possibility is an *Unknown* key, which again is disjoint from the other possibilities. The operation of inserting a key is completely described by the following:

$$InsertKey \hat{=} ClockIn \vee ClockOut \vee ReadOut \vee Unknown$$

An administrative operation is required to add a new employee. The identity of the new employee is chosen from those not already in use.

$\frac{Add_Employee}{\Delta Flexi}$
$ident! : Ident$
$\begin{aligned} & ident! \notin \text{dom}(worked) \wedge \\ & worked' = worked \cup \{ident! \mapsto \{\}\} \wedge \\ & in' = in \wedge \\ & Standard_Hours' = Standard_Hours \wedge \\ & Flexitime_Hours' = Flexitime_Hours \end{aligned}$

Acknowledgements This paper treats a simplified version of a problem first specified by Jolanta Imbert of the GEC Research Laboratories, Marconi Research Centre.

Chapter 8

Formal specification and design of a simple assembler

Ib Holm Sørensen and Bernard Sufrin

Abstract We present the formal specification of a simple assembler, outline the design of a simple implementation, and demonstrate its correctness. Both specification and design are presented at a rather abstract level, and are therefore unrealistic to some extent. However, it is this high level of abstraction which allows the specification and the design to be simply explained and easily understood, and permits a proof that the design meets the specification.

8.1 Introduction

An assembler is a program that translates a sequence of assembly language instructions into a sequence of machine language instructions ready to place in the store of a computer for execution. In this chapter we assume that the computer for which we are going to specify our assembler is a ‘one address machine’ – in other words each machine instruction has an opcode field and an operand field, and resides at a certain address in the store of the machine. Depending on the value of the opcode field, the operand field may be treated as an address or a number when the instruction is executed by the machine.

Each assembly language instruction determines the value of the opcode and address fields of a corresponding computer instruction, so an assembly language instruction has a symbolic opcode field and a symbolic or numeric operand field. When a symbol appears in the operand field of an assembly language instruction, the assembler should place the address with which the symbol is associated in the operand field of the corresponding machine instruction. A symbol is associated with an address by

including a symbolic label field in the assembly language instruction which determines the content of that address.

Some assembly language instructions, known as *directives*, do not correspond to machine instructions, but are used to signify things such as the end of the input, or a change of the radix in which numbers are expressed. In order to simplify our discussion we will not consider this kind of directive at all. A typical translation performed by the assembler is given in Figure 8.1.

Assembly Language			Machine Language		
Label	Opcode	Operand	Location	Opcode	Operand
v1 :	.const	100	1		100
v2 :	.const	4095	2		4095
loop :	load	v2	3	01	2
	subn	8	4	03	8
	store	v2	5	02	2
	compare	v1	6	50	1
	jumple	exit	7	61	9
	jump	loop	8	71	3
exit :	return		9	77	

Figure 8.1: A typical translation

We can specify the task an assembler must perform by explaining the relationship of its input (a sequence of assembly language instructions) to its output (a sequence of machine language instructions). We will find it easier to investigate the essence of this relationship if we avoid considering things like error listings. This is not to say that such things are not important in a more complete specification of requirements for the assembler, but at present we are interested in capturing the essence of its task, which is the translation of assembler instructions into machine instructions.

8.1.1 The structure of instructions

The first abstraction step we take is to decide that we need not understand how assembler instructions are represented as character sequences nor how machine instructions are represented as bit sequences. In particular, all we need to know about assembler instructions is that they have a label field or an opcode field or both, and that they *may* have an operand field which is *either* a symbolic reference or a number, but cannot be both. Similarly, all we need to know about machine instructions is that they must have an opcode field or an operand field and may have both.

Let A denote the set of all possible assembly language instructions and M the set of all possible machine language instructions. The next step in our construction of the predicate is to investigate the structure of the two kinds of instruction, A and M . In order to do so, we shall need to discuss label symbols and operation code symbols. So, let the set of all possible label symbols be denoted by SYM and the set of all possible opcode symbols by $OPSYM$.

$$[A, M, SYM, OPSYM]$$

We can now formalise our idea of the essential structure of an assembly language instruction.

$lab : A \leftrightarrow SYM$	A1
$op : A \leftrightarrow OPSYM$	A2
$ref : A \leftrightarrow SYM$	A3
$num : A \leftrightarrow \mathbb{N}$	A3
<hr/>	
$\text{dom } ref \cap \text{dom } num = \{\}$	A3
$\text{dom } op \cup \text{dom } num \cup \text{dom } ref = A$	A4

A1 formalises the requirement that assembly instructions *may* have a label, A2 that they may have an opcode, A3 that the optional operand may be numeric or symbolic but not both, and A4 the requirement that they must all have an opcode or an operand.

The essential structure of machine instructions may be formalised similarly.

$opcode : M \leftrightarrow \mathbb{N}$	M1
$operand : M \leftrightarrow \mathbb{N}$	M2
<hr/>	
$\text{dom } opcode \cup \text{dom } operand = M$	M3

Finally, we assume that we have been given a way of translating symbolic opcodes to the numbers which they represent, i.e. a function:

$$| \quad mnem : OPSYM \leftrightarrow \mathbb{N}$$

The domain of this function is the set of valid mnemonic opcode symbols.

8.2 Requirements

We require that the assembler translate symbolic operands, where they appear, to numbers representing the corresponding address, translate numeric operand fields without changing them, and that it translate symbolic opcodes to their corresponding number. Our specification is effectively the conjunction of predicates which formalise these requirements.

8.2.1 Symbol definitions

Suppose that the input sequence of assembly instructions is

$$| \quad in : \text{seq } A$$

A sequence is a special kind of function from the natural numbers, so the composition

$$in \circ lab$$

is a function of type, $\mathbb{N} \leftrightarrow SYM$, which maps the position of each assembler instruction in which a symbolic label is defined to the label which is defined there. For the example in Figure 8.1 we have

$$in \circ lab = \{1 \mapsto v1, 2 \mapsto v2, 3 \mapsto loop, 9 \mapsto exit\}$$

The inverse of this function is in general a relation that maps each symbol to the places in the input where it is defined as a label. For this reason we will find it convenient to define

$$symtab == (in \circ lab)^\sim \quad S1$$

In order to formalise the idea that there should be *no multiply defined symbols*, we require that *symtab* be a *function*:

$$\text{symtab} \in \text{SYM} \rightarrow \mathbb{N} \quad \text{S2}$$

In general the inverse of a function may be a one-to-many relation: requiring that *symtab* be a function is the same as requiring that it map each symbol in its domain to a *unique* address. Later we will be able to give additional justification for this intuitively obvious requirement.

8.2.2 Symbolic operands

The composition

$$\text{in} \circ \text{ref}$$

is a function of type, $\mathbb{N} \rightarrow \text{SYM}$, which maps the position of an assembler instruction in the input to the symbol which is referenced there. For the example in the Figure 8.1

$$\text{in} \circ \text{ref} = \{3 \mapsto v2, 5 \mapsto v2, 6 \mapsto v1, 7 \mapsto \text{exit}, 8 \mapsto \text{loop}\}$$

In addition

$$\text{ran}(\text{in} \circ \text{ref})$$

is the set of symbols referenced in the input. So, to express the requirement that all symbols which are referenced by the input are defined there, we write

$$\text{ran}(\text{in} \circ \text{ref}) \subseteq \text{dom } \text{symtab} \quad \text{S3}$$

8.2.3 Numeric operands

The function

$$\text{in} \circ \text{num}$$

of type, $\mathbb{N} \rightarrow \mathbb{N}$, maps the position of an assembler instruction in the input to the number which appears there. For our example

$$\text{in} \circ \text{num} = \{1 \mapsto 100, 2 \mapsto 4095, 4 \mapsto 8\}$$

Because of the axioms for *ref* and *num*, the two functions $(\text{in} \circ \text{ref})$ and $(\text{in} \circ \text{num})$ have disjoint domains.

8.2.4 Symbolic opcodes

The function

$$\text{in} \circ \text{op}$$

of type, $\mathbb{N} \rightarrow \text{OPSYM}$, maps the position of an assembler instruction in the input to the opcode symbol that is referenced by it. To formalise the requirement that all referenced opcode symbols be valid mnemonics, we write

$$\text{ran}(\text{in} \circ \text{op}) \subseteq \text{dom } \text{mnem} \quad \text{S4}$$

8.2.5 Operands of machine instructions

Suppose that the output sequence of machine instructions is

$$\mid \text{ out} : \text{seq } M$$

then the function

$$\text{out} \circ \text{operand}$$

of type, $\mathbb{N} \leftrightarrow \mathbb{N}$, maps each machine address to the value of the operand field of the instruction stored there. If the assembler instruction at position n has a symbolic operand, then the operand field of the instruction at location n should take the value of the symbol

$$(\text{in} \circ \text{ref})(n)$$

that is, $\text{ref}(\text{in}(n))$.

Provided that symtab is a function and that the symbol is indeed defined, then its value is uniquely determined by

$$(\text{in} \circ \text{ref} \circ \text{symtab})(n)$$

that is, $\text{symtab}(\text{ref}(\text{in}(n)))$.

If the assembler instruction at position n has a numeric operand then the operand of the corresponding machine instruction should take the value

$$(\text{in} \circ \text{num})(n)$$

We can express both of these requirements as a single equality, namely

$$(\text{out} \circ \text{operand}) = (\text{in} \circ \text{ref} \circ \text{symtab}) \cup (\text{in} \circ \text{num}) \quad \text{S5}$$

In order to check that our formalisation is sensible, we should ensure that the right hand side of this equality is a *function* (since we have already established that the left hand side must be so).

Since by virtue of the structure of the assembly language $\text{in} \circ \text{ref}$ and $\text{in} \circ \text{num}$ must be functions with disjoint domains, the only thing left to ensure is that symtab itself is a function; this condition corresponds to the *no multiply defined symbols* condition which we discussed earlier.

8.2.6 Opcode fields

All that remains is for us to state the relationship we require between the opcode fields of the input and the instruction fields of the output. This is simply

$$\text{out} \circ \text{opcode} = \text{in} \circ \text{op} \circ \text{mnem} \quad \text{S6}$$

Ensuring that every assembler instruction with an opcode field gives rise to a machine instruction with a corresponding field is just a question of ensuring that the domain of the right hand side is equal to the domain of $\text{in} \circ \text{op}$. This is so, provided that the range of $\text{in} \circ \text{op}$ is a subset of the domain of mnem , which corresponds to the condition *all referenced opcodes must be valid mnemonics* discussed earlier.

8.2.7 Specification summary

In this section we summarise the specification by defining the schema *ASSEMBLY*, which characterises the relationship we wish to hold between the inputs and outputs of an assembler. We have labelled the clauses of the specification so as to illuminate the proof steps we take later.

Context

$$[A, M, SYM, OPSYM]$$

$lab : A \leftrightarrow SYM$	A1
$op : A \leftrightarrow OPSYM$	A2
$ref : A \leftrightarrow SYM$	A3
$num : A \leftrightarrow \mathbb{N}$	A3
<hr/>	
$\text{dom } ref \cap \text{dom } num = \{\}$	A3
$\text{dom } op \cup \text{dom } num \cup \text{dom } ref = A$	A4
<hr/>	
$opcode : M \leftrightarrow \mathbb{N}$	M1
$operand : M \leftrightarrow \mathbb{N}$	M2
<hr/>	
$\text{dom } opcode \cup \text{dom } operand = M$	M3
<hr/>	
$mnem : OPSYM \leftrightarrow \mathbb{N}$	

Specification

$ASSEMBLY$	
<hr/>	
$in : \text{seq } A$	
$out : \text{seq } M$	
<hr/>	
$\exists \text{symtab} : SYM \leftrightarrow \mathbb{N} \bullet$	S2
$\text{symtab} = (in \circ lab)^\sim \wedge$	S1
$\text{ran}(in \circ ref) \subseteq \text{dom } \text{symtab} \wedge$	S3
$\text{ran}(in \circ op) \subseteq \text{dom } mnem \wedge$	S4
$(out \circ operand) = (in \circ ref \circ \text{symtab}) \cup (in \circ num) \wedge$	S5
$(out \circ opcode) = (in \circ op \circ mnem)$	S6

8.2.8 Consequences of the specification

It is easy to show that when an assembly is successful, the length of the output sequence is the same as that of the input sequence. More formally

$$ASSEMBLY \vdash \#out = \#in$$

Proof

1. $\text{dom } out = \text{dom}(out \circ operand) \cup \text{dom}(out \circ opcode)$ M3
2. $\text{dom } out = \text{dom}(in \circ ref \circ symtab) \cup \text{dom}(in \circ num) \cup \text{dom}(in \circ op \circ mnem)$ 1,S5, S6
3. $\text{dom } out = \text{dom}(in \circ ref) \cup \text{dom}(in \circ num) \cup \text{dom}(in \circ op)$ 2,S3,S4
4. $\text{dom } out = \text{dom } in$ 3,A4
5. $\#(\text{dom } out) = \#(\text{dom } in)$ 4
6. $\#out = \#in$ 5

□

The precondition for a successful assembly is the existence of an output that satisfies the predicate of *ASSEMBLY*. This is formally denoted by hiding the output.

$$PreASSEMBLY \triangleq ASSEMBLY \setminus (out)$$

Technical note: The hiding operator ‘ \setminus ’ of the schema calculus is defined as follows: the variables that are to be hidden are removed from the signature of the schema in which they are to be hidden; and the predicate of the schema is existentially quantified by the hidden variables. The definition above is therefore equivalent to the following:

$$\frac{PreASSEMBLY}{in : \text{seq } A} \quad \frac{}{\exists out : \text{seq } M \bullet ASSEMBLY}$$

Provided that we now insist that each machine instruction is *uniquely* characterised by its opcode and operand fields, i.e.

$$\begin{aligned} \forall m, n : M \bullet \\ ((opcode \ m) = (opcode \ n) \wedge (operand \ m) = (operand \ n)) \\ \Rightarrow m = n \end{aligned}$$

the schema *PreASSEMBLY* simplifies to the following:

$$\frac{PreASSEMBLY}{in : \text{seq } A} \quad \frac{}{\begin{aligned} \exists symtab : SYM \leftrightarrow \mathbb{N} \bullet & \quad S2 \\ symtab = (in \circ lab)^\sim \wedge & \quad S1 \\ \text{ran}(in \circ ref) \subseteq \text{dom } symtab \wedge & \quad S3 \\ \text{ran}(in \circ op) \subseteq \text{dom } mnem & \quad S4 \end{aligned}}$$

Thus the specification states, amongst other things, that a program which is to be assembled correctly must have no multiply-defined labels, no undefined symbolic references and no invalid opcode mnemonics.

8.2.9 Discussion

We have established the basis for a small theory of simple assemblers. Such a theory, however simple and abstract, gives us an intellectual handle by which we may grasp much more complicated machine and assembly languages, such as those outlined in exercises 4 and 5 below.

By formalising the essence of the relationship required between inputs and outputs of the simple assembler we have illustrated the two principal techniques used in system specification, namely representational abstraction and procedural abstraction. By representational abstraction we mean the statement of all essential characteristics of the information structures involved in describing a situation, without defining the storage structures used in their representation. By procedural abstraction we mean the statement of the input-output relationships involved in an activity without defining the computational structures used to achieve them.

Any program that can be proved to behave in the manner indicated by the *ASSEMBLY* schema, is, as far as we are concerned, an assembler. Of course we have not yet given any clues about how to go about constructing such a program, but that enterprise is the subject of the Section 8.3.

Exercises

1. What should the output sequence of instructions look like for erroneous input? Is it important?
2. Specify the appearance of a listing on which errors, such as multiply-defined and undefined symbolic references, are noted.
3. How could the specification be extended to cover radix directives in the input language?
4. How could the specification be extended so as to describe an assembler for a machine with registers?
5. Specify an assembler for a VAX-like machine, whose machine instructions do not all occupy the same number of addressable units.

8.3 High-level design

In this section we outline the design of a simple two-phase implementation of the assembler. During the first phase the assembler completely builds the machine instructions of the output which have operands specified numerically (or not at all) in the input. It only partly builds the instructions which have operands that are symbolically specified in the input, leaving the values of the operand fields of such instructions indeterminate. It also constructs a reference-table, which records the positions where symbols were referenced in the input, and a symbol table, which records their values. It uses these tables in the second phase to complete the building of the machine instructions by giving values to symbolically specified operands.

We describe the first phase by defining a schema *Phase1*, which specifies the value of the intermediate state in terms of the input. The second phase is described by defining a schema *Phase2*, which uses the intermediate state to derive a value for

the output. Once we have made and explained these definitions we show how to put them together to describe the complete implementation.

The definition and reference information of the intermediate state is characterised as abstractly as possible below – as the relation st (symbol table) and the function rt (reference table). The sequence $core$ consists of machine instructions, some of which may be only partly determined after the first phase.

IS
$st : SYM \leftrightarrow \mathbb{N}$
$rt : \mathbb{N} \rightarrow SYM$
$core : seq\ M$

The symbol table st is defined as a relation in the intermediate state to allow for input to the first phase that contains multiple definitions of a label.

8.3.1 Design of the first phase

After the first phase, the symbol and reference tables should have been built, and all the opcode and numeric operand fields should take the values in $core$ that they will have in the final output.

<i>Phase1</i>	
<i>in</i> : seq <i>A</i>	
<i>IS</i>	
<i>st</i> = (<i>in</i> ∘ <i>lab</i>) [~]	P1.1
<i>rt</i> = (<i>in</i> ∘ <i>ref</i>)	P1.2
(dom <i>rt</i>) ≪ (<i>core</i> ∘ <i>operand</i>) = (<i>in</i> ∘ <i>num</i>)	P1.3
(<i>core</i> ∘ <i>opcode</i>) = (<i>in</i> ∘ <i>op</i> ∘ <i>mnem</i>)	P1.4
ran(<i>in</i> ∘ <i>op</i>) ⊆ dom <i>mnem</i>	P1.5

The first condition, $P1.1$, specifies that the symbol table records all definitions of each label. $P1.2$ specifies that the reference table records the symbol referenced at each location whose assembly instruction had a symbolic operand. $P1.3$ specifies the in-core values of operand fields derived from numeric operands in the input – that is, those operand fields that are not symbolic references. The fourth condition, $P1.4$, specifies the in-core values of the opcode fields. Finally, a precondition of the first phase is that all opcode symbols are valid mnemonics, $P1.5$.

The reader may have noticed that the operand fields of instructions with *symbolic* operands are left unspecified. Our reason for leaving them so is that at least one well-known implementation technique uses the unfilled operand fields to store most of the information present in the reference-table, and we do not wish to exclude such an implementation technique at this early stage of design.

8.3.2 Design of the second phase

The second phase may assume that the output values of all opcode and numeric operand fields are already present in $core$. It must determine the values of operand fields which were specified symbolically. Since the input is no longer accessible, the only way to tell the difference between symbolic and numeric operand fields is by inspecting the domain of the reference table.

Formally, we have:

<i>Phase2</i>	
<i>IS</i>	
<i>out</i> : seq <i>M</i>	
$st \in SYM \leftrightarrow \mathbb{N}$	P2.1
$\text{ran } rt \subseteq \text{dom } st$	P2.2
$(out \circ \text{opcode}) = (core \circ \text{opcode})$	P2.3
$(\text{dom } rt) \triangleleft (out \circ \text{operand}) = (\text{dom } rt) \triangleleft (core \circ \text{operand})$	P2.4
$(\text{dom } rt) \triangleleft (out \circ \text{operand}) = (rt \circ st)$	P2.5

The first two conjuncts constitute a precondition for this phase (since *out* does not occur in them): there must be no multiply-defined symbols, *P2.1*, and all referenced symbols must be defined, *P2.2*. *P2.3* specifies that the opcode fields of the output be exactly the same as they were in *core*, and condition *P2.4* specifies that the operand fields of the output instructions with numeric operands also must be the same as they were in *core*. The last condition, *P2.5*, specifies that the operand fields of the output instructions with symbolic operands are given the values of the appropriate symbols.

8.3.3 Putting the phases together

It is tempting to describe the two-phase implementation by defining a schema

$$Implementation \triangleq Phase1 \wedge Phase2$$

However, when an assembly is complete we do not particularly care what the value of the intermediate state was, so we shall *hide* it in our definition:

$$Implementation \triangleq (Phase1 \wedge Phase2) \setminus (st, rt, core)$$

8.3.4 Correctness of the design

In order to prove that our design is correct, it would be sufficient to prove that it is at least as applicable as the specification, and that the result prescribed by the design is consistent with the specification whenever the specification is applicable. In fact we shall be able to show here that the implementation is *equivalent* to the specification. We do this by expanding and simplifying the definition of the implementation using the rules of the schema calculus and the laws of mathematics.

Taking the formal schema conjunction of the two phases and hiding the intermediate state gives the following:

<i>Implementation</i>	
<i>in</i> : seq <i>A</i>	
<i>out</i> : seq <i>M</i>	
$\exists st : SYM \leftrightarrow \mathbb{N}; rt : \mathbb{N} \rightarrow SYM; core : seq M \bullet$	
$st = (in \circ lab)^\sim \wedge$	P1.1
$rt = (in \circ ref) \wedge$	P1.2
$(dom\ rt) \triangleleft (core \circ operand) = (in \circ num) \wedge$	P1.3
$(core \circ opcode) = (in \circ op \circ mnem) \wedge$	P1.4
$ran(in \circ op) \subseteq dom\ mnem \wedge$	P1.5
$st \in SYM \rightarrow \mathbb{N} \wedge$	P2.1
$ran\ rt \subseteq dom\ st \wedge$	P2.2
$(out \circ opcode) = (core \circ opcode) \wedge$	P2.3
$(dom\ rt) \triangleleft (out \circ operand) = (dom\ rt) \triangleleft (core \circ operand) \wedge$	P2.4
$(dom\ rt) \triangleleft (out \circ operand) = (rt \circ st)$	P2.5

If we exploit the equalities P1.2, P1.3, P2.4, P1.4 and P2.3, this can be simplified to

<i>Implementation</i>	
<i>in</i> : seq <i>A</i>	
<i>out</i> : seq <i>M</i>	
$\exists st : SYM \leftrightarrow \mathbb{N} \bullet$	
$st = (in \circ lab)^\sim \wedge$	
$(dom(in \circ ref)) \triangleleft (out \circ operand) = (in \circ num) \wedge$	IX
$(out \circ opcode) = (in \circ op \circ mnem) \wedge$	
$ran(in \circ op) \subseteq dom\ mnem \wedge$	
$st \in SYM \rightarrow \mathbb{N} \wedge$	
$ran(in \circ ref) \subseteq dom\ st \wedge$	
$(dom(in \circ ref)) \triangleleft (out \circ operand) = (in \circ ref \circ st)$	IY

thus eliminating *core* and *rt* from the quantified predicate.

We now combine IX and IY to yield I5 using the restriction elimination law:

$$\forall R : X \leftrightarrow Y; S : \mathbb{P} X \bullet$$

$$R = (S \triangleleft R) \cup (S \triangleleft R)$$

The law follows directly from the definition of the domain exclusion (\triangleleft) and domain restriction (\triangleleft) operators.

Reordering the conjuncts of the quantified predicate, we obtain our final simplified description of the implementation.

<i>Implementation</i>	
<i>in</i> : seq <i>A</i>	
<i>out</i> : seq <i>M</i>	
$\exists st : SYM \leftrightarrow \mathbb{N} \bullet$	
$st = (in \circ lab)^\sim \wedge$	I1
$st \in SYM \rightarrow \mathbb{N} \wedge$	I2
$ran(in \circ ref) \subseteq dom\ st \wedge$	I3
$ran(in \circ op) \subseteq dom\ mnem \wedge$	I4
$(out \circ operand) = (in \circ num) \cup (in \circ ref \circ st) \wedge$	I5
$(out \circ opcode) = (in \circ op \circ mnem)$	I6

The specification is now equivalent to *ASSEMBLY*.

<i>ASSEMBLY</i>	
$in : \text{seq } A$	
$out : \text{seq } M$	
$\exists \text{symtab} : \text{SYM} \leftrightarrow \mathbb{N} \bullet$	S2
$\text{symtab} = (in \circ \text{lab})^\sim \wedge$	S1
$\text{ran}(in \circ \text{ref}) \subseteq \text{dom } \text{symtab} \wedge$	S3
$\text{ran}(in \circ \text{op}) \subseteq \text{dom } \text{mnem} \wedge$	S4
$(out \circ \text{operand}) = (in \circ \text{ref} \circ \text{symtab}) \cup (in \circ \text{num}) \wedge$	S5
$(out \circ \text{opcode}) = (in \circ \text{op} \circ \text{mnem})$	S6

Because we may rename the bound variables of a quantified predicate without changing the meaning of the predicate, the two schemas *ASSEMBLY* and *Implementation* are clearly equivalent, and we have therefore demonstrated the correctness of our design.

The two phases may now be used almost independently as subjects for further refinement. To be precise, independent refinements of the phases will refine the entire specification provided *either* that the refinement of *Phase1* is deterministic, *or* that all intermediate states produced by the refinement of *Phase1* satisfy the precondition of the refinement of *Phase2*.

8.3.5 Discussion

We have constructed specifications for the two phases of an in-store assembler. In each case we have captured the essence of the information processed by the phases, but in neither case have we specified the order in which the information is processed, nor have we specified the form in which this information will be stored in the computer. This leaves several possibilities open to those who will define a more computer-oriented realisation of the intermediate data structures and algorithmically more explicit realisations of the two phases. In particular, our earlier suggestion that the symbol reference-table be stored in the unfilled operand fields of *core* is consistent with the specification of both phases, and might usefully be explored further by those interested in completing a formally justified derivation of a real assembly program based on this design.

Acknowledgements This chapter is a much altered version of [23]. We have benefited from discussion over the years with colleagues and students at the Programming Research Group. Jean-Raymond Abrial first showed us how to put set theory to productive use as a software engineering tool, and remains a continuing source of inspiration.

Part III

DISTRIBUTED COMPUTING

Work on the Distributed Computing Software Project began at the Programming Research Group of the Oxford University Computing Laboratory in 1982. The goal of the project was to construct and publish the specification of a loosely coupled, distributed operating system, based on the model of autonomous clients having access to a number of shared services.

A fundamental objective of the project was to make use of mathematical techniques of program specification to assist the design, development and presentation of distributed system services.

Part III presents some of the results of the first stage of the project. It begins with an overview of the use of mathematics in system design, and its application to the specification of an example file service. It illustrates how abstraction from details of implementation can allow the exploration of novel system designs.

The rest of Part III contains the user documentation for some of the services that have been implemented. Surprisingly, it was possible to place the original specifications wholly within the corresponding user manuals; and so they illustrate how we have been able to blend our abstract mathematical descriptions with the detail even of the concrete syntax of a programming language interface.

The project was funded by a grant from the UK Science and Engineering Research Council.

Chapter 9

The role of mathematical specifications

Roger Gimson and Carroll Morgan

9.1 Introduction

The aim of the Distributed Computing Software Project was to explore the new possibilities of distributed operating system design which have been made possible by the low cost of distributed processing hardware. The mathematical techniques of program specification and development play a crucial part in this aim for the following reasons:

- we can use *mathematical specifications* to explore designs motivated purely by ease-of-use rather than by ease-of-implementation (since specification allows abstraction from implementation constraints);
- we have a precise notation in which such designs can be reliably communicated to others, and which assists the discovery and discussion of the designs' implications;
- it is possible to present the specifications directly in the user manuals of the distributed operating system, thus increasing their precision while decreasing their size; and
- we are able to use the mathematical techniques of *refinement* to produce implementations that are highly likely to satisfy their specifications (and hence are also accurately described by their user manuals).

It is especially important that those benefits should be realised in the construction of a *distributed* operating system – because distributed operating systems offer the rare opportunity for the user to control the system, rather than vice versa. The high

This chapter appeared in the book *Distributed Computing Systems Programme* (ed. D. A. Duce), published by Peter Peregrinus Ltd. 1984, under the title 'Ease of use through proper specification'.

bandwidth of current local area networks allows an efficient modularity; for example, a structure consisting of largely autonomous services and clients is entirely feasible. In such a system, the choice between (rival) services, and the manner in which they are used, would be entirely up to the clients. This is the basis of the *open systems* approach: provided services are well-specified, clients are free to make use of them in whatever manner is consistent with their specification.

9.2 A first example

One of the most visible parts of any operating system is its file system. Even today, the design of these range in quality from excellent to horrific. But others, of course, may think instead that they range from horrific to excellent: the features one user cannot do without, another may abhor. It is through such *features* that an operating system controls (even the thoughts of) its clients, and this is exactly what we hope to avoid.

A file *service* in a distributed operating system is there to be shared by as many clients as possible. To achieve this, it must be unopinionated: it must have so *few* features that there is nothing anyone could object to. It is only in the context of specification that we can propose such a radical design; any less abstract context introduces efficiency constraints. Some of these, of course, will have to be met eventually, but perhaps not all of the ones that might conventionally be presumed. We must not introduce such constraints simply because we could not express ourselves without them: first we state what we would like – then we can compromise.

As an example, let us consider the simplest file system design one could imagine. We describe it as a partial function *files* from the set *Name* of file names:

$$[Name]$$

to the set *FILE* of all possible files:

$$files : Name \rightarrow FILE$$

We say nothing about the structure of the set *Name*. The structure of the set *FILE* is developed below.

The mathematical notation above introduces the variable *files*, and gives its type as *Name* \rightarrow *FILE*. The English text states that this variable is to describe the file system. Our style of mathematical specification is an example of the Z specification technique, and we will continue to use it below.

We propose two operations only on the file system: *StoreFile* stores a (whole) file, and *RetrieveFile* (destructively) retrieves it.

StoreFile Let *files* be the state of the file system before the operation, and let *files'* be the state afterwards. Let *file?* be the file to be stored, and let *name!* be some filename, chosen by the filesystem, which will refer to the newly stored file. That is, given

$$\begin{aligned} files, files' &: Name \rightarrow FILE \\ file? &: FILE \\ name! &: Name \end{aligned}$$

the effect of *StoreFile* is to choose a new name, which is not currently in use

$$name! \notin \text{dom } files$$

and to update the partial function by overriding its current value, so that after the operation it maps the new name to the newly stored file:

$$files' = files \oplus \{name! \mapsto file?\}$$

(We notice as an immediate advantage of our abstraction that we have given the implementor the freedom to store identical but differently named files using shared or separate storage, as he chooses.)

RetrieveFile Let *files* be the state of the file system before the operation, and let *files'* be the state afterwards. Let *name?* be the name of the file to be retrieved, and let *file!* be the file itself. That is, given

$$\begin{aligned} files, files' &: Name \leftrightarrow FILE \\ name? &: Name \\ file! &: FILE \end{aligned}$$

the effect of *RetrieveFile* is to return the named file to the client

$$file! = files(name?)$$

provided it exists

$$name? \in \text{dom } files$$

and to remove the name (and hence the file) from the partial function that represents the file system

$$files' = \{name?\} \triangleleft files$$

The description above is ‘of course’ not feasible with today’s technology – which is a pity. It would be too impractical to have to retrieve a whole large file if we wished, say, just to read one small piece of it. But how wonderful it would be if a file system could be so simple! At least we were able to describe it.

9.3 The first compromises

The best we can do with our simple file system is to use it as the basis for a development of a more practical design – and the description above provides a context into which the necessary compromises can be introduced. Here are some of them (in no particular order).

Compromise	Reason
It must be possible to read the file without deleting it.	The communication medium is not entirely reliable – a breakdown during retrieval could destroy the file without returning its contents.
Clients must be prevented from destroying the files of others (remember, a file can't be updated).	Mistakes are inevitable – even honest clients could accidentally destroy other clients' files.
Files must be given a limited lifetime, and clients must be charged for their storage.	Any implementation of the file system, however capacious, will still be finite.

We introduce these compromises in a revised design. First, we name four new sets:

1. the set of client identifications

$[Client]$

2. the set of instants (e.g. seconds from 1 January 1980 – but we need not be specific here)

$Time == \mathbb{N}$

3. the set of costs (e.g. pence)

$Money == \mathbb{Z}$

allowing both debits and credits, and

4. the set that contains all the possible values a client could store in a file (its *contents*)

$[Data]$

The definition of a file is extended to include the identification of its owner, and its time of creation and (eventual) expiry. We collect these attributes in a schema *FILE*, and state at the same time that in any file, the creation time must precede the expiry time.

<i>FILE</i>
<i>owner</i> : <i>Client</i>
<i>created</i> ,
<i>expires</i> : <i>Time</i>
<i>contents</i> : <i>Data</i>
<i>created</i> ≤ <i>expires</i>

The schema FS below describes the state of the file storage system itself:

FS
$files : Name \leftrightarrow FILE$

and the schema ΔFS describes the general aspects of any operation on it.

ΔFS
$files, files' : Name \leftrightarrow FILE$
$who : Client$
$when : Time$

who is the identity of the client performing the operation, and $when$ is the time at which it is performed. We can abbreviate ΔFS (without changing its meaning) by building it from the schema FS instead of directly from the variable $files$.

ΔFS
FS
FS'
$who : Client$
$when : Time$

StoreFile The (revised) *StoreFile* operation we present as a schema including the variables $files$, $files'$, who and $when$ (supplied by ΔFS), as well as the data to be stored ($contents?$), the expiry time ($expires?$), the new name chosen by the service ($name!$) and the charge made in advance ($cost!$). The charge made is some function $Tariff$ of the file (hence of its owner, creation and expiry times, and contents). Here is a possible definition of $Tariff$ (which depends in turn on some function $Size$).

$Size : Data \rightarrow \mathbb{N}$
$Tariff : FILE \rightarrow Money$
$Tariff = (\lambda FILE \bullet (expires - created) * Size(contents))$
$StoreFile$
ΔFS
$contents? : Data$
$expires? : Time$
$name! : Name$
$cost! : Money$
$(\exists FILE' \bullet$
$owner' = who \wedge created' = when \wedge$
$expires' = expires? \wedge contents' = contents? \wedge$
$name! \notin \text{dom } files \wedge$
$files' = files \oplus \{name! \mapsto \theta FILE'\} \wedge$
$cost! = Tariff(\theta FILE'))$

A new file $FILE'$ is constructed which is owned by the client storing it, which records its creation time as the time it was stored, which will expire at the time the client specified (then becoming inaccessible), and whose contents the client supplies. A new name $name!$, not currently in use, is chosen and the file is stored under that name.

ReadFile The *ReadFile* operation returns the expiry time and the contents of the file stored under a given name. Its parameters are the name of the file to be returned (*name?*), when it will expire (*expires!*) and its contents (*contents!*).

<i>ReadFile</i>	_____
ΔFS	
<i>name?</i> : <i>Name</i>	
<i>expires!</i> : <i>Time</i>	
<i>contents!</i> : <i>Data</i>	
$\theta FS' = \theta FS$	
<i>name?</i> $\in \text{dom } files$	
$(\exists FILE \bullet$	
$\theta FILE = files(name?) \wedge$	
$expires > when \wedge$	
$expires! = expires \wedge$	
$contents! = contents)$	

ReadFile does not change the state of the service. The map *files* is applied to the name, to determine the file's value, $\theta FILE$, which must not have expired. Its expiry time and contents are returned.

DeleteFile The *DeleteFile* operation removes a file from the service. A rebate is offered as an incentive to deletion before expiry. *name?* is the name of the file to be deleted, and *cost!* is the (possibly negative) charge made for doing so (we assume negation '−' is defined on *Money*). The cost is determined by a function *Rebate* of the file and its deletion time. Here is a possible definition of *Rebate*:

<i>Rebate</i> : <i>FILE</i> \times <i>Time</i> \rightarrow <i>Money</i>	_____
$Rebate = (\lambda FILE; when : Time \bullet (expires - when) * Size(contents))$	
<i>DeleteFile</i>	_____
ΔFS	
<i>name?</i> : <i>Name</i>	
<i>cost!</i> : <i>Money</i>	
<i>name?</i> $\in \text{dom } files$	
$(\exists FILE \bullet$	
$\theta FILE = files(name?) \wedge$	
$expires > when \wedge$	
$owner = who \wedge$	
$files' = \{name?\} \triangleleft files \wedge$	
$cost! = -(Rebate(\theta FILE, when))$	

The map *files* is applied to the name, to determine the file's value, $\theta FILE$, which must not have expired. It must be owned by the deleting client. The file's name, *name?* (and hence the file itself), are removed from the partial function which represents the stored files.

Naturally, there are other compromises that could be made, in addition to or instead of those above. In the next section, however, we discuss a compromise which we suggest should *not* be made.

9.4 A compromise avoided

One glaring inefficiency remains in our proposal: that we must transfer whole files at once. Many clients will not have the time or resources (e.g. local memory) to do this. But here we will not compromise by modifying our file storage service to cater for this inefficiency – rather we insist that the business of the file storage service will be file storage exclusively. Partial examination and updating will be the business of a file updating service.

To propose a service that treats the contents of files as having structure, we must propose a structure. The proposal we make is the very simple view that the contents of a file is a sequence of *pieces*. (Recall that sequences are functions from the natural numbers \mathbb{N} to their base type, and begin at index 1.) We do not wish to say what a piece is, however, for this description.

$$\begin{array}{l} [Piece] \\ Data == \text{seq } Piece \end{array}$$

The file updating service, in fact, has no state; all its work is done in the calculation of its outputs from its inputs. Its two operations are *ReadData* and *UpdateData*.

ReadData *ReadData* takes the contents of a file, *contents?*, a starting position, *start?*, and a number of pieces to be read, *number?*, and returns the data, *pieces!*, at that position within *contents?*. (*#pieces!* is the length of the sequence *pieces!*, and $1 \dots \#pieces!$ is the set $\{i : \mathbb{N} \mid 1 \leq i \leq \#pieces!\}$.)

$\begin{array}{l} \text{ReadData} \\ \hline contents? : Data \\ start?, \\ number? : \mathbb{N} \\ pieces! : Data \\ \hline \#pieces! = \min\{number?, (\#contents? - start?)\} \\ (\forall i : 1 \dots \#pieces! \bullet pieces!(i) = contents?(i + start?)) \end{array}$
--

The length of the data returned is equal to the number of pieces requested, if possible; otherwise, it is as large as the length of *contents?* will allow. The *i*th piece of *pieces!* returned is equal to the $(i + start?)$ th piece of *contents?*.

UpdateData *UpdateData* takes the contents of a file, *contents?*, a position, *start?*, and some data, *pieces?*, and returns an updated contents, *contents!*.

$ \begin{array}{l} \text{UpdateData} \\ \text{contents?}, \\ \text{contents!} : \text{Data} \\ \text{start?} : \mathbb{N} \\ \text{pieces?} : \text{Data} \\ \hline \# \text{contents!} = \max\{\# \text{contents?}, (\text{start?} + \# \text{pieces?})\} \\ \text{start?} \leq \# \text{contents?} \\ (\forall i : 1 \dots \# \text{contents!} \bullet \\ \quad (i - \text{start?}) \in 1 \dots \# \text{pieces?} \Rightarrow \\ \quad \quad \text{contents!}(i) = \text{pieces?}(i - \text{start?}) \wedge \\ \quad (i - \text{start?}) \notin 1 \dots \# \text{pieces?} \Rightarrow \\ \quad \quad \text{contents!}(i) = \text{contents?}(i)) \end{array} $

The length of the new contents is equal to its original length, unless an extension was necessary to accommodate the new data; however, the new data must begin within the original contents or immediately at its end. The i th piece of contents! is equal to the $(i - \text{start?})$ th piece of pieces? , if this is defined; otherwise, it is equal to the i th piece of contents? .

Our proposal is of course only one of the many possible (for a different proposal, see the definition of these operations in Chapter 4). We could, of course, propose *several* updating services, each providing its own set of facilities. Moreover, the original operations which transferred whole files would still be available to those clients able to use them (see Figure 9.1).

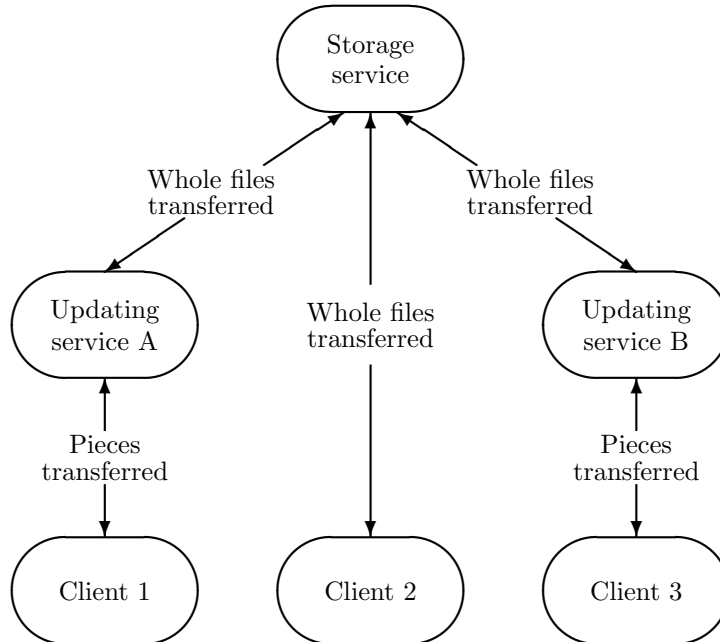


Figure 9.1: Separate updating and storage services

9.5 Modularity and composition of services

The structure we have presented above separates the issues of how files should be stored from how they should be manipulated. As a result, we have offered users an unusual freedom of choice – they can read just one piece of a file, or they can treat a file as a single object (with the corresponding conceptual simplification; Stoy and Strachey [59], for example, allow this in their operating system OS6).

Still, it is likely that a further compromise will be necessary: for large files, the time taken to transfer the file between the two services (storage and updating) may not be tolerable. We solve this not by changing our design, but by an engineering decision: for applications that require it, we will provide the two services together *in one box*, and the transfers will be internal to it (see Figure 9.2). Its specification we construct by combining the material already available.

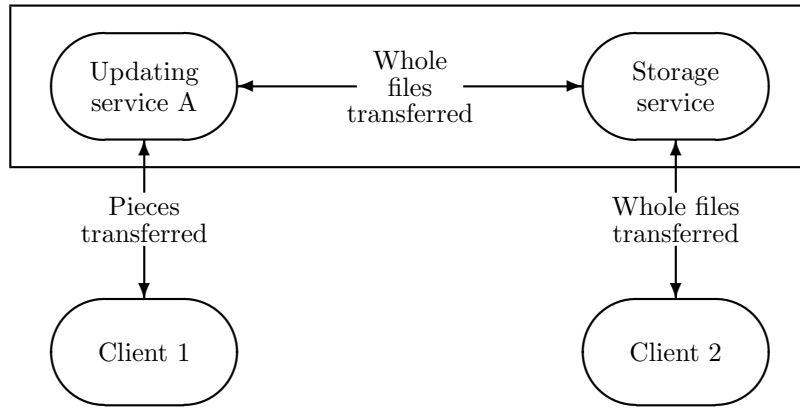


Figure 9.2: Combined updating and storage service

StoreFile, *ReadFile* and *DeleteFile* are available as before. However, we introduce two new operations, *ReadStoredFile* and *UpdateStoredFile*, whose specifications are formed by composing the specifications given above. (The schema piping operator ‘ \gg ’, used for this, is defined in the glossary.)

ReadStoredFile Reading a stored file is performed by first reading the whole file with *ReadFile*, and then reading the required portion of its contents using *ReadData*. In Z we write this

$$ReadStoredFile \hat{=} ReadFile \gg ReadData$$

If we were to expand this definition of *ReadStoredFile*, the result would be as follows:

<i>ReadStoredFile</i>	_____
ΔFS	
$name? : Name$	
$start?$	
$number? : \mathbb{N}$	
$expires! : Time$	
$pieces! : Data$	
$\theta FS' = \theta FS$	
$name? \in \text{dom } files$	
$(\exists FILE \bullet$	
$\theta FILE = files(name?) \wedge$	
$expires > when \wedge$	
$expires! = expires \wedge$	
$\#pieces! = \min\{number?, (\#contents - start?)\} \wedge$	
$(\forall i : 1.. \#pieces! \bullet pieces!(i) = contents(i + start?)))$	

ReadStoredFile takes a file name, $name?$, a starting position, $start?$, and a number of pieces, $number?$, and returns the expiry time of the file, $expires!$, and the data, $pieces!$, found at the position specified. ($expires!$ is returned by *ReadStoredFile* because *ReadFile* returns it; we could have dropped this extra output using the schema component hiding operator.)

UpdateStoredFile The complementary operation *UpdateStoredFile* is a more difficult composition, since we must accumulate the costs of the component operations, and we must ensure that the updated file is (re-)stored under its original name. For the sake of honesty, we give the definition, but we will not expand it.

$$\begin{aligned}
 \text{UpdateStoredFile} \hat{=} & \\
 & \text{ReadFile} \mathbin{\text{\&}} \\
 & \text{DeleteFile}[dcost!/cost!] \gg \\
 & \text{UpdateData} \gg \\
 & \text{StoreFile}[name?/name!, scost!/cost!] \mathbin{\text{\&}} \\
 & [dcost?, scost?, cost! : Money \mid cost! = dcost? + scost?]
 \end{aligned}$$

UpdateStoredFile first reads the whole file, then deletes it, then updates the contents, and then stores the new contents under the file's original name. Finally, it presents as its overall cost the sum of the two charges made by *DeleteFile* (which may well be negative) and *StoreFile*.

What we have done is to compose two simple but infeasible operations to produce a more complicated but feasible one (rather like the use of complex numbers in electrical engineering, for example). Naturally, the implementor need not transfer whole files back and forth within his black box on every read and update operation – but nevertheless the updating and storage service provided by the box *must* behave as if he does: that is, it must behave as we have specified. Our decomposition was chosen for economy of concept; the implementor's must be chosen for economy of time and equipment, and the whole range of engineering techniques are available to him to do so (caches, update-in-place, etc.).

9.6 Conclusions

The Distributed Computing Project has followed the general principles above, but it has in fact adapted to constraints in different ways. Its storage service, which we have implemented in prototype, stores *blocks* of a fixed size (rather like the service described by Biekert and Janssen [2]). This distinguishes it as a ‘universal’ storage service from, say, the one implemented at Cambridge (described by Needham and Herbert [47]). Organisation of blocks into files, the keeping of directories, etc., is done by software in the clients’ own machines (for example, using a ‘File Package’ as described by Gimson [12]). This allows clients freedom in the choice of what file structure they build, but of course makes the sharing of files more difficult. If one package should become popular, however, it could be placed in a machine of its own, and so become a service.

There are many aspects of the project that it has not been possible to cover. For example, the specification of the errors that may occur in use is an essential part of the full specification of a service. We include such details in the user manuals of the services we have implemented. The manuals follow the style of specification presented here, combining formal text and English narrative to give a precise yet easily understandable description of the user interface to a service.

So far, the pressure of simplicity in our mathematical descriptions has kept the designs correspondingly simple. At present, they are perhaps too much so; but by using mathematical specification techniques we have built basic services that genuinely *are* simple. And that is where one must begin.

The styles of specification, and of presentation of user manuals, have to some extent been developed in parallel with the software to which they have been applied. These styles are now more stable, and further services have been specified, designed and implemented in the same way.

The goal of the project was to produce a suite of designs from which implementations could be built on a variety of machines. Each design is documented, in a mathematical style, both for the user and for the implementor. Thus the primary goal is to construct a distributed system on paper.

For a paper construction to have any value, the designs proposed in it must be widely applicable, and genuinely useful. Machine-independent techniques of description take care of the first requirement. To ensure that the second is met, prototype implementations must be constructed of each of the designs, and experience must be gained of their use.

Chapter 10

Authentication of usernames

Roger Gimson and Carroll Morgan

10.1 Nicknames and usernames

As a short-term measure, a very simple scheme has been chosen to make it difficult for one client to impersonate another.

Each registered client has a *nickname* and a *username*. Nicknames are allocated from a set *Nickname*

$[Nickname]$

and the allocation is public – i.e. it is common for clients to know each others' nicknames. It is expected that nicknames will change only rarely, if at all.

Username are allocated privately, from a set *Client*; a client should not reveal his username to anyone else. Since usernames may become compromised (known by too many people) or forgotten (known by too few!), it might be necessary to change a client's username from time to time.

10.2 Authentication

Authentication is achieved by the existence of a (secret) partial function

$| \quad GetNickname : Client \rightarrow Nickname$

which gives for any username the nickname of the client who should be its sole possessor. Since the set *Client* of usernames has been made very large, and the set

$\text{dom } GetNickname$

of *authentic* usernames has been made a relatively small part of it, it will be hard for clients to guess the usernames of others. Services may therefore use the function *GetNickname* to authenticate their clients; they might reject requests for which

$client? \notin \text{dom } GetNickname$

10.3 Guest user

There is a *guest* username *GuestClient*, which some services might recognise as a special case. This username is public, and is expected to be used by clients temporarily without a private username of their own. It is guaranteed that the guest username is not the authentic username of any client.

$$\frac{}{GuestClient : Client}$$

$$GuestClient \notin \text{dom } GetNickname$$

Chapter 11

Time service – user manual

Roger Gimson and Carroll Morgan

11.1 Time service operation

The time service provides only one user operation, *GetTime*, which returns the current time *in seconds since 00:00:00 1 January 1980*. The description of the operation has three sections, titled *Abstract*, *Definition* and *Reports*.

The Abstract section gives a procedure heading for the operation, with formal parameters, as it might appear in some programming language. The correspondence between this procedure heading and an implementation of it in some real programming language must be obvious and direct. Each formal parameter is given a name ending with either ‘?’ or ‘!’. Those ending with ‘?’ are inputs, and those ending with ‘!’ are outputs.

The Definition section defines the meaning of the operation.

The Reports section lists the possible (success or failure reporting) values which the *report!* formal parameter can assume. Reports are discussed in more detail in Section 11.2.

GetTime

Abstract

```
GetTime ( client?  : Client;
          now!     : Time;
          cost!    : Money;
          report!  : Report)
```

Definition The current time, *now!*, is returned, measured in seconds from 00:00:00 1st January 1980. The cost, *cost!*, is fixed. The client’s username, *client?*, must be authentic (see Chapter 10).

$client? \in \text{dom } GetNickname$

Reports The following errors may be reported:

Success

ServiceError

11.2 Error reports

The report parameter, *report!*, indicates whether the operation succeeded or failed. The value *Success* indicates that the operation succeeded. The value *ServiceError* indicates that the operation failed; in this case, no reliance should be placed on any other values returned. Possible reasons for this report are the following:

- the service is not running; or
- there was a communication error.

11.3 Modula-2 interface

```

DEFINITION MODULE TI; (* Time Service - Modula-2 Interface *)
FROM SVTypes IMPORT Client, Time, Money;
EXPORT QUALIFIED Report, GetTime;
TYPE Report = ( Success, ServiceError);
PROCEDURE GetTime (InClient      : Client;
                   VAR OutNow    : Time;
                   VAR OutCost   : Money;
                   VAR OutReport : Report);
    (* return current time *)
END TI.
```

Chapter 12

Reservation service – user manual

Roger Gimson and Carroll Morgan

12.1 Introduction

The distributed operating system at the Programming Research Group is made up of various services which are largely independent. In particular, it is possible that one service can be turned on or turned off while other services and clients continue to run.

When a service is turned off (*shutdown*), there should not be any client who is at that moment involved in some *series* of interactions with it – because interruption of such a series could be quite inconvenient (for the client). If such series (or *transactions*) can be recognised by the service, it is possible to avoid this inconvenience by the following shutdown procedure.

1. The operator *requests* shutdown of the service.
2. The service rejects any attempt to begin a new transaction, but allows current transactions to continue.
3. When all transactions have completed, the service notifies the operator that shutdown is complete.

However, there are some problems; for example, clients might *themselves* fail to complete transactions (presumably due to accidental failure of their own software). If this happened, the service would *never* shutdown. A more serious problem is that for some services (e.g. the block storage service) there is no recognisable transaction structure, and so the above scheme cannot be used at all. We solve both problems with an independent *Reservation Service*.

The reservation service does not interact at all with the service it reserves; it interacts only with its own clients, and with the operator. It allows clients to state when, and for how long, they would like to use the reserved service, and it allows the operator to state a *shutdown time* beyond which all reservations are to be rejected.

It becomes the clients' responsibility to protect themselves from sudden shutdown of the service (by making reservations), and the operator's responsibility to turn off the service only after the shutdown time (which he or she may set). Thus a shutdown can be unexpected only by those clients who have made no reservation.

A typical use of the reservation service would be for clients to include a reservation request at the start of every program that uses the reserved service. The duration of the reservation should be long enough to allow the program to complete, but short enough to allow the operator to make a reasonably spontaneous decision to shutdown. The state of the reservation service has two components:

expires a map from clients' nicknames (their public identities – see Chapter 10) to the time at which their current reservation expires; and

shutdown the shutdown time most recently set by the operator.

<i>RS</i>
<i>expires</i> : <i>Nickname</i> \leftrightarrow <i>Time</i>
<i>shutdown</i> : <i>Time</i>

Each operation requested by clients includes the three values:

client? the username of the client;

cost! the cost of the operation; and

report! a report indicating whether the operation succeeded or failed.

Report ::= *Success* | *Service_Error* | *Not_Available* | *Too_Many_Clients*

ΔRS
<i>RS</i>
<i>RS'</i>
<i>client?</i> : <i>Client</i>
<i>nickname</i> : <i>Nickname</i>
<i>cost!</i> : <i>Money</i>
<i>report!</i> : <i>Report</i>
<i>client?</i> \in dom <i>GetNickname</i>
<i>nickname</i> = <i>GetNickname</i> (<i>client?</i>)

The username, *client?*, is supplied by the user; it is his or her *private* username (as distinct from his or her public nickname). *client?* must be authentic, i.e.

client? \in dom *GetNickname*

if the service is not to ignore the request.

The client's nickname is calculated by the service. *cost!* and *report!* are returned to the user by the service.

12.2 Reservation service operations

Three operations are described in this section: *Reserve*, which is requested by clients; *SetShutdown*, which is requested by the operator; and *Scavenge*, which is performed by the service itself (at its discretion). The latter two operations are included here only as an aid to the reader's intuition.

The description of each operation can have up to four sections, titled *Abstract*, *Definition*, *External Calls* and *Reports*.

The Abstract section gives a procedure heading for the operation, with formal parameters, as it might appear in some programming language. The correspondence between this procedure heading and an implementation of it in some real programming language must be obvious and direct. Each formal parameter is given a name ending with either '?' or '!'. Those ending with '?' are inputs, and those ending with '!' are outputs. A short description may accompany the procedure heading.

The Definition section mathematically defines the operation, by giving a schema that includes as a component every formal parameter of the procedure heading. Within the schema appear also subschema(s) whose components include the service state before and after the operation (this can be more (RS, RS') or less (ΔRS) explicit). Any other components appearing in the schema are either local to the operation (i.e. temporary) or represent values exchanged with other services (invisibly to the client). Only the formal parameters of the procedure heading are exchanged directly between client and service. A short description may accompany the schema.

The External Calls section lists the calls that this service may make on other services, in order to complete the requested operation. These appear as procedure calls which match the procedure headings given in the description of the operation called. (These are found in the user manual for the called service.) The correspondence between formal and actual parameters is positional, with missing (i.e. irrelevant) actual parameters indicated by commas.

The Reports section lists the possible (success or failure reporting) values that the formal parameter *report!* can assume. If such a value is followed by a predicate, it is to suggest that the reported value would occur *because* that predicate is satisfied. The predicate is therefore a hint to the cause of the report.

Reports are discussed in more detail in Section 12.3.

Reserve

Abstract

```
Reserve ( client?   : Client;
          interval? : Interval;
          until!    : Time;
          cost!     : Money;
          report!   : Report)
```

A reservation is made for a period of *interval?* seconds. *until!* returns the expiry time of the new reservation.

Interval == Time

A client can cancel his reservation by making a new reservation in which *interval?* is zero; see *Scavenge* below. There is a fixed cost for making a reservation

| *ReservationCost* : *Money*

Definition

<i>Reserve</i>	
ΔRS	
<i>interval?</i> : <i>Interval</i>	
<i>until!</i> ,	
<i>now</i> : <i>Time</i>	
<i>until!</i> = <i>now</i> + <i>interval?</i>	
<i>until!</i> ≤ <i>shutdown</i>	
<i>expires'</i> = <i>expires</i> ⊕ { <i>nickname</i> ↦ <i>until!</i> }	
<i>shutdown'</i> = <i>shutdown</i>	
<i>cost!</i> = <i>ReservationCost</i>	

The reservation must expire before the shutdown time. The current time *now* is obtained from the time service (see Chapter 11).

External Calls

Time service

GetTime (, *now*,, *Success*)

now is obtained by a successful call of *GetTime*. It is measured in seconds from 00:00:00 1st January 1980.

Reports The following errors may be reported.

<i>Success</i>	
<i>Service_Error</i>	
<i>Not_Available</i>	⇒ <i>shutdown</i> < <i>now</i> + <i>interval?</i>
	<i>shutdown</i> = <i>until!</i>
<i>Too_Many_Clients</i>	⇒ # <i>expires</i> = <i>Capacity</i>

SetShutdown

Abstract

SetShutdown (*shutdown?* : *Time*;
 threatens! : *Boolean*)

The operator may set a new shutdown time. He or she is informed if the new time threatens existing reservations; if it does, it is his or her responsibility to negotiate with the clients affected.

Definition

<i>SetShutdown</i>	_____
<i>RS</i>	
<i>RS'</i>	
<i>shutdown?</i> : <i>Time</i>	
<i>threatens!</i> : <i>Boolean</i>	
<i>shutdown'</i> = <i>shutdown?</i>	
<i>threatens!</i> = <i>True</i> $\Leftrightarrow (\exists \text{ expiry} : \text{ran expires} \bullet \text{expiry} > \text{shutdown'})$	
<i>expires'</i> = <i>expires</i>	

The shutdown time is changed to the new value *regardless of existing reservations*. Reservations are unaffected.

Scavenge**Abstract****Scavenge()**

The service can at any time remove reservations whose expiry time has passed. This is in fact the *only* way in which reservations are removed (by client, operator or service).

Definition

<i>Scavenge</i>	_____
<i>RS</i>	
<i>RS'</i>	
<i>now</i> : <i>Time</i>	
<i>shutdown'</i> = <i>shutdown</i>	
<i>expires'</i> \subseteq <i>expires</i>	
$(\forall \text{ removed} : \text{dom expires} \bullet$ $\text{removed} \notin \text{dom expires}' \Rightarrow \text{expires}(\text{removed}) \leq \text{now})$	

Scavenge does not change the shutdown time.

Scavenge can remove reservations, but it never makes new ones. A reservation is removed only if its expiry time has passed.

External Calls**Time service**

GetTime (, now,, Success)

now is obtained by a successful call of *GetTime*. It is measured in seconds from 00:00:00 1st January 1980.

12.3 Error reports

The *report!* parameter of each operation indicates either that the operation has succeeded or suggests why it failed; in most cases, failure leaves the service unchanged.

An operation can return only the report values listed in the reports section of its description. If it returns the value *Success*, it must satisfy its defining schema. If it returns any other value, it must satisfy instead the appropriate schema below.

ServiceError *ServiceError* indicates an unexpected failure, which might not be the client's fault.

<i>ServiceError</i>	_____
<i>RS</i>	
<i>RS'</i>	
<i>report!</i> : <i>Report</i>	
<i>report!</i> = <i>Service_Error</i>	

These are typical causes:

- service not running;
- network (hardware or protocol) failure;
- service hardware fault;
- service software error.

NotAvailable

<i>NotAvailable</i>	_____
ΔRS	
<i>interval?</i> : <i>Interval</i>	
<i>until!</i> ,	
<i>now</i> : <i>Time</i>	
<i>report!</i> = <i>Not_Available</i>	
<i>shutdown</i> < <i>now</i> + <i>interval?</i>	
<i>until!</i> = <i>shutdown</i>	
$\theta RS' = \theta RS$	

If the reservation cannot be made due to early shutdown, the shutdown time itself is returned in *until!*. *now* is obtained from the time service.

TooManyClients The service has finite capacity

| *Capacity* : \mathbb{N}

for recording reservations. This report occurs when that capacity would be exceeded.

<i>TooManyClients</i>	_____
ΔRS	
<i>now</i> : <i>Time</i>	
<i>report!</i> = <i>Too_Many_Clients</i>	
<i>#expires</i> = <i>Capacity</i>	
<i>nickname</i> \notin dom <i>expires</i>	
$\theta RS' = \theta RS$	

The report *cannot* occur if the client has a reservation (since it is overwritten by the new one).

now is obtained from the time service.

Clients who cannot themselves make reservations might be able to rely temporarily on the reservations of others.

12.4 Modula-2 interface

DEFINITION MODULE RI;

(* Reservation Service - Modula-2 Interface *)

FROM SVTypes IMPORT Client, Time, Interval, Money;

EXPORT QUALIFIED Report, Reserve;

TYPE Report = (Success,
 ServiceError,
 NotAvailable,
 TooManyClients);

PROCEDURE Reserve (InClient : Client;
 InInterval : Interval;
 VAR OutUntil : Time;
 VAR OutCost : Money;
 VAR OutReport : Report);

(* reserve use of the service for InInterval, terminating at
OutUntil, otherwise return the time at which the service
becomes unavailable in OutUntil *)

END RI.

Part IV

TRANSACTION PROCESSING

The chapters presented in this part were produced as part of a joint project between IBM (UK) Laboratories at Hursley, England and the Programming Research Group of the Oxford University Computing Laboratory into the application of formal software specification techniques to transaction processing.

The project was initiated in response to a talk by Professor C. A. R. Hoare to the British Computer Society [19], from which the following quotation is extracted.

I believe that the time has come to attempt to scale up the use of formal mathematical methods to industrial application. This can best be achieved by collaborative development projects between a university or polytechnic and an industrial company or software house.

The talk was attended by Tony Kenny from IBM Hursley and this led to the initial collaboration between Hursley and Oxford.

The initial project consisted of specifications of parts of the IBM Customer Information Control System (CICS). Chapter 13 describes the work carried out. A number of modules of the CICS command level application programmer's interface were specified; these include the CICS Exception Handling, which is documented within Chapter 13, and CICS Temporary Storage, which is described in Chapter 16. Chapter 17, on the CICS Message System, was later work not directly related to the other papers.

The work documented here was supported by research contract between IBM (UK) Laboratories and Oxford University and is published by kind permission of the Company. Carroll Morgan gave much needed and appreciated critiques of the specifications. The project was lucky to have had Rod Burstall, Tony Hoare and Cliff Jones as consultants. Peter Collins and John Nicholls have been responsible for the project from the IBM end and Ib Holm Sørensen was responsible for setting up the project from the Oxford end; the work reported here owes much to his guidance throughout the project.

For the second edition two new chapters (14 and 15) have been added to the book. Chapter 14 provides an update on the use of Z in part of the restructuring of IBM's CICS. For those interested in the experience of using such methods in practice, this chapter gives an update on one of the largest and longest running industrial projects making use of Z.

Chapter 15 gives an overview of a more recent project undertaken at IBM Hursley to specify the CICS Application Programming Interface. This work can be viewed as an extension of the work presented in Chapter 13. However, it is on a larger scale. Chapter 15 reports on the progress made on some of the problems outlined in Chapter 13.

Chapter 13

Applying formal specification to the development of software in industry

Ian Hayes

Abstract This chapter reports experience gained in applying formal specification techniques to an existing transaction processing system. The system is the IBM Customer Information Control System (CICS) and the work has concentrated on specifying a number of modules of the CICS application programmer's interface.

The uses of formal specification techniques are outlined, with particular reference to their application to an existing piece of software. The specification process itself is described and a sample specification presented.

One of the main benefits of applying specification techniques to existing software is that questions are raised about the system design and documentation during the specification process. Some problems that were identified by these questions are discussed.

Problems with the specification techniques themselves, which arose in applying the techniques to a commercial transaction processing system, are outlined.

13.1 Introduction

Oxford University and IBM (UK) Laboratories Limited are engaged in a joint project to evaluate the applicability of formal specification techniques to industrial scale software. The project is attempting to scale up formal mathematical methods, used so far within a research environment, to large-scale software in an industrial environment. This chapter reports the experience gained so far in applying these techniques to describe the application programmer's interface of the IBM Customer Information Control System (CICS).

Copyright © 1985 IEEE. Reprinted, with permission, from *IEEE Transactions on Software Engineering*, SE-11(2), pp. 169–178, February 1985.

CICS is widely used to support online transaction processing applications such as airline reservations, stock control and banking. It can support applications involving large numbers of terminals (thousands) and very large data bases (requiring gigabytes). The CICS General Information manual [30] gives the following description.

CICS/VS provides (1) most of the standard functions required by application programs for communication with remote and local terminals and subsystems; (2) control for concurrently running user application programs serving many online users; and (3) data base capabilities ...

CICS is general purpose in the sense that it provides the primitives of transaction processing. An individual application is implemented by writing a program invoking these primitives. The primitives are similar to operating system calls, but are at a higher level; they also provide such facilities as security checking, transaction logging, and error recovery.

CICS has been in use since 1968, and has undergone continuous development during its lifetime. In the original implementation, the application programmer's interface was at the level of control blocks and assembly language macro calls. This is referred to as the *macro*-level application programmer's interface. In 1976 a new interface, the *command*-level application programmer's interface, was introduced. It provides a cleaner interface which does not require the application programmer to have knowledge of the control blocks used in the implementation of the system. The command-level interface is the subject of our work on specification.

CICS is supported on a number of IBM operating systems in such a way that application programs written using the application programmer's interface may be transferred from one environment to another without recoding. In addition, the command-level interface supports a number of programming languages: PL/I, Cobol, Assembly language and RPG II. This is achieved by the use of a preprocessor that translates programs containing CICS commands into the appropriate statements in the language being used (usually a call on a CICS module). Hence the application programmer's interface provides a level of abstraction that hides a number of significantly different implementations.

The command level interface is split up into a number of relatively independent modules responsible for controlling various resources of the system. The formal specification work has so far concentrated on specifying individual modules in relative isolation. Of the sixteen modules comprising the command-level interface, three — temporary storage, exceptional condition handling and interval control — have been specified. Temporary storage provides facilities for setting up named temporary storage *queues* that may be used to communicate information between transactions or as temporary storage by a single transaction. Exceptional condition handling provides facilities to handle exceptions raised by calls on CICS commands in a manner similar to PL/I condition handling. Interval control provides facilities to set up time-outs and delays, as well as to start a new transaction at a given time and to pass data to it.

With the large number of CICS systems around the world, the usage of the CICS command level application programmer's interface is on a par with many programming languages. As with programming languages, it is important that the interface be clearly specified in a manner independent of a particular implementation.

13.2 Uses of formal specification

The work reported in this chapter deals with the specification of parts of an existing system. Before considering the benefits of specification when applied to existing software we will briefly review the benefits of specification in general. (For a more detailed discussion see [58].) In software development a formal specification can be used by

- designers** to formulate and experiment with the design of the system,
- implementors** as a precise description of the system being built, particularly if there is more than one implementation,
- documentors** as an unambiguous starting point for user manuals, and
- quality control** for the development of suitable validation strategies.

Using a specification, the designer of a system can reason about properties of the system before development starts; and during development, formal verification that an implementation meets its specification can be carried out.

When an existing system is being specified there are both short- and long-term benefits. In the short term, performing the specification

- uncovers those parts of the existing manuals that are either incomplete or inconsistent; and
- gives insights into anomalies in the existing system and can suggest ways in which the system could be improved.

In the longer term the specification can be used

- for reimplementation of all or part of the system;
- as a basis for discussing and developing specifications for changes or additions to the system; and
- to provide a model of the functional behaviour of the system suitable for educating new staff.

Reimplementation may involve a new machine architecture, programming language or operating system, or a restructuring to take advantage of multiprocessor or distributed systems. As the specification is implementation independent, it provides a suitable starting point for each of the above alternatives.

When changes or additions to the system are to be made, new specifications can be developed with reference to the previous specification. These developments will give insights into the effect of the changes and their interaction with existing parts of the system.

As the specification is a formal document it provides a more precise description for communication between the designers than natural language descriptions. This should help to reduce misunderstandings among the people involved.

Experimentation with specification provides a quicker and cheaper method of investigating a number of alternative changes to the system than implementing the changes. On the other hand, because the specification is implementation independent, it cannot provide direct answers to questions of how difficult the changes will be to

implement, or their impact on the performance of the system. However, as it is at a high level of abstraction it can give a better insight into the interaction of changes with other components of the system; it is just these high-level interactions which get lost in informal specifications and in the detail of implementation.

While working predominantly at a more abstract level the specifiers must be experienced in implementation and should be aware of the implementation consequences of their decisions. Those parts of the specification for which the implementation consequences are unclear should be further investigated before detailed implementation is begun.

13.3 The specification process

The starting point for our specification work was the CICS command-level application programmer's reference manual [29]. The style of this manual is a combination of formal notation describing the syntax of commands and informal English explanations of the operation of the commands. We developed our initial specification of a module of the system by reference to the corresponding section of the manual. The main goal was to come up with a mathematical model of the module that is consistent with its description in the manual. This involves forming a crude initial model of the module and extending it to cover operations (or facets of operations) not initially dealt with, or refining or redesigning the specification as inconsistencies are discovered between it and the manual.

In attempting the initial specification, questions arose that were not satisfactorily answered by the manual. At this stage, a list of questions was prepared, and an expert on that module of the system (along with the source code) was consulted. Questions can arise for the following reasons:

- the manual is incomplete or vague;
- the manual is not explicit as to whether *possible* special cases are treated normally or not;
- the manual is itself inconsistent; or
- the chosen mathematical model is inconsistent with the manual in some small way: either the model or the manual is incorrect.

As the system has been in use for some time the answers to the more straightforward questions about its operation have already found their way into the manual. Hence most questions that arose in the specification process were rather subtle and required reference to the source code of the module to be satisfactorily answered. Some of the questions led to inconsistencies being discovered between the manual and the implementation. These inconsistencies were either errors in the manual or bugs in the implementation. Which way they should be classified depends on the original intent of the designer.

The specification was also given to people experienced in formal specification who gave comments on its internal consistency and style, and who suggested ways in which the specification could be simplified or improved. They were also given a copy of the relevant section of the manual to read *after* they had understood the specification, and were asked to point out any inconsistencies they discovered between it and the specification.

The answers to questions and the review of the specification led to a revision of the specification, which led to further questions and further review, and so on.

13.3.1 Notation

The style of the specification document is a mixture of formal Z and informal explanatory English. The formal parts of the specification, given in Z, are surrounded in the text by boxes so that they stand apart from the explanatory surrounds and may be more easily found for reference purposes. To make a specification readable, both formal and informal parts are necessary; the formal text can be too terse for easy reading and often its purpose needs to be explained, while the informal natural language explanation can more easily be vague or ambiguous and needs the precision of a formal language to make the intent clear. The informal text provides the link between formality and reality without which the formal text would just be a piece of mathematics. To create a good specification the structuring of the specification and the composition and style of the informal prose are as important as the formal text.

The aim is to provide a specification at a high level of abstraction and thus avoid implementation details. The specification should reveal the operation of the system a small portion at a time. These portions can be progressively combined to give a specification of the whole. This style of presentation is preferred to giving a monolithic specification and trying to explain it; the latter can be rather overwhelming and incomprehensible because there are too many different facets to understand at once. It is hoped that by giving the specification in small portions each piece can be understood, and when the pieces are put together the understanding of the parts that has already been gained can lead more easily to an understanding of the whole.

For more complex specifications that are developed via numerous small steps, understanding the whole can be quite difficult, because one needs to remember the function of all the parts and understand the way in which they are combined. In such cases it can be useful to provide both a portion by portion development of the specification and an expanded monolithic specification as well. The latter is more assailable after one has been through a piece-by-piece development and has an understanding of its various components.

13.4 A sample specification

As a sample of the type of specification produced we will look in detail at the specification of exceptional condition handling within CICS. The exception check mechanisms of CICS are similar to those provided by PL/I [28]. This module was chosen for exposition because it is one of the smaller modules in the system. The manual entry on which the specification was initially based is given in Appendix 13.8. The specification given here is the final product of the specification process described in the previous section.

The *syntax* of the CICS commands depends, of course, on the environment in which they are written. Our notation below is intended to be uncommitted, but explicit enough to indicate exactly which command is meant.

13.4.1 Exceptional conditions specification

Exceptional conditions may arise during the execution of a CICS command. A transaction may either set up an action to be taken on a condition by using a *Handle Condition* command, or it may specify that the condition is to be ignored by using an *Ignore Condition* command. If a condition has been neither handled nor ignored, then the default action for that condition is used. For example, to handle condition x with action y we can use

$$\textit{Handle Condition}(c = x, a = y)$$

where the keyword parameter ' $c =$ ' gives the condition and ' $a =$ ' gives the action. To ignore condition z we use

$$\textit{Ignore Condition}(c = z)$$

We introduce the set *CONDITION*, which contains all the exceptional conditions that may occur, and also contains two special conditions:

success the condition that indicates that a command completed normally, and

error this is not a condition that can arise from the execution of a command; rather, it provides a mechanism for providing a catchall error handler for conditions that are not explicitly handled.

We do not list all the possible exceptional conditions here.

$$\textit{CONDITION} ::= \textit{success} \mid \textit{error} \mid \dots$$

We also introduce the set *ACTION*, which contains all actions that could be taken in response to some exceptional condition. The exact nature of *ACTION* is not discussed in detail here. For each programming language supported by CICS it has a slightly different meaning, but for all of the languages an action is represented by a label which is given control. There are four special actions used in this specification:

nil indicating a normal return (i.e. no action);

abort the action that abnormally terminates a transaction;

wait indicating that the transaction is to wait until the operation can be completed normally (e.g. wait until space becomes available); and

system used to simplify the specification of the *Handle Condition* command.

$$\textit{ACTION} ::= \textit{nil} \mid \textit{abort} \mid \textit{wait} \mid \textit{system} \mid \dots$$

13.4.2 The state

The state of the exception controlling system can be defined by the following schema:

<i>Exceptions</i>	
$\textit{Handler} : \textit{CONDITION} \leftrightarrow \textit{ACTION}$	
$\textit{Handler}(\textit{success}) = \textit{nil}$	

The mapping *Handler* gives the action to be taken for those conditions that have been set up by either an *Ignore Condition* or *Handle Condition* command. The handling action for condition *success* is always *nil* (i.e. return normally). The action for other conditions is determined by some fixed function

$$\begin{array}{|l} \hline \text{Default} : \text{CONDITION} \rightarrow \text{ACTION} \\ \hline \text{Default}(\text{error}) = \text{abort} \wedge \\ \text{ran}(\text{Default}) = \{\text{nil}, \text{abort}, \text{wait}\} \end{array}$$

The default action for the special condition *error* is to *abort* and the only default actions are *nil*, *abort*, and *wait*.

The initial state of the exception handling system for a transaction is given by the following schema:

$$\begin{array}{|l} \hline \text{Initial} \\ \hline \text{Exceptions} \\ \hline \text{Handler} = \{\text{success} \mapsto \text{nil}\} \\ \hline \end{array}$$

The initial state of the handler is to return normally if the operation completes successfully. As an example, if starting in the initial state the commands

$$\begin{array}{l} \text{Handle Condition}(c = x, a = y) \\ \text{Ignore Condition}(c = z) \end{array}$$

are executed, then the final state will satisfy

$$\text{Handler} = \{x \mapsto y, z \mapsto \text{nil}, \text{success} \mapsto \text{nil}\}$$

The *Handle Condition* command sets up a mapping from condition *x* to action *y* and the *Ignore Condition* command maps condition *z* onto the *nil* action.

13.4.3 The operations

The two operations, *Handle Condition* and *Ignore Condition*, work directly on the above state. We describe a state change using the following schema, which is called ' $\Delta\text{Exceptions}$ ':

$$\begin{array}{|l} \hline \Delta\text{Exceptions} \\ \hline \text{Exceptions} \\ \text{Exceptions}' \\ \hline \end{array}$$

Exceptions represents the state of the exception handling system before an operation and *Exceptions'* the state after.

The operation *Handle Condition* is used to set up the action, *a?*, to be performed on a particular exceptional condition, *c?*. It is defined by the following schema:

$$\begin{array}{|l} \hline \text{HandleCondition} \\ \hline \Delta\text{Exceptions} \\ c? : \text{CONDITION} \\ a? : \text{ACTION} \\ \hline c? \neq \text{success} \wedge a? \notin \{\text{nil}, \text{abort}, \text{wait}\} \wedge \\ \text{Handler}' = \text{Handler} \oplus \\ \{c? \mapsto (\text{if } a? = \text{system} \text{ then } \text{Default}(c?) \text{ else } a?)\} \\ \hline \end{array}$$

The first predicate gives the precondition for the operation: the special condition *success* cannot be handled, and the special actions *nil*, *abort* and *wait* cannot be given as handling actions. The second predicate describes the effect of the operation: if the action to be set up is specified as *system*, then, instead, the default action for the given condition will be set up as the handler for that condition; otherwise the supplied action, *a?*, will be set up. For example, if the command

$$\text{Handle Condition}(c = x, a = \text{system})$$

is executed in the initial state and $\text{Default}(x) = \text{wait}$, the resulting state will satisfy

$$\text{Handler} = \{x \mapsto \text{wait}, \text{success} \mapsto \text{nil}\}$$

The actual *Handle Condition* command accepts a set of condition–action pairs, rather than just a single pair as shown above. However, the effect of the command for each pair is as described above, so we will not bother to show the full command. Similarly, the *Ignore Condition* command accepts a set of conditions, but we only bother to show its effect for a single condition here.

The operation to specify that an exceptional condition is to be ignored is given by the following schema:

$\begin{array}{l} \text{IgnoreCondition} \\ \hline \Delta \text{Exceptions} \\ c? : \text{CONDITION} \\ \hline c? \neq \text{success} \\ \text{Handler}' = \text{Handler} \oplus \{c? \mapsto \text{nil}\} \end{array}$

The special condition *success* cannot be specified in an *IgnoreCondition* command. The action to be taken on an ignored condition is to return normally (i.e. *nil*).

13.4.4 Exception checking

Exception handling can take place on any CICS command except *HandleCondition* and *IgnoreCondition* themselves. We need to describe the exception checking that takes place on all other commands. The exception checking process determines the action, *a!*, to be taken on completion of a command. The value of *a!* is dependent on the condition, *c?*, returned by the command, and the current state of the exception handling mechanism. In addition, any command may specify whether or not all exceptions are to be handled for the execution of just that command. In describing the checking process we include the Boolean variable *handle?* to indicate this. The following defines the (complex) exception checking mechanism that is included in the definition of each operation (other than *Handle Condition* and *Ignore Condition*):

ExceptionCheck

*Exceptions**handle?* : *Boolean**c?* : *CONDITION**a!* : *ACTION*

```

a! = if handle? = False then nil
      else if c? ∈ dom Handler then Handler(c?)
      else if Default(c?) ≠ abort then Default(c?)
      else if error ∈ dom Handler then Handler(error)
      else abort

```

If exceptions are not being handled for the command (*handle?* = *False*) the action is to return normally; otherwise the action is determined from the exception handler. If the condition, *c?*, has been ignored or handled (including the case where the handle action was specified as *system*) then the corresponding handler action is used. Otherwise, if the default action for the condition is not *abort* the default is used, else if the special condition *error* is handled its handler action is used, otherwise the action is *abort*.

13.5 Questions raised

The questions raised about the system during the specification process are an important benefit of the process. They indicate problems either in the documentation of the system or in its logical design, and provide those responsible for maintaining the system with immediate feedback on problem areas.

In writing a formal specification one is creating a mathematical model of what is being specified, and in creating such a model one is encouraged to be more precise than if one were writing in a natural language. Because of the precision required, questions are raised during the specification process that are not answered by referring to the less formal manual. In fact, the task of formal specification is demanding enough to raise most of the questions about the functional behaviour of the system that would be raised by an attempt to implement it. The effort required for a specification, however, is considerably less than that required for an implementation.

We now discuss some of the questions that were raised during the specification work on CICS modules. It is interesting to note that most of the questions raised required the expert on the module to refer to the source code to give a conclusive answer. We begin with the questions about exceptional conditions, then a question about interval control, and finally a question about the interaction between temporary storage and exceptional conditions.

13.5.1 Exceptional conditions

We first list some questions that were raised during the specification of exceptional condition handling and then examine one of the more interesting questions in detail. All of these questions were resolved in producing the specification given in the previous section.

1. What is the range of possible default actions?

2. Is the default action for a particular condition the same for all commands that can raise that condition?
3. Can the special condition *error* be ignored?
4. Is the action for condition *error* only used if the default system action on a condition is *abort*?
5. If executed from the initial state, does the sequence

$$\begin{array}{l} \text{Handle Condition}(c = x, a = y) \\ \vdots \\ \text{Handle Condition}(c = x, a = \text{system}) \end{array}$$

return the handler to the initial state?

The reader is invited to try to answer these questions from the manual entry given in Appendix 13.8 and then from the specification given in Section 13.4.1. We now look in detail at question 5 above. It shows a subtle operation of the exceptional conditions mechanism that is counter-intuitive.

In an earlier model of the *Handle Condition* command the new value for the *Handler'* in the case when $a? = \text{system}$ was

$$\text{Handler}' = \{c?\} \triangleleft \text{Handler}$$

That is, if the action specified as an input is *system* then the entry for the condition $c?$ is removed from the handler ($c? \notin \text{dom Handler}'$). In the final model the new value of the *Handler'* in this case is

$$\text{Handler}' = \text{Handler} \oplus \{c? \mapsto \text{Default}(c?)\}$$

In this version, if the action is *system* the entry in the handler for condition $c?$ is set up to be $\text{Default}(c?)$ (therefore $c? \in \text{dom Handler}'$).

To see the effect of the difference we need to look at the *Exception Check* mechanism given in Section 13.4.1. If we use the second line above, then the action when the exception $c?$ occurs is $\text{Default}(c?)$ (assuming *handle?* is true). In the earlier model, however, the action also depends on whether a handler has been set up for the special condition *error*: the action is $\text{Default}(c?)$ unless $\text{Default}(c?)$ is *abort* and $\text{error} \in \text{dom Handler}$, in which case the action is $\text{Handler}(\text{error})$. The difference between the two versions is subtle and the reader is encouraged to study the definitions of *Handle Condition* and *Exception Check* in order to understand the difference.

The exception check mechanism is quite complex. None of the people experienced with CICS who were questioned about exceptional condition handling was aware of the problem detailed above. It is interesting to conjecture why this is so. The most plausible explanation is that the operation of the exception check mechanism is counter-intuitive. For example, the sequence given in question 5, i.e.

$$\begin{array}{l} \text{Handle Condition}(c = x, a = y) \\ \vdots \\ \text{Handle Condition}(c = x, a = \text{system}) \end{array}$$

does not leave the exceptional condition handler in its initial state if the default action for condition x is *abort* and a handler has been set up for the special condition *error*;

before the above sequence the *error* handler is used on an occurrence of condition x , but after, the action *Default*(x) (i.e. *abort*) is used on an occurrence of x .

If the above sequence did restore the exception condition handler to its initial state, then it could be used to handle condition x temporarily for the duration of the statements between the *Handle Condition* commands. This form of operation is more what those using the exceptional conditions module expect.

The *Exception Check* mechanism is so complex that most readers of either the manual or the specification given in the previous section do not pick up the above subtle operation unless it is explicitly pointed out in some form of warning. This is probably a good argument in favour of revising exception handling so that it becomes more intuitive.

The discussion about question 5 above also raises the point that a specification can be incorrect. This case shows one advantage of getting a second opinion on the specification and how it compares with the manual, from a person experienced in formal specification. It is important that the reviewer should read the specification before reading the manual. The reviewer's mental model of the system is thus based on the mathematical model in the specification. When the reviewer reads the manual looking for inconsistencies with the specification, any questions that arise can be answered by consulting the precise model given in the specification. This contrasts with the person writing the specification who forms a model from the manual and often has to consult other sources to answer questions that arise. Getting a second opinion on the specification and how it compares to the manual is an important ingredient for increasing confidence in the accuracy and readability of the specification.

13.5.2 Interval control

As another example we consider one of the problems raised during the specification of the CICS interval control module. Interval control is responsible for operations that deal with the interval timer. The operations provided by interval control can be split logically into two groups: those concerned with starting new transactions at specified times, and those concerned with time-outs and delays.

In specifying a module of the system we define the state components of the module (in the case of exceptional conditions there was only one state component, *Handler*). The state components of interval control can be split into two groups that are concerned respectively with the two groups of interval control operations. For the most part, operations only refer to or change components of the corresponding state. One exception is the command *Start* (to start a new transaction) which in some circumstances changes the time-out state components. This can be considered to be a carefully documented anomaly of the current implementation. Both the implementation and documentation could be simplified if the *Start* command did not destroy the current time-out. More importantly, removal of this interaction would lead to a more useful time-out mechanism, because time-outs would not be affected by a transaction start.

This anomaly is interesting because it points out an unwanted interaction between different parts of a module. In attempting to write the specification this interaction stood out because it involved the *Start* operation using the time-out state. This form of interaction between parts of modules tends to be pinpointed in the formal specification process because the offending operations require access to state information other than that of the part to which they belong.

Two further facts reinforce the view that the current operation of the *Start* command is not the most desirable: if the new transaction is to be started on a different computer system to the one issuing the *Start* command, or if the start is protected (from the point of view of recovery on system failure), then the start does not destroy the current time-out. Ideally we do not want to have to specify distributed system and recovery effects individually with each operation. We would like to add extra levels of abstraction to describe these effects for the whole system.

13.5.3 Interaction between modules

As an example of an interaction between two CICS modules we consider an interaction between exceptional conditions and temporary storage. When temporary storage is exhausted it can raise the exceptional condition *nospace*. This is processed in the normal way if it has been explicitly handled; the default action, however, is to wait until space becomes available.

Thus the specification of the temporary storage operations that can lead to a *nospace* exception require access to the exceptional conditions state to determine whether or not the *nospace* exception is handled; if it is handled it can occur, but if it is not, it cannot. These operations would more simply be specified (and implemented) if they had an extra parameter indicating whether or not to wait. It is interesting to note that, in the implementation, such temporary storage commands are transformed into a call with an additional parameter after the exception handling state has been consulted. It is also interesting that these commands were not correctly implemented if the *nospace* exception was ignored.

Interactions between modules of the system are pinpointed during the formal specification process (just as they would be in an implementation) because an operation from one module needs access to the state components of another. Any such interactions discovered during the specification process should be examined closely as they may indicate a breakdown in the modular structure of the system.

13.6 Problems with specification

In this section we examine the problems encountered in applying the formal specification techniques. This is in contrast to the previous section, in which we concentrated on the system being specified. The problems encountered in applying specification techniques can be split into the following categories:

- communication problems between the people involved;
- the general problem of achieving the ‘right’ level of abstraction in the specification; and
- more technical problems related to the particular specification technique.

13.6.1 Communication problems

As a specification group from a university working with a commercial development laboratory we faced a communications problem. Each party has its own language: the specifiers use the language of mathematics based on set theory, while the developers use terminology and concepts specific to the system which they are developing. The

communication problem is in both directions. This requires that each party learn the language of the other.

In performing a formal specification the specifier needs to understand what is being specified in order to be able to develop a mathematical model of it. To understand the system it is necessary to read manuals and consult experts, both of which use IBM and CICS terminology. Once a specification is written, the specifier would like to get feedback on its suitability from these same experts. This requires that they need to be educated in mathematics to a level at which they can understand a specification. At the current stage of the project the educational benefit has been more to the advantage of the specifiers learning about the system. In performing a specification of part of a system the specifier, of necessity, becomes an expert on the functional behaviour of that part (but not on the implementation of the part).

13.6.2 The right level of abstraction

In this context ‘right’ means that a piece of specification conveys the primary function of the part of the system it specifies and is not unduly cluttered with details. It is most important that a specification should not be biased towards a particular implementation. However, getting the right specification also involves choosing the most appropriate model and structuring the specification so that the minute details of the specified object do not obscure the primary function.

We can use hierarchical structuring to achieve this. Details of some facet of a component can be specified separately and then that specification can be referred to by the higher level specification. Different cases of an operation (e.g. the normal case and the erroneous case) can be specified independently and combined to give a specification of the whole.

The structure of a good specification may not correspond to the structure one may use to provide an efficient implementation. In specification one is trying to provide a clear logical separation of concerns, while in implementation one may take advantage of the relationships between logically separate parts to provide an efficient implementation of the combined entity. The intellectual ability required of a good specifier is roughly equivalent to that of a good programmer; however, the view taken of the system must be different.

13.6.3 Technical problems

The following technical specification problems were discovered in applying formal specification techniques to CICS:

- putting the module specifications together to provide a specification of the system as a whole;
- specifying parallelism;
- specifying recovery on system failures; and
- specifying distributed systems.

We shall briefly discuss each of these in turn.

Putting modules together Currently, three modules out of the sixteen modules that form the application programmer's interface have been specified and we now feel we have enough insight into the system to consider the problem of putting the module specifications together. Each module has state components and a set of operations that work on those state components. Putting the modules together amounts to combining the states together to form the state of the system, and extending the operations of the modules to operations on the whole system. The problems encountered in putting modules together were as follows:

- avoiding name clashes when the modules were combined;
- specifying the effect on the whole system state of an operation defined within a module of the system; and
- coping with situations in which an operation of one module refers to state components of another module.

Parallelism In our current specifications the operations are assumed to be atomic operations acting on the state of the system. We have a sufficient underlying theory to allow one to reason formally about a single sequential transaction. An area for future research is to extend the theory to allow reasoning about the interactions between parallel processes. The current specifications will still be used but they will need to be augmented with additional specifications which constrain the way in which the parallel processes interact.

Recovery An important part of a transaction processing system is the mechanism for recovery on failure of the system. The current specifications do not address the problem of recovery. Again we would like to augment the current specifications so that recovery can be incorporated without requiring the existing part of the specification to be rewritten.

Distributed systems A number of CICS systems may cooperate to provide services to users. The main facility provided within CICS to achieve this is the ability to execute certain operations or whole transactions on a remote system. While the individual operation specifications could be augmented to reflect remote system execution, it was thought better to wait until we had a specification of the system and extend that to a distributed system. To reason effectively about a distributed system we need to be able to reason about parallelism.

13.7 Conclusions

Formal specification techniques have been successfully applied to modules of an existing system and as an immediate benefit have uncovered a number of problems in the current documentation as well as flaws in the current interface design. In the longer term the formal specifications should provide a good starting point for specifying proposed changes to the system, a more precise description for educating new personnel, and a basis for improved documentation.

In part the reason we have been successful in applying our specification techniques is that the modular structure of CICS is quite good, and we have been able to take advantage of this by concentrating on individual modules in relative isolation.

The main short-term benefits that are obtained by applying formal specification techniques to existing software are the questions that are raised during the specification process. They highlight aspects of the system that are incompletely or ambiguously described in the manual, as well as focusing attention on problems with its structure, for example, undesirable interactions between modules.

In the longer term a formal specification provides a precise description which can be used to communicate between people involved with the system. The specification is less prone to misunderstanding than less formal means of communication, such as natural language or diagrams. It can be used as a basis for a new specification which incorporates modifications to the original design, and it provides an excellent starting point for people responsible for improving the documentation. (In another group at Oxford work on incorporating formal specifications into user manuals is being done by Roger Gimson and Carroll Morgan [43].)

The time required to specify a module of the system varied from about 4 weeks for Exceptional Conditions to 12 weeks for Interval Control. The time required was related to the size of the module (the number of operations, etc.) and also to the number and severity of problems raised about the behaviour of the module. The size of a module specification (in pages) turned out to be roughly comparable to the size of the manual entry for the module. The specification sizes ranged from 4 pages (handwritten) for Exceptional Conditions to 16 pages for Interval Control.

The difficulties encountered with the specification process itself were the language gap between university and industry, and the problem of achieving the right level of abstraction. There were also a number of more technical specification problems that arose when applying the techniques: the problem of putting together module specifications to provide a specification of the system as a whole, specifying parallelism, specifying recovery on system failure, and specifying distributed systems. These problems are areas for further research.

Acknowledgements I would like to thank IBM for their permission to publish this chapter and reproduce part of one of their manuals as an appendix. Several members of the IBM Development Laboratory at Hursley, England assisted the author to understand some parts of CICS; of special note are Peter Alderson, Peter Collins and Peter Lupton.

This work has benefited from consultations with Tony Hoare, Cliff Jones and Rod Burstall. Tim Clement was responsible for the initial specification of temporary storage and exceptional conditions. Paul Fertig, Roger Gimson, John Nicholls and Bernard Sufrin gave useful comments on this chapter. Finally, I would like to express my gratitude to Carroll Morgan and Ib Holm Sørensen for their help as reviewers of the specifications, and for their instruction in specification techniques.

13.8 Appendix: exceptional conditions manual

The following is an extract of the manual entry for exceptional conditions taken from [29].

Exceptional conditions may occur during the execution of a CICS/VS command and, unless specified otherwise in the application program by an *IGNORE CONDITION* or *HANDLE CONDITION* command or by the *NOHANDLE* option, a default action for each condition will be taken by it. Usually, this default action is to terminate the task abnormally.

However, to prevent abnormal termination, an exceptional condition can be dealt with in the application program by a *HANDLE CONDITION* command. The command must include the name of the condition and, optionally, a label to which control is to be passed if the condition occurs. The *HANDLE CONDITION* command must be executed before the command which may give rise to the associated condition.

The *HANDLE CONDITION* command for a given condition applies only to the program in which it is specified, remaining active until the associated task is terminated, or until another *HANDLE CONDITION* command for the same condition is encountered, in which case the new command overrides the previous one.

When control returns to a program from a program at a lower level, the *HANDLE CONDITION* commands that were active in the higher-level program before control was transferred from it are reactivated, and those in the lower-level program are deactivated.

Some exceptional conditions can occur during the execution of any one of a number of unrelated commands. For example, *IOERR* can occur during file-control operations, interval-control operations, and others. If the same action is required for all occurrences, a single *HANDLE CONDITION IOERR* command will suffice.

If different actions are required, *HANDLE CONDITION* commands specifying different labels, at appropriate points in the program will suffice. The same label can be specified for all commands, and fields *EIBFN* and *EIBRCODE* (in the *EIB*) can be tested to find out which exceptional condition has occurred, and in which command.

The *IGNORE CONDITION* command specifies that no action is to be taken if an exceptional condition occurs. Execution of a command could result in several conditions being raised. CICS/VS checks these in a predetermined order and only the first one that is not ignored (by an *IGNORE CONDITION* command) will be passed to the application program.

The *NOHANDLE* option may be used with any command to specify that no action is to be taken for any condition resulting from the execution of that command. In this way the scope of the *IGNORE CONDITION* command covers specified conditions for all commands (until a *HANDLE CONDITION* for the condition is executed) and the scope of the *NOHANDLE* option covers all conditions for specified commands.

The **ERROR** exceptional condition

Apart from the exceptional conditions associated with individual commands, there is a general exceptional condition named *ERROR* whose default action also is to terminate the task abnormally. If no *HANDLE CONDITION* command is active for a condition, but one is active for *ERROR*, control will be passed to the label specified for *ERROR*. A *HANDLE CONDITION* command (with or without a label) for a condition overrides the *HANDLE CONDITION ERROR* command for that condition.

Commands should not be included in an error routine that may give rise to the same condition that caused the branch to the routine; special care should be taken not to cause a loop on the *ERROR* condition. A loop can be avoided by including a *HANDLE CONDITION ERROR* command as the first command in the error routine. The original error action should be reinstated at the end of the error routine by including a second *HANDLE CONDITION ERROR* command.

Handle exceptional conditions

HANDLE CONDITION

<pre> HANDLE CONDITION condition [(label)] [condition [(label)]] ... </pre>
--

This command is used to specify the label to which control is to be passed if an exceptional condition occurs. It remains in effect until a subsequent *IGNORE CONDITION* command for the condition encountered. No more than 12 conditions are allowed in the same command; additional conditions must be specified in further *HANDLE CONDITION* commands. The *ERROR* condition can also be used to specify that other conditions are to cause control to be passed to the same label. If ‘*label*’ is omitted, the default action for the condition will be taken.

The following example shows the handling of exceptional conditions, such as *DUPREC*, *LENGERR*, and so on, that can occur when a *WRITE* command is used to add a record to a data set. *DUPREC* is to be handled as a special case; system default action (that is, to terminate the task abnormally) is to be taken for *LENGERR*; and all other conditions are to be handled by the generalized error routine *ERRHANDL*.

```

EXEC CICS HANDLE CONDITION
      ERROR(ERRHANDL)
      DUPREC(DUPRIN)
      LENGERR

```

If the generalized error routine can handle all exceptions except *IOERR*, for which the default action (that is, to terminate the task abnormally) is required, *IOERR* (without a label) would be added to the above command.

In an assembler-language application program, a branch to a label caused by an exceptional condition will restore the registers in the application program to their values at the point where the EXEC interface program is invoked.

In a PL/I application program, a branch to a label in an inactive procedure or in an inactive begin block, caused by an exceptional condition, will produce unpredictable results.

Handle condition command option

condition [(label)] ‘*condition*’ specifies the name of the exceptional condition, and ‘*label*’ specifies the location within the program to be branched to if the condition occurs. If this option is not specified, the default action for the condition is taken, unless the default action is to terminate the task abnormally, in which case the *ERROR* condition occurs. If the option is specified without a label, any *HANDLE CONDITION* command for the condition is deactivated, and the default action taken if the condition occurs.

Ignore exceptional conditions

IGNORE CONDITION

IGNORE CONDITION	condition
	[condition]
	...

This command is used to specify that no action is to be taken if an exceptional condition occurs. It remains in effect until a subsequent *HANDLE CONDITION* command for the condition is encountered. No more than 12 conditions are allowed in the same command; additional conditions must be specified in further *IGNORE CONDITION* commands. The option '*condition*' specifies the name of the exceptional condition that is to be ignored.

Chapter 14

The use of Z in the restructure of IBM CICS

Steve King

Abstract In April 1992, Oxford University Computing Laboratory and IBM (UK) Laboratories Ltd were joint winners of a Queen's Award for Technological Achievement, for the use of the Z notation in the development of a transaction processing system. This chapter describes the work which led to this award: how Z was introduced into the development process at IBM Hursley, the changes to the process which were then possible, and the results obtained, in terms of numbers of errors reported.

14.1 Introduction

IBM (UK) Laboratories Ltd and Oxford University have been engaged in a joint research project since 1981. The aim of the project is to investigate the applicability of formal specification techniques to industrial scale software. Earlier results from this project have been reported in [8, 17, 67]; this chapter continues the story. We relate how the descriptive work of [17] was followed by the use of Z in the actual development of code, and the improvements in quality which resulted.

The next chapter (Chapter 15) describes how a more recent project has built on the work reported in [17] to give a formal description of a large part of the CICS Application Programming Interface (API).

14.2 The CICS program product

CICS (Customer Information Control System) is the main IBM System/370 online transaction processing system. It is widely used throughout the world, with applications including banking, airline reservations and insurance. CICS can support installations with large databases being accessed by many terminals. Descriptions of CICS can be found in [27] and [68]. CICS was originally developed in 1968, and it now

Copyright © 1992 International Business Machines Corporation.

consists of over 800,000 lines of code, some written in System/370 Assembler language but most written in an IBM internal systems programming language (PLAS). Since 1968, CICS has been continuously developed and extended. The product lifecycle is such that a new release is produced approximately every two years, usually maintaining the behaviour of previous releases but providing more functionality. Since the period between planning and release of a new version is longer than two years, there are several releases being worked on at Hursley at any one time.

Because of its long history, involving adding functionality to an already existing system, it is not surprising that, by 1982, the internal structure of the CICS program itself had become somewhat complicated. It was therefore decided to restructure and rewrite part of the CICS program, using Z to provide formal specifications of the new code that was to be written. In the event, Z was used on only some of the new code (approximately 14 per cent), but this still amounted to several tens of thousands of lines of code. This work is described in this chapter.

It should be noted that the success of the project described here is due to many people: a list of the main participants may be found at the end of the chapter.

14.3 Early experiments

As we mentioned above, by 1982 there was general agreement in the CICS Technical Office on the need for internal restructure and rewriting of the CICS product. This was scheduled to be a part of CICS/ESA Version 3 Release 1. There was also a growing interest in the use of formal specification techniques. Professor C. A. R. Hoare of the Oxford University Programming Research Group (PRG) had suggested that the time was right to see whether these techniques could be successfully used on industrial-scale problems [20]. Thus the scene was set for a joint research contract between IBM Hursley and the PRG to investigate these mathematical methods of developing software. Until then, the PRG had only been able to apply them to small and medium-sized examples, so they were interested to see whether the methods would scale up to industrial problems.

The first stage in the joint research project involved the PRG researchers working on case studies. Specifications of several small program modules and parts of the API were produced (some appear in this volume – see Chapter 13).

The primary objective of IBM's preliminary work on formal specification was to decide on the notation to be used. Two possibilities were considered: Common Design Language (CDL) and Z. CDL was an internal IBM design language which has many features derived from high-level programming languages. The first case studies were written in both CDL and Z, so that comparisons could be made. It was eventually decided to concentrate on Z, rather than CDL, because it seemed to be possible to write Z specifications which were more elegant than their CDL counterparts, and more easily captured the precise requirements. It was also felt that the mixture of formal statements and explanatory English text would make the Z documents more 'acceptable' to the many non-experts in formal methods who would have to come into contact with them.

While the Oxford researchers were working on the case studies, there were also a few Hursley designers who were experimenting with Z in their day to day work. While there was as yet no official policy on Z, there was a real interest in the use of formal methods. The case study work was important for several reasons:

- It enabled the researchers to learn about the IBM culture. They spent a considerable amount of time at Hursley, attending seminars and discussion sessions on the internal structure of CICS. This greatly improved communications at later stages in the project, when the researchers were advising designers who perhaps had some problem in expressing a requirement in the formal notation, or could not see how to model a particular structure: the researchers were then able to concentrate on the problem, rather than needing lengthy explanations of its background.
- It also meant that there was a resource available to the IBM developers, which contained examples from a problem domain that was familiar to them. In some cases, these case studies were used as templates on which to base the ‘real’ specifications which were written later. They were also useful as education material.

It should be noted that the emphasis at this time in the use of Z was on specification, rather than development, on recording requirements at a suitable level of abstraction, rather than on how to produce designs and code to implement those requirements. Although some work had been done on Z and refinement, it was still at a theoretical level, rather than being practically useful. However, the CICS staff decided that it would still be worth introducing formal methods into the development process, even if it was only as a notation for recording specifications and designs. It was recognised that further work was needed on the practicalities of applying formal or rigorous development techniques in an industrial environment.

14.4 The decision to use Z

The start of the second stage of the project was marked by the decision, in 1984, by the CICS development managers to accept a recommendation from the CICS Technical Office that Z should be used on (at least some of) the next release. This decision brought about some changes to the development process which we will now describe.

Prior to 1984, there was already a well-established development process in place in CICS development – the IBM programming process architecture, which is described in [51]. The development process is divided into 13 stages (see Figure 14.1), and there are well-defined criteria for progressing from one stage to the next, involving formal reviews and inspections. Before the introduction of Z, specifications (at the PLD stage) were written in English prose, and designs (at CLD and MLD) were written in a combination of pseudocode and prose. It was decided to use Z for recording specifications and high-level designs, and to use some other notation for lower-level designs (see below). The most obvious change in the process which came about because of the use of Z was the introduction of another inspection. Prior to the use of Z, there were inspections at the CLD and MLD stages (known as I0 and I1). After the introduction of Z into the process from the PLD stage on, it was felt that it would be valuable to have an inspection at the end of the PLD stage: this became known as DR0. Since there was now a formal document at this stage which it was possible to inspect, it seemed a good idea to do so, in order to catch any possible errors as early as possible.

There still remained a decision to be taken about what notation should be used for lower-level designs. This notation was important because it had to bridge the gap between the state-based descriptions of the high-level design and the code which

Family	Stage	Acronym	Inspection
Requirements	Systems requirements and design	REQ	
	Product requirements and design		
Design	Product level design	PLD	DR0
	Component level design	CLD	I0
	Module level design	MLD	I1
Implementation	Code	CODE	
	Unit test	UT	
Testing	Functional verification test	FVT	
	Product verification test	PVT	
	System verification test	SVT	
Package and validate	Package and release	ISD	
	Early support program	ESP	
General availability		GA	

Figure 14.1: The thirteen stages of the IBM programming process architecture, and the families into which they can be grouped

was to implement those designs. After several experiments, it was decided to use Dijkstra's language of guarded commands [10, 11], because of its simplicity and sound mathematical foundations, which meant that, at some future date, it might be possible to bring the ideas of formal refinement and proof into the development process. It was also significant that there was a textbook available, which introduced the notation and gave many examples of its use [14]. By a happy coincidence, the book's author, David Gries, was on sabbatical leave at Oxford at the time, and he was invited to visit Hursley to give a seminar on program development, using material from his book. This was valuable in showing the practical application of the method.

In summary, Z was used for specification, at the PLD stage, rather than the English that had been used previously; and a combination of Z and the guarded command language was used for designs (CLD and MLD), rather than the English and pseudocode which was previously used.

Because of the changes to the development process that were being introduced, two modules were selected for accelerated development. The idea was that two senior and experienced developers would carry out the work in order to gain practical experience of the Z approach. All aspects of the work were carefully documented to provide guidance for the developers of other modules.

14.5 Education and tools

Once the decision had been taken to use Z on part of the new release of CICS, there was obviously a need to provide Z education for designers and developers. There was a great variety in the experience of members of the development teams – some had been writing software for 17 years, others for six months. There was also a great mixture of backgrounds and abilities, although none had any experience in the use of formal methods. Many had attended the IBM Software Engineering Workshop, which introduced software engineering techniques, such as abstraction and loop invariants, and used the CDL notation. An additional three-day course on Z was offered, taught initially by Oxford researchers. This introduced the notation, and taught designers and developers how to write Z. It was also found helpful to have a 'Z for readers'

course: this was a course to help those who had to read documents containing the Z notation, rather than needing to write Z, such as testers and writers of IBM publications. A few courses on ‘Refinement from Z specifications’ were also taught, although the techniques were new, and were not put into practical use. By 1985, approximately 130 students had attended the courses on writing Z specifications, 30 had attended the courses on reading Z specifications, and 50 had attended the courses on refinement.

When IBM started using Z, there were few tools for Z available anywhere. The tools developed for use at Hursley were simple, and were actually written by the users of Z. The first need was for mechanisms for displaying and printing the many Z mathematical symbols: this involved designing a new font for the IBM 3800 printer, and defining SCRIPT variables and GML tags. A cross-reference tool was developed, and later this was revised and improved by a student working at Hursley for a year during his university course. This tool was an invaluable aid in helping people to navigate around large specifications. It should be noted that this work on tools was carried out well before there was an agreed syntax for the language. In fact, pressure from Hursley, as well as elsewhere, was a major factor in persuading the PRG to start its effort on the standardisation of the language. Although there were so few tools available to the users of Z at Hursley, there did not seem to be any great pressure to provide more tools, until much later, when more sophisticated tools were provided as part of a more organised tools strategy. This change came about when tools became available in the wider Z community, and IBM Z users realised how they could benefit from their use.

14.6 Results

It is an integral part of the software development process at Hursley to collect measurements at various points in the process. However, since there is a significant time lag between specification and shipping to customers, the first feedback that was obtained on the use of Z was more subjective than objective. Later on, it has become possible to look at a comparison of the numbers of errors found by customers in code which has been specified in Z and code which has not: since CICS/ESA Version 3 Release 1.1 was only made generally available to customers in June 1990, this comparison has only recently been possible.

14.6.1 Subjective results

The early feedback from designers and developers working in Z was generally favourable: they were impressed with the simplicity and elegance of the notation, and few were unable to cope with the mathematics required. One of the perceived advantages lay in the use of the schema calculus to construct specifications from smaller parts – in particular, it was felt to be helpful to describe the successful version of an operation separately from the error cases, and then to combine them using schema disjunction. The subjective impression of the users of Z was that the quality of their work was improved. They were more confident that the code they were producing was ‘correct’. Overall there did not seem to be a significant effect on project

scheduling – more time was spent on the earlier stages of the process, but this was offset by reduced time on coding. However there was a one-time startup cost of about three months due to the need for the developers to be educated in Z and to learn the art of writing abstract specifications. It may be that the improvement in quality was simply caused by the discipline of writing the formal document – it was much harder to gloss over important issues. It was also found that the use of Z encouraged the writing of more precise English prose. Most of the Z language was used, but not all: it seemed that bags were not often used, and the facilities for generic definitions were seldom required.

Several other factors were important in the introduction of Z:

- The users of Z actually believed in the techniques – they were not just being imposed by management, but the users appreciated how the new techniques would improve their work, and hence the product. An important factor in bringing about this belief was education, which included the use of examples in the users' own application areas.
- Skilled consultancy was available to the Z users. A three-day course on Z can teach students the notation, but real fluency comes only with practice, and novices are bound to come across problems in their first few attempts at non-trivial examples. It is important that experts are readily available for consultation at this stage, to prevent disillusion setting in! At first, consultancy was available to Hursley Z users from the Oxford researchers, but, later, local experts were able to take over.
- There was management backing for the work. The success of the Z project was actually important to management, since they had made a decision to go outside the company's accepted methods of software development: the success of the project would vindicate their decision.

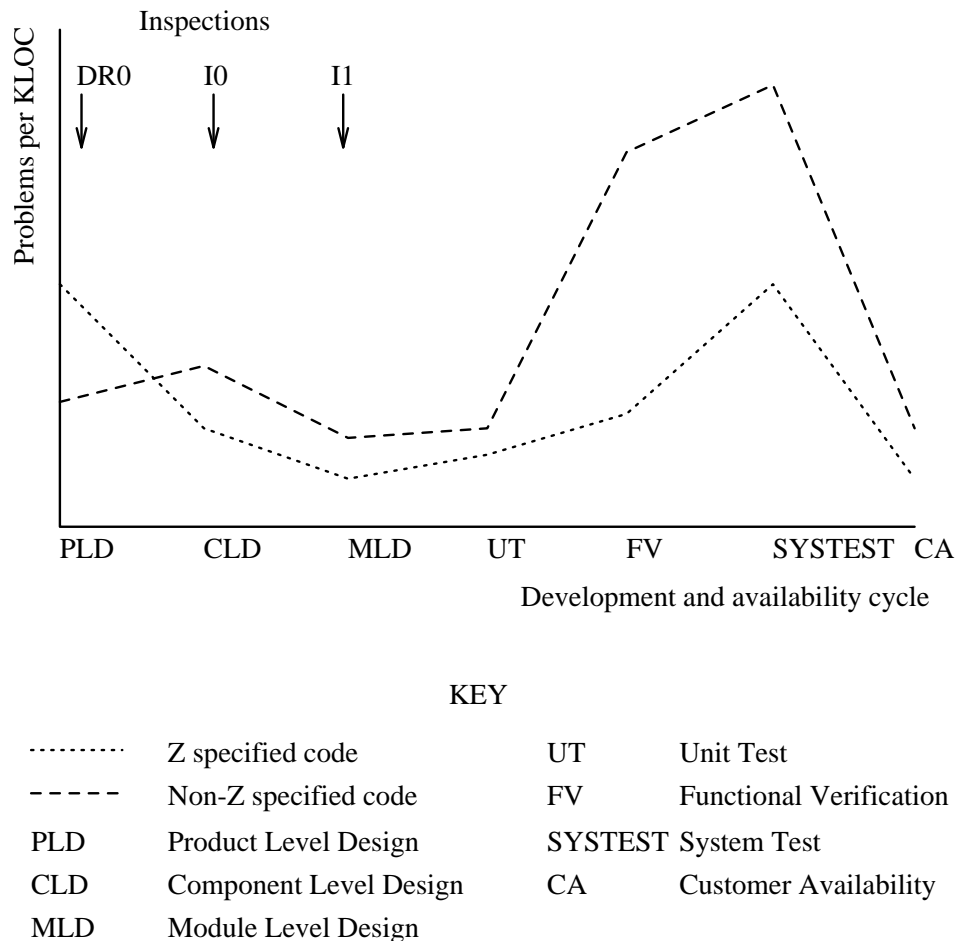
14.6.2 Quantitative results

As we mentioned above, a large number of measurements were taken during the development process for CICS/ESA Version 3 Release 1. Some of these figures have been released by IBM, and they make encouraging reading for supporters of formal methods. However, a word of caution is in order here: it should be noted that IBM do not claim to have been running a carefully designed scientific experiment to test the merits of formal methods – there are no control cases, where we can look at the results of developing the same piece of code with formal methods, and without. All we can see are the results of applying formal methods in certain specific cases, and we can give only tentative interpretations of the figures. It would not be difficult for an opponent of formal methods to give different, less favourable, interpretations of the data: for instance, it might have been the case that Z was used only on the least complex parts of the work, or that it was actually the best developers that were using Z.

In order to interpret the figures on error rates, it is important to understand the scale of the software being produced. CICS/ESA Version 3 Release 1 is the largest release of CICS that IBM has so far produced. The release consisted of 268,000 lines of new and modified code (together with over 500,000 lines of unchanged code). Of the new and modified code, approximately 37,000 lines (14 per cent) were produced from Z specifications and designs, and a further 11,000 lines (4 per cent) were partially

specified in Z. Some 2000 pages of formal documentation were produced – this includes both specifications and designs.

IBM has released the graph in Figure 14.2, which enables us to examine the number of problems found at various stages of the development cycle, with figures for Z specified and non-Z specified code. The first impression is that the problem



The vertical scale shows problems per thousand lines of code (KLOC), on a linear scale, starting from zero.

Figure 14.2: Comparison of problems found with two development methods in CICS/ESA
Version 3 Release 1

rate is lower in the Z specified code. However, when we look more closely, we can see that the error rate for Z specified code is actually higher at the PLD stage. (These are the errors found at the DR0 inspection.) This can be explained because of the precision that is forced on the writer of the formal specification: he or she cannot ignore difficult issues, and so is more prone to making a wrong decision, which can be picked up by an inspector. Before the introduction of Z, the document produced

at this stage was written in English. This allowed the possibility that the writer and a reader might have very different views of the function to be provided: it was only later in the development, when the design was becoming more concrete, that the disagreement about the meaning of the specification would come to light. It is important that these errors are trapped at an early stage in the process, since a later recognition of the same problem would be far more expensive.¹ In fact, the graph shows a worrying number of problems caught only at the system test stage for the non-Z specified code: correction of these errors is not likely to be cheap.

Based on the above results about the reduction in frequency of errors in the Z specified code, IBM has also calculated that there is a reduction in the total development cost of the release. Since there is a corresponding reduction in programmer days spent fixing problems, they estimate a 9 per cent reduction, as compared to developing the 37,000 lines without Z specifications.

One of the better measurements of quality for a software product like CICS is the number of errors reported by customers, since this is an obvious way for the customer to judge quality. However, the length of time and the size of the sample mean that figures available so far should be treated with some caution. CICS/ESA Version 3 Release 1.1 was only made generally available to customers at the end of June 1990, and it is IBM's experience that many customers do not change immediately to a new release when it is made available, unless there is some new functionality that they particularly need, or they are working at the limits of the capacity of the previous release. However, bearing those two provisos in mind, the figures on number of problems reported by customers are extremely encouraging: in the first 18 months after the release was made available, the code that was specified in Z seems to have approximately $2\frac{1}{2}$ times fewer problems than the code not specified in Z. These figures are even more encouraging when it is realised that the overall number of problems reported is lower than on previous releases. There is also evidence to show that the severity of the problems for code specified in Z is much lower than for the other problems.

Taken as a whole, the quantitative results for the use of Z in CICS are encouraging – they are certainly encouraging enough for IBM to have decided to continue the use of Z on the next release. It is to be hoped that the results will remain as good when the number of users of the new release increases. It will also be interesting to see how the Z specified code compares to the old code for ease of maintenance.

14.7 The Oxford–Hursley collaboration

As we mentioned before, the research contract between IBM and the PRG started in 1981. It is interesting to look at the work items on this contract and how they have changed over the years, because this reflects the areas in which the Hursley users of Z have been having problems at particular times. As we have mentioned before, the earliest work on the contract involved showing how Z could be used for the specification of (sequential) systems. The notation itself was still being developed, and Hursley was a good source of case studies: having worked on small and medium-sized examples, the PRG researchers were able to test out their notations on larger-scale examples, such as parts of the CICS API. An important part of the PRG researchers'

¹Studies have shown ([4, p. 44]) that, for large projects, it is 100 times more expensive to fix an error found during maintenance than it is to fix the same error when it is found during requirements capture.

work at this time was in education and consultancy: they conducted courses and seminars on Z at Hursley, both for technical staff and for managers. They also spent a lot of time at Hursley, simply being available to help novice users of the notations with any problems that might arise.

Around 1985, the nature of the PRG researchers' work began to change: they spent less time at Hursley in direct contact with Z users, and more time working at Oxford. By then, the Z notation had become fairly stable, a formal semantics had been published [54], and there was a growing pressure on the PRG from the user community to start a standardisation effort. Part of the reason for this pressure was the way in which the notation had developed: it had been driven by case studies, and there were now parts of the language which were not covered at all by the earlier descriptions. There was also pressure from tool builders who wanted an agreed standard on which to base their tools. Despite initial resistance from some parts of the Z community in Oxford, work progressed, and at least two syntaxes were published [36, 57]. In 1990, a project was started, which was funded by government and industry, with IBM as one of the partners, which had the goal of standardising the notation. A draft of the complete standard has now been produced [7], and this will be submitted for BSI and ISO consideration in the near future.

One of the research items of perennial interest to IBM has been the specification of concurrency. Since a transaction processing system involves many users contending for resources, it is convenient to be able to exploit parallelism. However, there are difficulties in combining the model-based approach of Z with the specification of concurrency. Various approaches, including CSP and action systems, have been investigated, and work is continuing.

The final two research areas are closely linked: refinement and proof. It seems likely that formal refinement techniques will not be widely used at Hursley until there is at least some sort of mechanical assistance for proof, due to the problems of scale involved. So work on refinement and proof is continuing at present. However, significant advances have been made in the theory of refinement, as a direct result of problems encountered by IBM developers at Hursley. In the early 1980s, the proof obligations for data refinement were formulated in terms of a retrieve function: a function that mapped concrete to abstract states. However, this function was not sufficiently general to allow certain refinements which IBM developers believed to be valid. This led to the more general concept of a retrieve relation, and to a re-examination of the theory of refinement by other members of the PRG, eventually producing the work on upwards and downwards simulations by Hoare and others [18, 32].

In all these research areas, the collaboration between Hursley and Oxford has been fruitful. While IBM has obtained solutions to some of the theoretical problems which have arisen in their use of formal methods, the PRG has had an industrial-scale testing ground for its formal specification techniques, as well as a plentiful source of research problems. In recent years, there has also been some exchange of personnel, with two IBM members of staff each spending a year's study leave at Oxford, and a PRG researcher spending a year working on the specification of the CICS API (described in Chapter 15).

14.8 Conclusions

This chapter has described the use of Z in a development project at IBM Hursley. The project involved introducing Z into the software development process for a release of the CICS transaction processing system. Specifications and high-level designs were written in Z, while lower-level designs were written in Dijkstra's language of guarded commands. As part of the IBM development process, measurements were taken at various times. These make encouraging reading for supporters of formal methods, since they show a greatly reduced error rate in the code produced from Z specifications.

Acknowledgements The work described in this chapter has been carried out by many people, and there are many who have contributed significantly to its success. I can only attempt to acknowledge their work by giving a list of names. Any omissions are accidental, and are entirely my fault! Many of these people have made comments on this chapter, and I am grateful to them – in particular, Peter Collins, Ian Hayes, Tony Hoare and John Wordsworth. I would also like to thank IBM (UK) Laboratories Ltd for permission to use their figures on the use of Z in CICS/ESA Version 3 Release 1. Two earlier reports [8, 67] have described some aspects of these two projects; I am happy to acknowledge my debt to their authors.

At IBM Hursley, those involved with Z have included, as writers: Peter Alderson, David Blyth, Jonathan Hoare, Iain Houston, Steve King, Peter Lupton, Paul Mundy, Glyn Normington, Mark Phillips, David Renshaw, Colin Shade, John Wordsworth; and, as managers: Peter Bauchop, Peter Collins, Julian Jones, John Nicholls, Mark Phillips, Tony Rogers.

At Oxford, those involved in the IBM CICS project have included, as research officers: Tim Clement, Ian Hayes, Mark Josephs, Steve King, Divya Prasad, Jane Sinclair, Ib Sørensen, Jim Woodcock; as coordinators: John Nicholls, Ib Sørensen; and, as consultants: Tony Hoare, Cliff Jones, Mark Josephs, Jim Woodcock.

Chapter 15

Specifying the IBM CICS Application Programming Interface

Steve King

Abstract This chapter reports on a recent project at IBM Hursley, which involved the description of a large part of the Application Programming Interface for the CICS transaction processing system. The purpose of this exercise was not to produce code, nor was it simply an academic exercise in specification. Instead, the aim was to provide, for commercial reasons, a formal description of the already existing interface.

15.1 Introduction

At the outset of the joint research contract between IBM Hursley and Oxford University's Programming Research Group, one of the main items of interest was the specification of particular modules from the Application Programming Interface (API) for CICS, IBM's major transaction processing system. These case studies, some of which are included in Chapter 13, were written by the Oxford researchers partly as a test of the Z notation, to show IBM that it could be successfully used on problems of this size and complexity. Later, after it had been decided to use Z in the restructuring of the CICS code (see Chapter 14), these case studies were useful as examples, in a familiar problem domain, for the IBM developers who were writing their own specifications.

Some years later, after the CICS restructure had been completed, it was decided to revisit the earlier work, and to write Z specifications for most of the application programming interface. This work is described in this chapter.

Copyright © 1992 International Business Machines Corporation.

15.2 Using Z to describe interfaces

As well as the work on the particular modules of the API described above, which was carried out by the Oxford researchers, there had also been two previous pieces of work within IBM Hursley which involved defining programming interfaces using Z. In 1988, a decision was made to extend the CICS API by adding the Common Programming Interface for Communications, a part of IBM's Systems Application Architecture. In order to help the developers who had to implement the interface within CICS, a Z specification was written, which was intended to make more precise the informal description which had been given by the architects (who were based in the United States). As might be expected, the process of writing this specification revealed various inaccuracies and omissions in the original informal description. The specification itself has been published [9], as well as reports of the work [66].

Work on specifying programming interfaces continued in 1989, with a project to specify a small part of the CICS API. In the release which was then being worked on, CICS file control was enhanced by adding a new feature, known as data tables. Once again, in order to help the developers, a Z specification of this feature was written, which has also been published [24].

With the knowledge that these two projects had demonstrated the practicality of describing programming interfaces in Z, it was decided in November 1989 to describe a large part of the CICS API. This work has now almost finished, and we describe it here.

15.3 The CICS Application Programming Interface (API)

For early versions of CICS, writing CICS application programs involved embedding macro calls in the program. The macro interface exposed information on CICS internal control blocks and encouraged a style of programming which exploited this knowledge of the internal structure of CICS. In 1976, with the release of CICS/OS/VS Version 1 Release 3, a command-level application programming interface was provided as an alternative to the macro-level interface. The aim was to move towards a cleaner interface, making application programming easier and less error-prone, and removing the need for the programmer to understand the CICS control blocks. (Support for the macro-level interface was withdrawn for COBOL and PL/I applications in 1990, with the release of CICS/ESA Version 3 Release 1.1, and for assembler applications in 1991, with the release of CICS/ESA Version 3 Release 2.1.)

The purpose of a transaction processing system such as CICS is to control the way in which many tasks are allowed access to resources such as data files, storage and terminals. The Application Programming Interface (API) is the means by which CICS application programs can request access to these resources. Figure 15.1 gives a diagrammatic representation of the interface.

Application programs are written in a familiar imperative language, and CICS commands are inserted where needed. For instance, a CICS COBOL program might contain:

```
MOVE DATA1 TO REC1.
EXEC CICS WRITE FILE(FNAME)
      FROM(REC1)
```

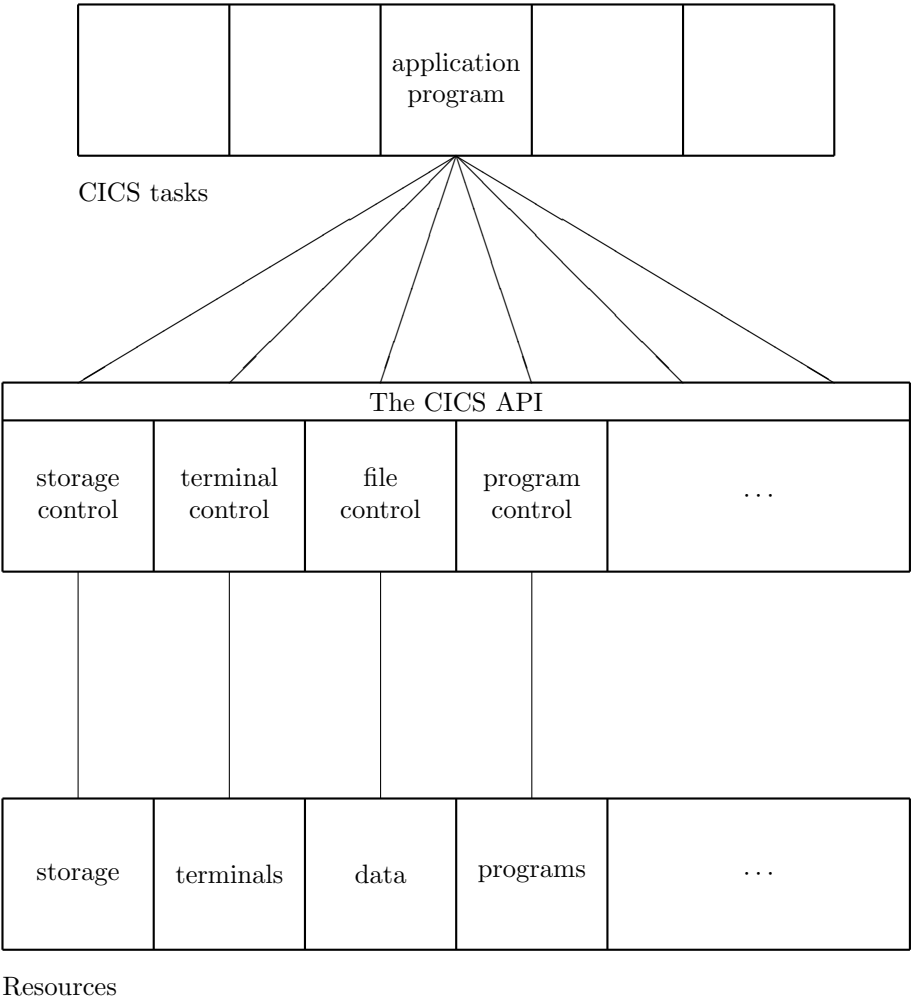


Figure 15.1: The CICS API as a layer between programs and resources

```

LENGTH(255)
RESP(RESPCODE) END-EXEC.

```

A preprocessor for each supported programming language is supplied as part of CICS. This is used to translate the embedded commands into calls, in the appropriate language, to the CICS command processor.

The API is described informally in an IBM manual [25], which contains 91 commands. Each command has several options which may be supplied by the programmer – there are 312 options in all – and each command gives a response – either the *normal* response, or one of the 60 error responses. The API has traditionally been divided into several groups of commands – some of them are shown in Figure 15.1. These groupings had been used in early versions of the informal description [25], and so it was decided to use them as the basis of the formal specifications; rather than producing one monolithic document describing the whole of the API, several documents were produced, each of which was smaller and easier to comprehend, and only described a section of the API. In fact, there is not an exact one-to-one correspondence between the traditional groupings and the specifications that were produced; in several cases, a document was produced which covered the API commands from more than one area, and one additional specification was produced, which did not describe any API commands, but instead described some aspects of Transaction Management – how tasks are created and destroyed, and how they are associated with terminals. It was also decided to divide up the descriptions of two special API commands and distribute them among the other specifications (see below).

15.4 Reasons for specifying the API

At present, there are versions of CICS available for five different operating systems: CICS/ESA, CICS/OS/VS, CICS/MVS, CICS/OS/2, CICS/VSE.¹ Although there are some pieces of functionality that are only supported for one or two operating systems, it is intended that there should be a command *base*, consisting of about 60 commands, which will contain commands whose behaviour is consistent across all operating systems. The aim of the API specification project was to describe formally this command base. This definition would support the idea of a family of CICS products, which would provide a common interface, enabling portable applications to be written.

A second reason for writing the specifications was to give a definitive statement of the behaviour which CICS *guarantees* of its API, as opposed to what might be termed ‘accidental’ behaviour. This, in turn, would allow much better communication with the user of the API. Without the formal specification, the application programmer has to rely on the informal descriptions in the CICS manuals in order to understand the effect of a particular command. It comes as no great surprise to proponents of formal methods that these informal sources are incomplete, and often ambiguous. Thus the programmer is often reduced to experimenting with the system to find out what a command does in a particular set of circumstances. These experiments are often time-consuming, and may also be inconclusive. However a more serious problem is that they may uncover *accidental* behaviour which was not part of the original designer’s intention. Since the behaviour was accidental, it is possible that it might disappear in a future release, which could have disastrous consequences if programmers have based

¹All trademarks of the IBM Corporation.

their programs on that behaviour. With over 30,000 licences for CICS worldwide, it is likely that virtually every single strange behaviour of a CICS command has been discovered, and is being relied upon, by some programmer somewhere. Thus the developers of CICS are themselves constrained by having to ensure backward compatibility. The provision of a formal description of the API is an important step in improving the communication between the suppliers of CICS and their customers. It tells the customers what behaviours of the API they can safely exploit, and it helps them to avoid exploiting behaviour which is not guaranteed. Similarly it can protect the supplier from inadvertently destroying part of the guaranteed function.

A subsidiary aim of the API specification project was to encourage the use of formal methods, in the rest of CICS design and development, in other parts of IBM, and among the many suppliers of CICS-based application packages. Since the use of Z on the restructure project (see Chapter 14) had involved work of a proprietary nature, none of that Z work could actually be published. However, since the API is a public interface, the new work could be made widely available. It was hoped that this large, published, example of formal specification, produced in a commercial environment, would encourage the teaching of formal methods in higher education centres, and would also encourage other companies to use such methods.

15.5 How the specifications were written

Each specification was the responsibility of a single author. This author carried out the necessary research and wrote the Z specification. In almost all cases, the authors were ‘Z experts’ rather than ‘CICS experts’. The first stage of the specification process involved large amounts of research to discover what the particular CICS commands actually did. The primary source of information for this research was the manuals which are supplied to customers [25, 26], together with any other printed material that might be available – for instance, the ‘prototype’ API specifications mentioned above, which were written in the early 1980s. It was also useful at this stage to establish a good relationship with the Hursley experts in the relevant area – these were the people to whom the specifier could go with a list of questions and expect definitive answers about exactly what happened when a particular command was executed under certain circumstances. Interestingly, the answers often seemed to be based on the experts’ intimate knowledge of the code, rather than because they already had a good abstract model of the behaviour of the command. Other techniques were also used to understand the behaviour of a command: in some cases, questions could be answered by reading the code. As a last resort, some specifiers used the very method that they had condemned when it was used by application programmers to explore a command’s behaviour – they wrote simple test programs!

When the specifier thought that he or she had a reasonably coherent idea of the behaviour of the commands, a model for the state of the system was formulated. This model was presented to a meeting of interested parties, together with some indications of how the operations could be described. Those present at this meeting included both Z experts and CICS experts, and the specifier received feedback from both groups. Taking into consideration the ideas expressed at this meeting, the specifier then produced a document which contained a draft version of the full specification. This was circulated to the same interested parties and feedback was again obtained. Based on these responses, a document was produced for a formal inspection. Such inspections are a normal part of the Hursley software development process, so all

the participants were used to them. Yet another version of the document was then produced, and, when the moderator of the inspection was satisfied that all points had been satisfactorily dealt with, a final version was produced and published as an IBM Hursley Technical Report.

15.6 Experiences

It is interesting to compare the problems that arose in this API specification project with those reported by Hayes in [17], which arose when the initial prototype API specifications were carried out in the early 1980s. Between the two projects, there had been several years of development of the Z language, to the extent that a standardisation effort was now under way. Several of the problems encountered in the earlier work on the API specification seem to have been solved, while others remain intractable.

15.6.1 Communication problems

The problems of communication mentioned in [17] were not as serious in the more recent work. In the intervening years, many IBM personnel had become familiar with the Z notation, and, of the people involved in writing the new specifications, all but one were experienced Z users. In fact, most of the specifiers were IBM employees who had been working at Hursley for several years, and so they had a good understanding of IBM and CICS terminology. Thus the communication problems between specifiers and CICS experts, which had been seen in the earlier work, were not too serious.

15.6.2 The ‘right’ level of abstraction

Of course, the problem mentioned by Hayes of finding the ‘right’ level of abstraction was still present in the more recent work: it is just one part of the problem of writing a ‘good’ specification. One of the difficulties in finding the right level of abstraction has already been mentioned, namely deciding whether a certain behaviour was part of the designer’s original intention, or whether it was ‘accidental’, in which case the specifier could justifiably abstract away from it.

A second major problem in making the specifications comprehensible lay in the actual complexity of the system being described. One technique that was found to be useful here was promotion; since many of the specifications were concerned with managing resources of various kinds, it was possible to give them the same overall structure. First a single, unnamed, instance of the resource was described – for instance, a temporary storage queue – together with the operations on that single instance – reading or writing an entry on the queue. Then a collection of named instances was described, and the operations on the full system were described, using a promotion schema and the previously defined operations on the single instance. The realisation that promotion was such a powerful technique led to more research work being carried out at Oxford on the mathematical ideas behind promotion [39, 64].

However, in those cases where it was not possible to use promotion, it was still necessary to structure the state. The state schema would often consist of a large number of variables and it would be necessary to define several smaller schemas and operations on them, in order to be able to construct comprehensible definitions of the operations on the whole state. Obviously there were many ways in which a large state

schema could be divided up, and choosing the best way involved deciding between a large number of small sub-schemas with easy operations, and a smaller number of larger sub-schemas with more complex operations. This decision was never easy, and often came down to a matter of taste. Sometimes, the only way to decide was to experiment with different divisions of the state.

15.6.3 Putting modules together

In essence, the problems mentioned by Hayes concerning putting the specifications of the separate parts of the API together were exactly the same in the more recent work. Once again, the simplistic view was that all that was necessary was to take the conjunction of the separate state schemas to form a state schema for the whole system, and to describe the effect of each API command on the whole state by showing its effect on the relevant small state, together with a constraint that none of the other small states should change. Unfortunately, in practice, the specifications were not entirely independent, and did not fit together quite so neatly. The first problem arose when the writer of the Interval Control specification realised just how complicated the link between transactions and terminals was. This led to his work on the Interval Control specification being suspended for about five months, while a new specification, on Transaction Management, was written. This specification was not part of the original plan, and it did not describe any API commands, but it was essential for describing not only Interval Control but also Program Control. It became one of the larger specifications written as part of this project.

The second problem with putting together the specifications was actually one which had been reported by Hayes in [17]. It concerned the interaction between the Condition Handling specification and other specifications, such as Temporary Storage, which contained commands which could raise ‘suspend conditions’. It is interesting to note that, although the problem had been identified during the initial prototype work on the API specifications, the more recent writers of the specifications had forgotten the problem, and were thus surprised to come across it in their work!

A third problem arose in the area of Storage Control. The CICS API provides operations `GETMAIN` and `FREEMAIN` for acquiring and releasing storage for application programs. One parameter to these commands is the address of the storage in question, so storage is naturally modelled as a partial function from addresses to bytes. There is a constraint that the storage acquired by one program must not be confused with the storage offered to another. Operations that may be described include modifying and inspecting the contents of storage. Unfortunately, the explicit nature of this model of storage does not fit well with the abstract way in which the other parameters of the API commands have been modelled. Input and output parameters are usually modelled by saying that a value is passed across the interface: in fact, of course, it is often the address of a piece of storage containing that value which is passed. There is also a problem since several of the commands which return data to the application program have an option whereby, instead of the data, they return the address of a piece of storage containing the data. Storage Control has internal operations to provide such storage for the relevant resource managers, and it is not difficult to model them. But now there is a problem with modelling the other method of obtaining the data, as an output sequence of bytes. With the explicit model of storage, we should model this as an input address, with the effect of the command being to modify storage at that address. This is clearly a less abstract description than is desirable, and it was eventually decided not to describe the effect of the `SET` option, which

returns an address, in the cases where it is an alternative to the INTO option, which returns the actual data.

A final problem with the integration of the different parts of the API specifications actually arose from the solution of a problem identified by Hayes. When the earlier work on the API specifications was carried out, it did not consider the recoverability of resources. In the later work, a decision was made at the start of the project that recoverability would be included. This made many of the states more complicated, as recoverable and non-recoverable resources behaved differently. It also meant that SYNCPOINT and SYNCPOINT ROLLBACK had to be described. These two commands could not be specified in the usual way, by their effect on a local ‘recoverability’ state, as this state was actually distributed around the different resource managers. So it was decided to describe the commands in many smaller pieces, each giving the effect of SYNCPOINT or SYNCPOINT ROLLBACK on the state of a single resource manager. The overall effect of the commands could then be obtained by taking the conjunction of these smaller descriptions.

15.6.4 Parallelism

The problem of dealing with parallelism which Hayes mentioned was dealt with in the more recent API specification work, using a notation that had been developed for the earlier work on the specification of the Common Programming Interface for Communications [66]. This notation was invented by the writers of the specification, and they described, informally, what they wanted it to mean. Formal semantics were then worked out at Oxford [63]. The notation was particularly intended to help in the specification of non-atomic operations – these are needed on occasions when the outcome of one program’s request depends on some action being taken by another program. Since operations in Z are usually interpreted as being atomic, a new notation was needed. Using this notation, we could define an operation *Op*:

$$\begin{aligned} Op &\hat{=} Op_atomic \\ &\square Op_front_end \rightarrow Op_back_end \end{aligned}$$

where *Op_atomic*, *Op_front_end* and *Op_back_end* are all schemas. Informally, this specification is interpreted as follows:² on execution, the precondition of either *Op_atomic* or *Op_front_end* must be true. If both are true, the choice between the branches is non-deterministic. If the precondition of *Op_atomic* is true, and the first branch is chosen, the effect of the command is as described by *Op_atomic*, and control is then returned to the program. If the precondition of *Op_front_end* is true, and the second branch is chosen, then *Op_front_end* has its effect, but control is not returned to the program. Instead the program is suspended until the precondition of *Op_back_end* is true – this can only happen because of the activity of other transactions in the system – then *Op_back_end* has its effect, and control is finally returned to the program.

15.6.5 Distributed systems

The final technical problem identified by Hayes was that of describing a distributed collection of CICS systems. It was decided at the outset of the recent work not to attempt to deal with this problem. It is an interesting specification problem to

²The formal semantics in [63] are defined in terms of action systems.

describe the effect of the remote execution of a command in a distributed CICS system, and one that may be attempted by Hursley and Oxford at some later stage.

15.7 Results

Since the aim of the API project was to produce specifications, it is not possible to give quantitative results for the success of the work, as was possible for the earlier use of Z as a basis for code development, described in Chapter 14. At the time of writing, the following areas of the API have been specified:

- automatic transaction initiation;
- basic mapping support;
- condition handling;
- interval control;
- program control;
- storage control;
- task control;
- temporary storage;
- terminal control;
- transactions and principal facilities;
- transient data.

These have all been published as IBM Hursley Technical Reports. Some use has been made of these reports as education material within Hursley: when developers or contractors have needed to learn about an area of CICS, the Z descriptions of the API have provided an unambiguous and well-structured source of information.

An interesting change in the development cycle for the API specification occurred during the course of the project. At the start of the project, the technology used to produce the specifications was fairly primitive: there were tools to help with the printing, previewing and cross-referencing of documents, but nothing as sophisticated as a type-checker. Midway through the project, a Z tool, running on the IBM PS/2 machine, became available for use on the project. The tool was based on the same markup language that had been used previously, but it incorporated a parser, type-checker and cross-referencer. Use of the tool meant that a specifier could be confident that the document presented for formal inspection contained no Z type errors. This had a marked effect on the nature of the inspections, since the inspectors were able to concentrate more on the meaning of the mathematics, rather than worrying about its syntax. The tool has also had an effect on the way in which inspectors prepare for inspections: instead of reading a hard copy of the document, they can read it online, using the tool's facilities for schema expansion and definition finding to help them navigate around the document.

The success or otherwise of this project may be measured in the future by seeing what use is made of the specifications that have been written. There are some plans

to provide Z education for possible audiences for the specifications: designers and developers of CICS application programs and packages, those involved with CICS education, etc. It will be interesting to see what decision these audiences make about the investment needed to understand the specifications, as compared to the benefits to be gained by so doing.

15.8 Conclusions

This chapter has reported how the Z notation was used to write a formal description of a programming interface. This was not a specification from which code was to be developed, but a precise record of an already existing interface. The specification was written for commercial reasons, since it was felt that a formal specification would greatly improve the interaction between the supplier and the user of the interface, reducing the chance of a user relying on accidental behaviours.

Rather than writing one large specification to describe the whole interface, several smaller documents were written, each covering a part of the interface. The interactions between these documents revealed some interesting relationships between different parts of CICS. The Z notation was sufficiently powerful to describe the concepts required, and the technique of promotion was found particularly useful in structuring many of the specifications.

Acknowledgements The specifiers who worked on the API specification project were David Blyth, Steve King, Jonathan Hoare, Iain Houston and John Wordsworth, and Mark Phillips was the manager responsible. I am grateful to them, and to Pete Collins, Ian Hayes and Tony Hoare, for their comments on this chapter.

Many others, both at Oxford and Hursley, have been involved with the use of Z at IBM Hursley: a list of contributors may be found in Chapter 14.

Chapter 16

CICS Temporary Storage

Ian Hayes

Abstract CICS Temporary Storage provides facilities for storage of information in named queues. The operations that can be performed on an individual queue are either the standard queue-like operations (append to the end and remove from the beginning), or array-like random access read and write operations.

16.1 A single queue

An element of a queue is a sequence of bytes:

$$\begin{aligned} \textit{BYTE} &== 0 \dots 255 \\ \textit{TSElem} &== \text{seq } \textit{BYTE} \end{aligned}$$

A single queue may be defined by the following schema:

\textit{TSQ}
$ar : \text{seq } \textit{TSElem}$
$p : \mathbb{N}$
$p \leq \#ar$

The sequence ar contains the items in the queue. The size of the sequence is always equal to the number of append operations that have been performed on the queue since its creation – independent of the number of other (remove, read or write) operations. The pointer p keeps track of the position of the item which was last removed or read from the queue. The initial state of a queue is given by an empty sequence and a zero pointer.

$$\textit{TSQ_Initial} \hat{=} [\textit{TSQ} \mid ar = \langle \rangle \wedge p = 0]$$

16.1.1 Operations

We define four operations on a single queue. The definitions of these operations use the following schema:

$$\Delta TSQ \triangleq TSQ \wedge TSQ'$$

ΔTSQ (Δ for change) defines a before state TSQ , with components ar and p (satisfying $p \leq \#ar$), and an after state TSQ' , with components ar' and p' (satisfying $p' \leq \#ar'$). The definitions of the operations follow.

<i>Append0</i>	
ΔTSQ	
$from? : TSElem$	
$item! : \mathbb{Z}$	
	$ar' = ar \frown \langle from? \rangle \wedge$ $item! = \#ar' \wedge$ $p' = p$

The new element $from?$ is appended to the end of ar to give the new value of the sequence. The position of the new item is returned in $item!$. The pointer position is unchanged.

<i>Remove0</i>	
ΔTSQ	
$into! : TSElem$	
	$p < \#ar \wedge$ $p' = p + 1 \wedge$ $into! = ar(p') \wedge$ $ar' = ar$

The pointer must not already have reached the end of the sequence. The pointer is incremented to indicate the next item in the queue and the value of that item is returned in $into!$. The contents of the sequence are unchanged.

<i>Write0</i>	
ΔTSQ	
$item? : \mathbb{Z}$	
$from? : TSElem$	
	$item? \in 1.. \#ar \wedge$ $ar' = ar \oplus \{item? \mapsto from?\} \wedge$ $p' = p$

The position $item?$ must lie within the bounds of the current sequence. The item at that position in ar is overridden by the value of $from?$ to give the new value of the sequence. The pointer position is unchanged.

<i>Read0</i>	
ΔTSQ	
$item? : \mathbb{Z}$	
$into! : TSElem$	
$item? \in 1 \dots \#ar \wedge$	
$into! = ar(item?) \wedge$	
$p' = item? \wedge$	
$ar' = ar$	

The value of the item at position $item?$, which must lie within the bounds of the sequence, is returned in $into!$. The pointer position is updated to the value of $item?$. The sequence is unchanged.

In the above, all of the operations have been specified in terms of the sequence ar and pointer p . While this is reasonable for the *Read* and *Write* operations it does not show the queue-like nature of the *Append* and *Remove* operations. Let us now show that the queue-like operations are the familiar ones. We can define a standard queue by

$$Q == \text{seq } TSElem$$

Operations on queues refer to before and after components q and q' , respectively.

$$\Delta Q \hat{=} [q, q' : Q]$$

The standard ‘append to the end of a queue’ operation is given by

<i>Standard_Append</i>	
ΔQ	
$from? : TSElem$	
$q' = q \frown \langle from? \rangle$	

The standard ‘remove from the front’ of the queue operation is given by

<i>Standard_Remove</i>	
ΔQ	
$into! : TSElem$	
$q = \langle into! \rangle \frown q'$	

The predicate in the above specification may be unconventional to some readers. It states that the value of the queue before the operation is equal to the value returned in $into!$ concatenated with the value of the queue after the operation. This form of specification more closely reflects the symmetry between *Standard_Append* and *Standard_Remove* than the more conventional and more operational

$$into! = head(q) \wedge q' = tail(q)$$

To see the relationship between standard queues and temporary storage queues we need to formulate the correspondence between the respective states.

$QLike$	
$q : Q$	
TSQ	
$q = tail^p(ar)$	

A standard q corresponds to the sequence ar with the first p elements removed. Given the relationship $QLike$ we can show the relationship between $Append0$ and $Standard_Append$. This is formalised by the following theorem.

Theorem If an $Append0$ is performed with initial state TSQ and final state TSQ' , then the corresponding standard queue states Q and Q' (as determined by $QLike$ and $QLike'$ respectively) are related by $Standard_Append$.

$$Append0 \wedge QLike \wedge QLike' \vdash Standard_Append$$

Proof

1. $q, q' : Q$ $QLike, QLike'$
2. $from? : TSElem$ $Append0$
3. $q' = tail^{p'}(ar')$ $QLike'$

$$= tail^p(ar \frown \langle from? \rangle)$$
 $Append0$

$$= (tail^p(ar)) \frown \langle from? \rangle$$
 as $p \leq \#ar$ from TSQ

$$= q \frown \langle from? \rangle$$
 $QLike$

$$Standard_Append$$
 (1), (2), (3)

□

We can now do the same for $Remove$.

Theorem If an $Remove0$ is performed with initial state TSQ and final state TSQ' , then the corresponding standard queue states Q and Q' (as determined by $QLike$ and $QLike'$ respectively) are related by $Standard_Remove$.

$$Remove0 \wedge QLike \wedge QLike' \vdash Standard_Remove$$

Proof

1. $q, q' : Q$ $QLike, QLike'$
2. $into! : TSElem$ $Remove0$
3. $p < \#ar$ $Remove0$
4. $q = tail^p(ar)$ $QLike$

$$= \langle ar(p+1) \rangle \frown (tail^{p+1}(ar))$$
 (3), property of tail

$$= \langle into! \rangle \frown (tail^{p'}(ar'))$$
 $Remove0$

$$= \langle into! \rangle \frown q'$$
 $QLike'$

$$Standard_Remove$$
 (1), (2), (4)

□

16.1.2 Errors

To cope with errors we can introduce a report to indicate success or failure of an operation. The set *CONDITION* contains all the error reports plus the report *Success*.

$$CONDITION ::= Success \mid ItemErr \mid NoSpace \mid QIdErr \mid SysIdErr$$

If an error occurs we would like the queue to remain unchanged.

$\frac{ERROR \quad \Delta TSQ \quad report! : CONDITION}{\theta TSQ' = \theta TSQ}$

In the operations described above there are three errors that can occur: trying to remove an item from a queue with no items left to remove, trying to read or write at a position outside the sequence, and running out of space to store an item.

$\frac{NoneLeft \quad ERROR}{p = \#ar \wedge report! = ItemErr}$
--

$\frac{OutofBounds \quad ERROR \quad item? : \mathbb{Z}}{item? \notin 1 \dots \#ar \wedge report! = ItemErr}$

$\frac{OutOfSpace \quad ERROR}{report! = NoSpace}$
--

If the operations work correctly the report indicates *Success*.

$$Successful \triangleq [report! : CONDITION \mid report! = Success]$$

The operations given previously can now be combined with the erroneous situations. We redefine the operations in terms of their previous definitions.

$$\begin{aligned} Append &\triangleq (Append0 \wedge Successful) \vee OutOfSpace \\ Remove &\triangleq (Remove0 \wedge Successful) \vee NoneLeft \\ Write &\triangleq (Write0 \wedge Successful) \vee OutofBounds \vee OutOfSpace \\ Read &\triangleq (Read0 \wedge Successful) \vee OutofBounds \end{aligned}$$

Note that *OutOfSpace* does not specify under what conditions it occurs. The specifications of *Append* and *Write* do not allow us to determine whether or not the

operation will be successful from the initial state and inputs to an operation. This is an example of a non-deterministic specification. It is left to the implementor to determine when a *NoSpace* report will be returned (we hope it will not be on every call).

At the level of abstraction of this description we have no knowledge of the space required for storing queues and hence this is not the appropriate place to define under what conditions this error can occur. At some stage during implementation the conditions under which this error can occur can be defined. At this point the implementer and client should get together once more to make sure they agree on the definition.

16.2 Named queues

We now want to specify a system with more than one queue. A particular queue can be specified by name and the above operations can be performed on it. We use a mapping from queue names

$$[TSQName]$$

to queues. The state of our system of queues is given by

$$TS == TSQName \leftrightarrow TSQ$$

The initial state of the system of queues is given by an empty mapping.

$$\frac{TS_Initial : TS}{TS_Initial = \{}}$$

The before and after states of the operations are ts and ts' , respectively.

$$\Delta TS \triangleq [ts, ts' : TS]$$

Our operations require the updating of a particular named temporary storage queue. We introduce a framing schema, ΦTS , to encapsulate the common part of updating for operations on queues that already exist.

$$\frac{\Phi TS \quad \begin{array}{l} queue? : TSQName \\ \Delta TS \\ \Delta TSQ \end{array}}{\begin{array}{l} queue? \in \text{dom}(ts) \wedge \\ \theta TSQ = ts(queue?) \wedge \\ ts' = ts \oplus \{queue? \mapsto \theta TSQ'\} \end{array}}$$

Note that ΦTS specifies that the named queue (alone) is updated but does not specify in what way it is updated. The latter is achieved by combining ΦTS with the single queue operations to get the operation on named queues.

In adding named queues we have added the possibility of a new error: trying to perform operations on non-existent queues.

<i>NonExistent</i>
ΔTS $queue? : TSQName$ $report! : CONDITION$
$queue? \notin \text{dom}(ts) \wedge$ $ts' = ts \wedge$ $report! = QIdErr$

Our single queue operations, except *AppendQ* which is allowed on a non-existent queue, can now be redefined in terms of our previous definitions.

$$\begin{aligned}
RemoveQ &\hat{=} (\exists \Delta TSQ \bullet \Phi TS \wedge Remove) \vee NonExistent \\
WriteQ &\hat{=} (\exists \Delta TSQ \bullet \Phi TS \wedge Write) \vee NonExistent \\
ReadQ &\hat{=} (\exists \Delta TSQ \bullet \Phi TS \wedge Read) \vee NonExistent
\end{aligned}$$

The auxiliary variables in ΔTSQ (ar, p, ar', p') are hidden in the signatures of the final operations and the operations inherit the errors from the equivalent single queue operations.

A queue is created by performing an *AppendQ* operation on a queue that does not exist. The following schema describes the creation of a queue:

<i>CreateQ</i>
ΔTS $queue? : TSQName$ $TSQ_Initial$ TSQ'
$queue? \notin \text{dom}(ts) \wedge$ $ts' = ts \cup \{queue? \mapsto \theta TSQ'\}$

The relationship between $TSQ_Initial$ (ar, p) and TSQ' (ar', p') is not defined within this schema. This is supplied by *Append* in the following definition:

$$AppendQ \hat{=} (\exists \Delta TSQ \bullet (\Phi TS \vee CreateQ) \wedge Append)$$

Note that for a non-existent queue, if an error occurs (i.e. a *NoSpace* condition), then an empty queue will be created.

In addition to these promoted operations on named queues we have an operation to delete a named queue.

<i>DeleteQ0</i>
ΔTS $queue? : TSQName$ $report! : CONDITION$
$queue? \in \text{dom}(ts) \wedge$ $ts' = \{queue?\} \triangleleft ts \wedge$ $report! = Success$

An exception occurs if the queue to be deleted does not exist; *DeleteQ* becomes

$$DeleteQ \hat{=} DeleteQ0 \vee NonExistent$$

16.3 A network of systems

Temporary storage queues may be located on more than one system. Let us call the set of all possible system identifiers $SysId$.

$$[SysId]$$

We can represent temporary storage queues on a network of systems by

$$NTS == SysId \leftrightarrow TS$$

For a network, $nts : NTS$, $\text{dom}(nts)$ is the set of systems that share temporary storage queues and for a system with identity $sysid$ such that $sysid \in \text{dom}(nts)$, $nts(sysid)$ is the temporary storage state of that system. A change of network state uses the following schema:

$$\Delta NTS \triangleq [nts, nts' : NTS]$$

The operations on temporary storage queues may be promoted to operate for a network of systems by the following framing schema:

$\frac{\Phi NTS \quad \Delta NTS \quad sysid? : SysId \quad \Delta TS}{sysid? \in \text{dom}(nts) \wedge ts = nts(sysid?) \wedge nts' = nts \oplus \{sysid? \mapsto ts'\}}$

As with the promotion of operations to work on named queues, the above schema only specifies which system is updated but not how it is updated. The latter is supplied when ΦNTS is combined with the definitions of the operations on a single system.

Network operation also introduces the possibility of an error if the given system does not exist.

$\frac{NoSystem \quad \Delta NTS \quad sysid? : SysId \quad report! : CONDITION}{sysid? \notin \text{dom}(nts) \wedge nts' = nts \wedge report! = SysIdErr}$
--

The operations on a multiple system are given by

$$\begin{aligned} AppendQN0 &\triangleq (\exists \Delta TS \bullet AppendQ \wedge \Phi NTS) \vee NoSystem \\ RemoveQN0 &\triangleq (\exists \Delta TS \bullet RemoveQ \wedge \Phi NTS) \vee NoSystem \\ ReadQN0 &\triangleq (\exists \Delta TS \bullet ReadQ \wedge \Phi NTS) \vee NoSystem \\ WriteQN0 &\triangleq (\exists \Delta TS \bullet WriteQ \wedge \Phi NTS) \vee NoSystem \end{aligned}$$

The *sysid?* and *queue?* name supplied as inputs are not necessarily those on which an operation takes place. A queue name on a given system may be marked as actually being located on another (remote) system, possibly with a different name on that system. We model this by the function *remote* which takes an input pair (*sysid?*, *queue?*) and gives the corresponding actual pair (*sysid!*, *queue!*) on which the operation is performed

$$| \quad \text{remote} : (\text{SysId} \times \text{TSQName}) \rightarrow (\text{SysId} \times \text{TSQName})$$

In many cases the input *sysid?* and *queue?* name are the actual system and queue name; in these cases *remote* behaves as the identity function.

We use the following schema to incorporate *remote* into the operations:

$\begin{array}{l} \text{TSRemote} \\ \hline \text{sysid?}, \text{sysid!} : \text{SysId} \\ \text{queue?}, \text{queue!} : \text{TSQName} \\ \hline (\text{sysid!}, \text{queue!}) = \text{remote}(\text{sysid?}, \text{queue?}) \end{array}$
--

The outputs, *sysid!* and *queue!*, of *TSRemote* form the inputs to the operations. If a *sysid?* parameter is supplied then the operations on temporary storage queues are defined by

$$\begin{aligned} \text{AppendQN1} &\hat{=} \text{TSRemote} \gg \text{AppendQN0} \\ \text{RemoveQN1} &\hat{=} \text{TSRemote} \gg \text{RemoveQN0} \\ \text{ReadQN1} &\hat{=} \text{TSRemote} \gg \text{ReadQN0} \\ \text{WriteQN1} &\hat{=} \text{TSRemote} \gg \text{WriteQN0} \end{aligned}$$

Recall that the schema operator ‘ \gg ’ (piping) identifies the outputs (variables ending in ‘!’) of its left operand with the inputs (variables ending in ‘?’) of its right operand; these variables are hidden in the result. All other components are combined together as for schema conjunction (\wedge).

If no *sysid?* parameter is given then the operations are given by

$$\begin{aligned} \text{AppendQN2} &\hat{=} \text{AppendQN1}[\text{cursysid?}/\text{sysid?}] \\ \text{RemoveQN2} &\hat{=} \text{RemoveQN1}[\text{cursysid?}/\text{sysid?}] \\ \text{ReadQN2} &\hat{=} \text{ReadQN1}[\text{cursysid?}/\text{sysid?}] \\ \text{WriteQN2} &\hat{=} \text{WriteQN1}[\text{cursysid?}/\text{sysid?}] \end{aligned}$$

That is, the *sysid?* parameter is replaced by a parameter giving the identity of the current system (the system on which the operation was initiated).

16.3.1 A note on the current implementation

Each system keeps track of the names of queues that are located on other (remote) systems and, for each remote queue, the identity of the remote system and the name of the queue on that system. It is possible that the referred request could be for a queue name that is also remote to the referred system, in which case the request is further referred to another system. To find the system on which the queue actually resides we need to follow through a chain of systems until we get to a system on

which the queue name is considered local. We can model the implementation by the function

$$\mid \text{rem} : (\text{SysId} \times \text{TSQName}) \leftrightarrow (\text{SysId} \times \text{TSQName})$$

which for a *sysid* and queue name gives the *sysid* and queue name of the next link in the chain; if a *sysid* and queue name pair is not in the domain of *rem*, then the chain is finished. The function *remote* should be the same as the repeated application of *rem* until the result is no longer in its domain. We define the function *repeat*, which repeatedly applies its argument (another function) until the result is no longer in the domain of the argument function.

$$\begin{array}{|l} \hline [X] \\ \hline \text{repeat} : (X \leftrightarrow X) \leftrightarrow (X \rightarrow X) \\ \hline \text{dom repeat} = \{f : X \leftrightarrow X \mid \neg (\exists x : X \bullet (x, x) \in (f^+))\} \\ \forall f : \text{dom repeat}; x : X \bullet \\ \quad (x \notin \text{dom } f \Rightarrow (\text{repeat } f)(x) = x) \wedge \\ \quad (x \in \text{dom } f \Rightarrow (\text{repeat } f)(x) = (\text{repeat } f)(f \ x)) \\ \hline \end{array}$$

The relationship between *remote* and *rem* is simply

$$\text{remote} = \text{repeat}(\text{rem})$$

that is,

$$\begin{array}{l} \forall s : \text{SysId}; q : \text{TSQName} \bullet \\ \quad (s, q) \notin \text{dom rem} \Rightarrow \text{remote}(s, q) = (s, q) \wedge \\ \quad (s, q) \in \text{dom rem} \Rightarrow \text{remote}(s, q) = \text{remote}(\text{rem}(s, q)) \end{array}$$

Since *remote* is a total function the equality of *remote* and *repeat(rem)* requires that no chain of *rem*'s contains a loop (so that *repeat(rem)* is also total).

Given the function *rem*, if we take the corresponding (curried) function *r*

$$\begin{array}{|l} \hline r : \text{SysId} \rightarrow (\text{TSQName} \leftrightarrow (\text{SysId} \times \text{TSQName})) \\ \hline \forall s : \text{SysId}; q : \text{TSQName} \bullet \\ \quad \text{dom}(r \ s) = \{q : \text{TSQName} \mid (s, q) \in \text{dom rem}\} \wedge \\ \quad (q \in \text{dom}(r \ s) \Rightarrow r(s)(q) = \text{rem}(s, q)) \\ \hline \end{array}$$

the mapping that needs to be stored on each system *s* is given by *r(s)*, and is of type

$$\text{TSQName} \leftrightarrow (\text{SysId} \times \text{TSQName})$$

Acknowledgements The work reported in this chapter was supported by a grant from IBM. The starting point was an earlier specification created by Tim Clement. This specification has benefited greatly from the detailed comments of Carroll Morgan and Ib Holm Sørensen.

Chapter 17

CICS message system

Ian Hayes

Abstract The following message system is based on the message handling in CICS. The specification itself is an interesting example: it combines states (of input and output devices), and gives a number of examples of the use of the piping operator, ‘ \gg ’, on schemas.

17.1 Message output

We can represent a set of output devices by a mapping from a device name, from the set

$$[Name]$$

to a sequence of messages, from the set

$$[Message]$$

that have been output to that device.

$\begin{array}{l} NOUT \\ noq : Name \mapsto \text{seq } Message \end{array}$

The operations on output that are discussed here neither create nor destroy devices.

$$\Delta NOUT \triangleq [NOUT; NOUT' \mid \text{dom } noq' = \text{dom } noq]$$

Sending a message to a device simply appends the message to the queue for that device.

$NSend_0$
$\Delta NOUT$
$n? : Name$
$m? : Message$
$n? \in \text{dom } noq \wedge$
$noq' = noq \oplus \{n? \mapsto noq(n?) \cap \langle m? \rangle\}$

17.2 Multiple destinations

A message may be sent to a set of destinations.

$NSendM_0$
$\Delta NOUT$
$ns? : \mathbb{P} Name$
$m? : Message$
$ns? \subseteq \text{dom } noq \wedge$
$noq' = noq \oplus \{n : ns? \bullet n \mapsto noq(n) \cap \langle m? \rangle\}$

All of the names in $ns?$ must correspond to valid output devices. The message is sent to each device in $ns?$.

Theorem Given

$$ToSet \hat{=} [n? : Name; ns! : \mathbb{P} Name \mid ns! = \{n?\}]$$

$$ToSet_NSendM_0 \hat{=} ToSet \ggg NSendM_0$$

then

$$NSend_0 \Leftrightarrow ToSet_NSendM_0$$

Recall that the schema operator ‘ \ggg ’ identifies the outputs (variables ending in ‘!’) of its left operand with the inputs (variables ending in ‘?’) of its right operand; these variables are hidden in the result. All other components are combined together as for schema conjunction (\wedge).

17.3 Message input

We can represent a set of input devices by a mapping from a device name to a sequence of messages yet to be input from that device.

NIN
$niq : Name \leftrightarrow \text{seq } Message$

The operations on input described here neither create nor destroy devices.

$$\Delta NIN \hat{=} [NIN; NIN' \mid \text{dom } niq' = \text{dom } niq]$$

Receiving a message from a device simply removes it from the head of the input queue for that device.

$$\begin{array}{c}
 \hline
 NReceive_0 \\
 \Delta NIN \\
 n? : Name \\
 m! : Message \\
 \hline
 m! = head(niq(n?)) \wedge \\
 niq' = niq \oplus \{n? \mapsto tail(niq(n?))\} \\
 \hline
 \end{array}$$

17.4 Send and receive

We can define an operation that both sends a message to a device and receives a message from that device.

$$NSendReceive_0 \triangleq NSend_0 \wedge NReceive_0$$

17.5 Combining input and output

We introduce $NDEV$ to describe the combined input and output state for all of the devices. If a device can be used for input then it must be able to be used for output.

$$\begin{array}{c}
 \hline
 NDEV \\
 NIN \\
 NOUT \\
 \hline
 \text{dom } niq \subseteq \text{dom } noq \\
 \hline
 \end{array}$$

$$\Delta NDEV \triangleq NDEV \wedge NDEV'$$

Input and output operations preserve the output and input states respectively.

$$\begin{aligned}
 \Xi NOUT &\triangleq [\Delta NDEV \mid \theta NOUT' = \theta NOUT] \\
 \Xi NIN &\triangleq [\Delta NDEV \mid \theta NIN' = \theta NIN]
 \end{aligned}$$

The operations on the combined state follow.

$$\begin{aligned}
 NSend &\triangleq NSend_0 \wedge \Xi NIN \\
 NSendM &\triangleq NSendM_0 \wedge \Xi NIN \\
 NReceive &\triangleq NReceive_0 \wedge \Xi NOUT \\
 NSendReceive &\triangleq NSendReceive_0 \wedge \Delta NDEV
 \end{aligned}$$

17.6 Logical names

Rather than work with actual (physical) device names, as we have up to this point, we would like to work with logical names, from the set

$$[LName]$$

The logical names are mapped into physical device names by the function $ltop$.

$$\frac{LtoP}{ltop : LName \leftrightarrow Name}$$

None of the operations discussed here modify the mapping from logical names to physical names. Hence we will use the following schema:

$$\exists LtoP \triangleq [LtoP; LtoP' \mid \theta LtoP' = \theta LtoP]$$

If a logical name corresponds to an actual device we perform the operation on that device, otherwise we use the device with physical name *console*.

$$\begin{array}{|l} console : Name \\ \hline \frac{MapName}{\exists LtoP} \\ dev : Name \leftrightarrow \text{seq Message} \\ ln? : LName \\ n! : Name \\ \hline ln? \in \text{dom}(ltop \circ dev) \Rightarrow n! = ltop(ln?) \wedge \\ ln? \notin \text{dom}(ltop \circ dev) \Rightarrow n! = console \end{array}$$

The operations on a single device become as follows:

$$\begin{aligned} LSend &\triangleq MapName[noq/dev] \gg NSend \\ LReceive &\triangleq MapName[niq/dev] \gg NReceive \\ LSendReceive &\triangleq MapName[niq/dev] \gg NSendReceive \end{aligned}$$

17.7 Multiple logical destinations

To send a message to a set of logical names we need to map the set of logical names into physical names. If none of the logical names correspond to a device we send the message to the device with physical name *console*.

$$\begin{array}{|l} MapSet \\ \hline \exists LtoP \\ lns? : \mathbb{P} LName \\ ns! : \mathbb{P} Name \\ NOUT \\ \hline \text{let } names == ltop(\downarrow lns?) \cap \text{dom } noq \bullet \\ names = \{\} \Rightarrow ns! = \{console\} \wedge \\ names \neq \{\} \Rightarrow ns! = names \end{array}$$

The operation to send a message to a set of logical devices is

$$LSendM \triangleq MapSet \gg NSendM$$

Theorem Given

$$\begin{aligned} ToSetL &\triangleq [ln? : LName; lns! : \mathbb{P} LName \mid lns! = \{ln?\}] \\ ToSetL_LSendM &\triangleq ToSetL \gg LSendM \end{aligned}$$

then

$$LSend \Leftrightarrow ToSetL_LSendM$$

17.8 Domains of the operations

In practice we would like all the operations to be total (defined for all inputs). Unfortunately, this is not the case. If a name (or a set of names) does not correspond to an actual device, then the name is translated to the special device *console*; if the *console* does not exist, the operation is not defined. For the output operations, ensuring that the *console* exists is a sufficient pre-condition for the operation to be defined (we also need this pre-condition for input).

$$Pre \triangleq [NDEV; LtoP; m? : Message \mid console \in \text{dom } niq]$$

Remember that $\text{dom } niq \subseteq \text{dom } noq$, so $console \in \text{dom } noq$.

Theorems

$$\begin{aligned} Pre &\Rightarrow \text{pre } LSend \\ Pre &\Rightarrow \text{pre } LSendM \end{aligned}$$

For the input operations we need the additional requirement that the queue of messages yet to be input on the device is not empty.

$$PreIn \triangleq [Pre; n? : Name \mid niq(n?) \neq \langle \rangle]$$

Theorems Given

$$MapName_PreIn \triangleq MapName[niq/dev] \gg PreIn$$

then

$$\begin{aligned} MapName_PreIn &\Rightarrow \text{pre } LReceive \\ MapName_PreIn &\Rightarrow \text{pre } LSendReceive \end{aligned}$$

Acknowledgement This specification is based on a message system specified by David Renshaw of IBM (UK) Laboratories, Hursley, England.

Part V

APPENDICES

Appendix A

Glossary: Z mathematical notation

A.1 Definitions and declarations

Let $x, x_1, x_2, \dots, x_n, X, X_1, X_2, \dots, X_n$ be identifiers and let T, T_1, T_2, \dots, T_n be set-valued expressions.

$LHS == RHS$

Definition of LHS as equivalent to RHS . A definition is distinguished from an equality ($=$) syntactically by the use of the symbol $==$. A definition *defines* the left side to be equivalent to the right side, while an equality is a predicate that is either true or false.

$LHS[X_1, X_2, \dots, X_n] == RHS$

Generic definition of LHS , where X_1, X_2, \dots, X_n are variables denoting formal parameter sets. When the LHS is used the actual parameter sets are supplied by placing them in square brackets after the use:

$LHS[T_1, T_2, \dots, T_n]$

Commonly the context uniquely determines the choice of parameter sets; in these cases the parameter sets may be omitted.

$x : T$

A declaration, $x : T$, introduces a new variable x of type T . This should be distinguished from the membership test, $x \in T$, which is a predicate that is either true or false.

$x_1 : T_1; x_2 : T_2; \dots; x_n : T_n$

List of declarations.

$x_1, x_2, \dots, x_n : T$

$== x_1 : T; x_2 : T; \dots; x_n : T$

$[X_1, X_2, \dots, X_n]$

Introduction of basic types named X_1, X_2, \dots, X_n . These are distinct new types whose structure is not constrained by this introduction, but may be constrained by predicates in the remainder of a specification.

The glossaries may be copied for educational purposes.

A.2 Axiomatic definitions

Let D be a list of declarations and P a predicate.

The following axiomatic definition introduces the variables in D with the types as declared in D . These variables must also satisfy the predicate P . The scope of the variables is the whole specification.

D	
P	

For example,

$small, large : \mathbb{N}$	
$small < large$	

introduces two natural number variables $small$ and $large$ such that the value of $small$ is less than the value of $large$.

The predicate part of an axiomatic definition is optional. For example, a global variable $MaxSize$ may be introduced by the following axiomatic definition.

$MaxSize : \mathbb{N}$	
------------------------	--

This variable may later be constrained by (global) predicates occurring in the specification.

$$MaxSize < 100$$

A.3 Generic definitions

Let D be a list of declarations, P a predicate and X_1, X_2, \dots, X_n variable names standing for generic sets.

The following generic definition is similar to an axiomatic definition, except that the variables introduced are generic over the sets X_1, X_2, \dots, X_n .

$[X_1, X_2, \dots, X_n]$	
D	
P	

The declared variables must be uniquely defined by the predicate P .

For example, the definitions of $head$ and $tail$ on sequences are given by the following:

$[X]$	
$head : seq_1 X \rightarrow X$	
$tail : seq_1 X \rightarrow X$	
$\forall x : X; s : seq X \bullet$	
$head (\langle x \rangle \frown s) = x \wedge$	
$tail (\langle x \rangle \frown s) = s$	

For $s : seq_1 \mathbb{N}$, the first element of s is given by $head[\mathbb{N}](s)$. As the type of s uniquely determines the choice of generic parameter in this case, the parameter may be elided: $head(s)$.

A.4 Logic

Let P, Q be predicates; D a declaration or a list of declarations; T, T_1, \dots, T_n set-valued expressions; x, y, x_1, \dots, x_n variables; and t, t_1, \dots, t_n expressions.

$true, false$	Logical constants.
$\neg P$	Negation: ‘not P ’.
$P \wedge Q$	Conjunction: ‘ P and Q ’.
$P \vee Q$	Disjunction: ‘ P or Q or both’.
$P \Rightarrow Q$	$== (\neg P) \vee Q$ Implication: ‘ P implies Q ’ or ‘if P then Q ’.
$P \Leftrightarrow Q$	$== (P \Rightarrow Q) \wedge (Q \Rightarrow P)$ Equivalence: ‘ P is logically equivalent to Q ’.
$\forall x : T \bullet P$	Universal quantification: ‘for all x in the set T , P holds’. The scope of the variable x is the quantified predicate P . In Z, the scope of quantifiers extends as far to the right as is possible and hence the quantifier may need to be enclosed in parentheses to limit the scope. All quantifiers in Z specify the type of the bound variable, and hence explicitly define the values over which the quantification ranges. If the set T is empty, then the quantification is vacuously true.
$\exists x : T \bullet P$	Existential quantification: ‘there exists an x in the set T such that P ’. If the set T is empty then the quantification is false.
$\exists_1 x : T \bullet P$	$== (\exists y : T \bullet (\forall x : T \bullet x = y \Leftrightarrow P))$ Unique existence: ‘there exists a unique x in the set T such that P holds’. Note that in the definition above, y is assumed to be a fresh variable other than x and not occurring in P .
$\forall x_1 : T_1; x_2 : T_2; \dots; x_n : T_n \bullet P$	‘For all x_1 in T_1 , x_2 in T_2 , \dots , and x_n in T_n , P holds.’ Note that the set expressions T_1, \dots, T_n may not refer to any of the variables x_1, \dots, x_n being declared. All variables referenced in T_1, \dots, T_n have to be defined globally to the whole quantified expression. This is to avoid ambiguity because the same variable name could occur both in the declarations and global to the scope of the quantifier.
$\exists x_1 : T_1; x_2 : T_2; \dots; x_n : T_n \bullet P$	‘There exist x_1 in T_1 , x_2 in T_2 , \dots , and x_n in T_n , such that P holds.’
$\exists_1 x_1 : T_1; x_2 : T_2; \dots; x_n : T_n \bullet P$	‘There exist unique x_1 in T_1 , unique x_2 in T_2 , \dots , and unique x_n in T_n , such that P holds.’
$\forall D \mid P \bullet Q$	$== \forall D \bullet P \Rightarrow Q$
$\exists D \mid P \bullet Q$	$== \exists D \bullet P \wedge Q$

$$\exists_1 D \mid P \bullet Q == \exists_1 D \bullet P \wedge Q$$

$$P[t/x] \text{ or } P\left[\frac{t}{x}\right]$$

Substitution: the predicate P with every free occurrence of the variable x replaced by the expression t , e.g.

$$(x = y)[x + 1/x] \Leftrightarrow (x + 1 = y)$$

Substitution can also be applied to expressions.

$$P[t_1, \dots, t_n/x_1, \dots, x_n] \text{ or } P\left[\frac{t_1, \dots, t_n}{x_1, \dots, x_n}\right]$$

Simultaneous substitution of t_1 for x_1 , \dots , t_n for x_n , e.g.

$$(x < y)\left[\frac{y, x}{x, y}\right] \Leftrightarrow (y < x)$$

$$t_1 = t_2 \quad \text{Equality between expressions.}$$

$$t_1 \neq t_2 \quad == \neg (t_1 = t_2)$$

A.5 Sets

Let X be a set; S and T be subsets of X ; T_1, \dots, T_n set-valued expressions; t, t_1, \dots, t_n expressions; x, x_1, \dots, x_n variables; P a predicate; and D declarations.

$$t \in S \quad \text{Set membership: ‘} t \text{ is a member of } S \text{’.$$

$$t \notin S \quad == \neg (t \in S)$$

$$S \subseteq T \quad == (\forall x : S \bullet x \in T)$$

Set inclusion.

$$S \subset T \quad == S \subseteq T \wedge S \neq T$$

Strict set inclusion.

$$\{\} \text{ or } \emptyset \quad \text{The empty set.}$$

$$\{t_1, t_2, \dots, t_n\}$$

The set containing the values of expressions t_1, t_2, \dots, t_n . Note that duplication of values in the list is allowed but duplicates do not change the value of the set.

$$\{x : T \mid P\} \quad \text{The set containing exactly those } x \text{ in the set } T \text{ for which } P \text{ holds.}$$

$$(t_1, t_2, \dots, t_n)$$

Ordered n -tuple of t_1, t_2, \dots, t_n .

$$T_1 \times T_2 \times \dots \times T_n$$

Cartesian product: the set of all n -tuples such that the k th component is of type T_k .

$$first(t_1, t_2) == t_1$$

$$second(t_1, t_2) == t_2$$

$\{x_1 : T_1; x_2 : T_2; \dots; x_n : T_n \mid P\}$	The set of all n -tuples (x_1, x_2, \dots, x_n) with each x_k of type T_k such that P holds.
$\{D \mid P \bullet t\}$	The set of values of the expression t for the variables declared in D ranging over all values for which P holds. For example, given sets of integers S and T the set of sums of pairs of integers, one taken from S and one taken from T , such that the sum is strictly positive is given by $\{n : S; m : T \mid n + m > 0 \bullet n + m\}$.
$\{D \bullet t\}$	$== \{D \mid \text{true} \bullet t\}$
$S \cap T$	$== \{x : X \mid x \in S \wedge x \in T\}$ Set intersection.
$S \cup T$	$== \{x : X \mid x \in S \vee x \in T\}$ Set union.
$S \setminus T$	$== \{x : X \mid x \in S \wedge x \notin T\}$ Set difference.
$\mathbb{P} S$	Powerset: the set of all subsets of S . For example, $\mathbb{P}\{1, 2, 3\} = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$.
$\mathbb{P}_1 S$	$== (\mathbb{P} S) \setminus \{\emptyset\}$ The set of all non-empty subsets of S .
$\mathbb{F} S$	Set of finite subsets of S .
$\mathbb{F}_1 S$	$== (\mathbb{F} S) \setminus \{\emptyset\}$ Set of finite non-empty subsets of S .
$\bigcap SS$	$== \{x : X \mid (\forall S : SS \bullet x \in S)\}$ Intersection of a set of sets; SS is a set containing as its members subsets of X , i.e. $SS : \mathbb{P}(\mathbb{P} X)$.
$\bigcup SS$	$== \{x : X \mid (\exists S : SS \bullet x \in S)\}$ Union of a set of sets; $SS : \mathbb{P}(\mathbb{P} X)$.
$\#S$	Size (number of members) of a finite set.

A.6 Numbers

\mathbb{Z}	The set of integers (positive, zero and negative).
\mathbb{N}	$== \{n : \mathbb{Z} \mid n \geq 0\}$ The set of natural numbers (non-negative integers).
\mathbb{N}_1	$== \mathbb{N} \setminus \{0\}$ The set of strictly positive natural numbers.

$m \dots n$	$== \{k : \mathbb{Z} \mid m \leq k \wedge k \leq n\}$ <p>The set of integers between m and n inclusive.</p>
$\min S$	<p>Minimum of a set; for $S : \mathbb{P}_1 \mathbb{Z}$</p> $\min S \in S \wedge (\forall x : S \bullet x \geq \min S).$ <p>Note, for an infinite set of numbers, such a minimum may not exist, in which case \min is not defined.</p>
$\max S$	<p>Maximum of a set; for $S : \mathbb{P}_1 \mathbb{Z}$</p> $\max S \in S \wedge (\forall x : S \bullet x \leq \max S).$ <p>Note, for an infinite set of numbers, such a maximum may not exist, in which case \max is not defined.</p>
\mathbf{R}	The set of real numbers.

A.7 Binary relations

A binary relation is modelled by a set of ordered pairs. Hence operators defined for sets can be used on relations. Let X , Y and Z be sets; $x, x_1, \dots, x_n : X$; $y, y_1, y_2, \dots, y_n : Y$; S be a subset of X ; T be a subset of Y ; and R a relation between X and Y .

$X \leftrightarrow Y$	$== \mathbb{P}(X \times Y)$ <p>The set of relations between X and Y. The set X is referred to as the <i>source</i> of the relation R and the set Y as its <i>destination</i>.</p>
$x \underline{R} y$	$== (x, y) \in R$ <p>x is related by R to y. The name of a relation may either be an identifier or an infix operator symbol. A relation with an identifier name may be used as an infix operator by underlining it. For an infix relation operator, the whole relation may be referred to by placing underscores either side of the symbol and enclosing that in parentheses. For example, the whole relation corresponding to the infix operator '$<$' is referred to by '$(- < -)$', so $(x < y) \Leftrightarrow (x, y) \in (- < -)$.</p>
$x \mapsto y$	$== (x, y)$
$\{x_1 \mapsto y_1, x_2 \mapsto y_2, \dots, x_n \mapsto y_n\}$	$== \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ <p>The relation relating x_1 to y_1, x_2 to y_2, \dots, and x_n to y_n.</p>
$\text{dom } R$	$== \{x : X \mid (\exists y : Y \bullet x \underline{R} y)\}$ <p>The domain of a relation: the set of x components that are related to some y.</p>
$\text{ran } R$	$== \{y : Y \mid (\exists x : X \bullet x \underline{R} y)\}$ <p>The range of a relation: the set of y components that some x is related to.</p>
$R_1 \circ R_2$	$== \{x : X; z : Z \mid (\exists y : Y \bullet x \underline{R_1} y \wedge y \underline{R_2} z)\}$ <p>Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$. The composition relates x to z if there is some y such that x is related to y by R_1 and y is related to z by R_2.</p>

$R_1 \circ R_2$	$== R_2 \circ R_1$ <p>Relational composition. This form is primarily used when R_1 and R_2 are functions.</p>
R^\sim	$== \{y : Y; x : X \mid x \underline{R} y\}$ <p>Transpose of a relation R. R^\sim relates y to x if and only if R relates x to y.</p>
$\text{id } S$	$== \{x : S \bullet x \mapsto x\}$ <p>Identity function on the set S.</p>
R^k	<p>The relation R composed with itself k times. This operator (sometimes called <i>iteration</i>) is only defined for homogeneous relations: relations that have the same source and destination sets. Given a homogeneous relation $R : X \leftrightarrow X$ and $k : \mathbb{N}$</p> $R^0 = \text{id } X \text{ and } R^{k+1} = R^k \circ R.$
R^+	$== \bigcup \{n : \mathbb{N}_1 \bullet R^n\}$ $= \bigcap \{Q : X \leftrightarrow X \mid R \subseteq Q \wedge Q \circ Q \subseteq Q\}$ <p>Transitive closure of relation R. A pair (x_1, x_n) is in the relation R^+ if and only if there exists a finite sequence of values x_1, x_2, \dots, x_n, where $n \geq 2$, such that $(x_1, x_2) \in R$, $(x_2, x_3) \in R$, ... and $(x_{n-1}, x_n) \in R$.</p>
R^*	$== \bigcup \{n : \mathbb{N} \bullet R^n\}$ $= R^+ \cup \text{id } X$ $= \bigcap \{Q : X \leftrightarrow X \mid \text{id } X \subseteq Q \wedge R \subseteq Q \wedge Q \circ Q \subseteq Q\}$ <p>Reflexive transitive closure.</p>
$R \downharpoonright S$	$== \{y : Y \mid (\exists x : S \bullet x \underline{R} y)\}$ <p>Image of the set S through the relation R.</p>
$S \triangleleft R$	$== \{x : X; y : Y \mid x \in S \wedge x \underline{R} y\}$ <p>Domain restriction: the relation R with its domain restricted to the set S.</p>
$S \triangleleft\!\!\triangleleft R$	$== (X \setminus S) \triangleleft R$ <p>Domain exclusion: the relation R with the members of S excluded from its domain.</p>
$R \triangleright T$	$== \{x : X; y : Y \mid x \underline{R} y \wedge y \in T\}$ <p>Range restriction to T.</p>
$R \triangleright\!\!\triangleright T$	$== R \triangleright (Y \setminus T)$ <p>Range exclusion: the relation R with the members of T excluded from its range.</p>
$R_1 \oplus R_2$	$== ((\text{dom } R_2) \triangleleft\!\!\triangleleft R_1) \cup R_2$ <p>Overriding; $R_1, R_2 : X \leftrightarrow Y$.</p>

A.8 Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let X and Y be sets, and T be a subset of X ; and f a function from X to Y .

$f\ t$ The function f applied to the expression t . A function f is a set of pairs with each member of its domain associated with a unique member of its range. $f\ t$ is only defined provided $t \in \text{dom } f$, and its value is the unique value in the range associated with the value t in its domain: $f\ t = y \Leftrightarrow (t, y) \in f$.

$X \rightarrowtail Y$ $== \{f : X \leftrightarrow Y \mid (\forall x : \text{dom } f \bullet (\exists_1 y : Y \bullet x \underline{f} y))\}$
The set of partial functions from X to Y . Note that the domain of a partial function does not necessarily contain the whole of X , but it may.

$X \rightarrow Y$ $== \{f : X \rightarrowtail Y \mid \text{dom } f = X\}$
The set of total functions from X to Y .

$X \rightarrowtail Y$ $== \{f : X \rightarrowtail Y \mid (\forall y : \text{ran } f \bullet (\exists_1 x : X \bullet x \underline{f} y))\}$
The set of partial one-to-one functions (partial injections) from X to Y .

$X \rightarrowtail Y$ $== \{f : X \rightarrowtail Y \mid \text{dom } f = X\}$
 $== (X \rightarrowtail Y) \cap (X \rightarrow Y)$
The set of total one-to-one functions (total injections) from X to Y .

$X \twoheadrightarrowtail Y$ $== \{f : X \rightarrowtail Y \mid \text{ran } f = Y\}$
The set of partial onto functions (partial surjections) from X to Y .

$X \twoheadrightarrow Y$ $== (X \twoheadrightarrowtail Y) \cap (X \rightarrow Y)$
The set of total onto functions (total surjections) from X to Y .

$X \rightarrowtail Y$ $== (X \twoheadrightarrowtail Y) \cap (X \rightarrowtail Y)$
The set of total one-to-one onto functions (total bijections) from X to Y .

$X \rightsquigarrowtail Y$ $== \{f : X \rightarrowtail Y \mid f \in \mathbb{F}(X \times Y)\}$
The set of finite partial functions from X to Y .

$X \rightsquigarrow Y$ $== (X \rightsquigarrowtail Y) \cap (X \rightarrowtail Y)$
The set of finite partial one-to-one functions from X to Y .

$(\lambda x : T \mid P \bullet t)$
 $== \{x : T \mid P \bullet x \mapsto t\}$
Lambda abstraction: the function that, given an argument x in the set T such that P holds, gives a result which is the value of the expression t .

$(\lambda x_1 : T_1; \dots; x_n : T_n \mid P \bullet t)$
 $== \{x_1 : T_1; \dots; x_n : T_n \mid P \bullet (x_1, \dots, x_n) \mapsto t\}$

disjoint $== \{S : I \leftrightarrow \mathbb{P} X \mid \forall i, j : \text{dom } S \bullet i \neq j \Rightarrow S(i) \cap S(j) = \emptyset\}$
 Pairwise disjoint, where I is a set and S an indexed family of subsets of X .

S partitions T
 $== \text{disjoint } S \wedge \bigcup \text{ran } S = T$

A.9 Orders

reflexive X $== \{R : X \leftrightarrow X \mid \forall x : X \bullet x \underline{R} x\}$
 The set of reflexive relations on X .

antisymmetric X
 $== \{R : X \leftrightarrow X \mid \forall x, y : X \bullet x \underline{R} y \wedge y \underline{R} x \Rightarrow x = y\}$
 The set of antisymmetric relations on X .

transitive X $== \{R : X \leftrightarrow X \mid \forall x, y, z : X \bullet x \underline{R} y \wedge y \underline{R} z \Rightarrow x \underline{R} z\}$
 The set of transitive relations on X .

preorder X $== \text{reflexive } X \cap \text{transitive } X$
 The set of preorders on X .

partial_order X
 $== \text{reflexive } X \cap \text{antisymmetric } X \cap \text{transitive } X$
 The set of partial orders on X .

total_order X
 $== \{R : \text{partial_order } X \mid \forall x, y : X \bullet x \underline{R} y \vee y \underline{R} x\}$
 The set of total orders on X . Note that the term *total* when applied to orders does not have the same meaning as the term *total* applied to relations. A total order is not necessarily a total relation.

A.10 Sequences

Let X be a set; A and B be sequences with elements taken from X ; and $a_1, \dots, a_n, b_1, \dots, b_n$ expressions of type X .

seq X $== \{A : \mathbb{N}_1 \leftrightarrow X \mid (\exists n : \mathbb{N} \bullet \text{dom } A = 1..n)\}$
 The set of finite sequences whose elements are drawn from X .

A The length of sequence A . (This is just ‘#’ on the set representing the sequence.)

\langle \rangle $== \{\}$
 The empty sequence.

seq₁ X $== \{s : \text{seq } X \mid s \neq \langle \rangle\}$
 The set of non-empty sequences.

$\langle a_1, \dots, a_n \rangle$	$= \{1 \mapsto a_1, \dots, n \mapsto a_n\}$
$\langle a_1, \dots, a_n \rangle \frown \langle b_1, \dots, b_m \rangle$	$= \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle$
	Concatenation.
	$\langle \rangle \frown A = A \frown \langle \rangle = A$.
<i>head</i> A	The first element of a non-empty sequence: $A \neq \langle \rangle \Rightarrow \text{head } A = A(1)$.
<i>tail</i> A	All but the head of a non-empty sequence: $\text{tail } (\langle x \rangle \frown A) = A$.
<i>last</i> A	The final element of a non-empty sequence: $A \neq \langle \rangle \Rightarrow \text{last } A = A(\#A)$.
<i>front</i> A	All but the last of a non-empty sequence: $\text{front } (A \frown \langle x \rangle) = A$.
<i>rev</i> $\langle a_1, a_2, \dots, a_n \rangle$	$= \langle a_n, \dots, a_2, a_1 \rangle$
	Reverse of a sequence.
	$\text{rev } \langle \rangle = \langle \rangle$.
\frown / AA	$= AA(1) \frown \dots \frown AA(\#AA)$
	Distributed concatenation; where $AA : \text{seq}(\text{seq}(X))$.
	$\frown / \langle \rangle = \langle \rangle$.
\circledast / AR	$= AR(1) \circledast \dots \circledast AR(\#AR)$
	Distributed relational composition; where $AR : \text{seq}(X \leftrightarrow X)$.
	$\circledast / \langle \rangle = \text{id } X$.
\oplus / AR	$= AR(1) \oplus \dots \oplus AR(\#AR)$
	Distributed override; where $AR : \text{seq}(X \leftrightarrow Y)$.
	$\oplus / \langle \rangle = \{\}$.
$A \subseteq B$	$== (\exists C : \text{seq } X \bullet A \frown C = B)$
	A is a prefix of B .
$A \text{ suffix } B$	$== (\exists C : \text{seq } X \bullet C \frown A = B)$
	A is a suffix of B .
$A \text{ in } B$	$== (\exists C, D : \text{seq } X \bullet C \frown A \frown D = B)$
	Contiguous subsequence.
<i>squash</i> (f)	Convert a finite function, $f : \mathbb{Z} \multimap X$, into a sequence by squashing its domain. That is, $\text{squash}(\{\}) = \langle \rangle$, and if $f \neq \{\}$ then $\text{squash}(f) = \langle f(i) \rangle \frown \text{squash}(\{i\} \triangleleft f)$, where $i = \min(\text{dom } f)$. For example, $\text{squash}(\{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\}) = \langle A, B, C \rangle$.

$S \upharpoonright A$	$== \text{squash}(S \triangleleft A)$ Restrict the sequence A to those items whose index is in the set S , for $S : \mathbb{P}\mathbb{N}$. The result is a sequence.
$A \upharpoonright T$	$== \text{squash}(A \triangleright T)$ Restrict the range of the sequence A to the set T , for $T : \mathbb{P}X$. The result is a sequence.

A.11 Bags

Let $B, B1, B2, \dots$ be bags with elements from the set X ; t, t_1, t_2, \dots, t_n be expressions of type X ; x, x_1, x_2, \dots, x_n be variables; and n, k_1, k_2, \dots, k_n be integers.

$\text{bag } X$	$== X \rightarrow \mathbb{N}_1$ The set of bags whose elements are drawn from the set X . Only positive frequencies are recorded.
$\{\}$	$== \{\}$ The empty bag.
$\llbracket t \mapsto n \rrbracket$	The bag which contains (only) t , n times. $= \{t \mapsto n\}$, if $n \neq 0$ $= \{\}$, if $n = 0$.
$\llbracket t_1 \mapsto k_1, t_2 \mapsto k_2, \dots, t_n \mapsto k_n \rrbracket$	$== \llbracket t_1 \mapsto k_1 \rrbracket \uplus \llbracket t_2 \mapsto k_2 \rrbracket \uplus \dots \uplus \llbracket t_n \mapsto k_n \rrbracket$ Note that with this definition we do not require the t_j to be distinct; for example, $\llbracket t \mapsto k_1, t \mapsto k_2 \rrbracket = \llbracket t \mapsto k_1 \rrbracket \uplus \llbracket t \mapsto k_2 \rrbracket$ $= \llbracket t \mapsto k_1 + k_2 \rrbracket$.
$\llbracket t_1, t_2, \dots, t_n \rrbracket$	$== \llbracket t_1 \mapsto 1, t_2 \mapsto 1, \dots, t_n \mapsto 1 \rrbracket$ The bag containing the elements t_1, t_2, \dots, t_n with the frequency in which they occur in that list.
$B \# t$	The frequency of occurrence of the value of t in the bag B : $(t \in \text{dom } B \Rightarrow B \# t = B(t))$, and $(t \notin \text{dom } B \Rightarrow B \# t = 0)$.
$\text{count}(B)(t)$	The frequency of occurrence of the value of t in the bag B : $(t \in \text{dom } B \Rightarrow \text{count}(B)(t) = B(t))$, and $(t \notin \text{dom } B \Rightarrow \text{count}(B)(t) = 0)$.
$t \text{ in } B$	$== B \# t \neq 0$ Test whether the element t occurs in the bag B with non-zero frequency.
$B1 \uplus B2$	The sum of two bags. Each element of the sum of the bags has a frequency which is the sum of its frequencies in the two bags: $(B1 \uplus B2) \# x = (B1 \# x) + (B2 \# x)$.

$B1 \bowtie B2$	The (pairwise) product of two bags. Each element of the product has a frequency which is the product of its frequencies in the two bags: $(B1 \bowtie B2) \# x = (B1 \# x) * (B2 \# x)$.
$n \otimes B$	An integer constant times a bag. Each element of the product has a frequency which is the product of its frequency in B and the constant: $(n \otimes B) \# x = n * (B \# x)$.
$B1 \sqcap B2$	The pairwise minimum of two bags. $(B1 \sqcap B2) \# x = \min\{B1 \# x, B2 \# x\}$.
$B1 \sqcup B2$	The pairwise maximum of two bags. $(B1 \sqcup B2) \# x = \max\{B1 \# x, B2 \# x\}$.
$B1 \sqsubseteq B2$	$== (\forall x : X \bullet (B1 \# x) \leq (B2 \# x))$. $B1$ is a sub-bag of $B2$. One bag is contained in another if the frequency of every element in the first bag does not exceed its corresponding frequency in the second bag.
$B1 \subset B2$	$== B1 \sqsubseteq B2 \wedge B1 \neq B2$ $B1$ is a proper sub-bag of $B2$.
setof B	$== \{x : X \mid B \# x \neq 0\}$ The set of items in the bag B that occur with non-zero frequency.
bagof S	The bag formed from the set S by including all the elements of S (and no others) with a frequency of one. $\text{dom}(\text{bagof } S) = S \wedge \text{ran}(\text{bagof } S) = \{1\}$.
bag _f X	$== \{B : \text{bag } X \mid (\text{setof } B) \in \mathbb{F} X\}$ The set of all finite bags: those bags with only a finite number of elements with non-zero frequency.
size B	The size of a finite bag is the total number of items in the bag taking into account the frequency of occurrence of each item. $\begin{aligned} \text{size}\{\} &= 0 \\ \text{size}[t] &= 1 \\ \text{size}(B1 \uplus B2) &= (\text{size } B1) + (\text{size } B2) \\ \text{size}(n \otimes B) &= n * \text{size } B \end{aligned}$
$\sum B$	The sum of all the items in the finite bag of numbers B taking into account their frequency in B . $\begin{aligned} \sum\{\} &= 0 \\ \sum[t] &= t \\ \sum(B1 \uplus B2) &= (\sum B1) + (\sum B2) \\ \sum(n \otimes B) &= n * \sum B \end{aligned}$
items(R)	The bag of items which occur in the range of the relation R . The frequency of each item is the number of domain elements that are paired with the item in R . The relation must be ‘finitary’, that is, for each

element in the range of R there are only finitely many domain elements related to it by R . Given $R : W \leftrightarrow X$

$$\begin{aligned} R \in \text{dom } \textit{items} &\Leftrightarrow \\ &(\forall x : \text{ran } R \bullet \{w : \text{dom } R \mid (w, x) \in R\} \in \mathbb{F} W) \\ R \in \text{dom } \textit{items} &\Rightarrow \\ &(\textit{items } R) \# x = \#\{w : \text{dom } R \mid (w, x) \in R\} \end{aligned}$$

As both functions and sequences can be considered as special cases of relations, *items* can be used on functions and sequences.

$\llbracket x : B \mid P \bullet t \rrbracket$

The bag of all the values of the expression t , for x ranging over all the items in the bag B such that the predicate P holds. If a value of x occurs multiple times in the bag B , then we add the corresponding value of t that many times to the resultant bag;

$$\begin{aligned} \llbracket x : B \mid P \bullet t \rrbracket \# y = \\ \sum \textit{items}(\lambda x : \text{setof } B \mid P \wedge y = t \bullet B \# x) \end{aligned}$$

A bag comprehension is only well-defined if each value of t occurs only finitely often. If the expression t is omitted, the default expression is x .

$\llbracket x_1 : B_1; x_2 : B_2; \dots; x_n : B_n \mid P \bullet t \rrbracket$

Multiple variables may be declared in a bag comprehension; each declared variable ranges over the values in the associated bag with the frequency of occurrence of the value in that bag. If the expression t is omitted, the default expression is the tuple of the variables: (x_1, x_2, \dots, x_n) .

$\llbracket D \bullet t \rrbracket$

$$== \llbracket D \mid \textit{true} \bullet t \rrbracket$$

For example, if $B : \text{bag } X$ and $C : \text{bag } Y$ then

$$\llbracket x : B; y : C \bullet (x, y) \rrbracket,$$

is of type $\text{bag}(X \times Y)$, and the pair (x, y) occurs in this bag $(B \# x) * (C \# y)$ times; this is the bag generalisation of Cartesian product.

$\biguplus BB$

The distributed bag sum of the bag of bags BB taking into account the frequency of each bag in BB as well as the frequencies of the items in the individual bags. Given $BB : \text{bag}(\text{bag } X)$

$$(\biguplus BB) \# x = \sum \llbracket B : BB \bullet B \# x \rrbracket.$$

A.12 Generalised bags

Bags can be generalised to allow both positive and negative frequencies. All the operators from the previous section can be generalised to work with bags allowing negative frequencies. The operator definitions given in the previous section have been written so that they are appropriately defined if occurrences of bag are replaced by bag. See [15] for further details and examples.

$\text{bag } X$	$== X \rightarrow (\mathbb{Z} \setminus \{0\})$ <p>The set of generalised bags whose elements are drawn from the set X. Both positive and negative frequencies are allowed in generalised bags.</p>
$-B$	<p>The negation of bag B. Each element of the negation has a frequency which is the negation of its frequencies in B:</p> $(-B) \# x = -(B \# x).$
$B1 \cup B2$	<p>The difference between two bags. Each element of the difference between the bags has a frequency which is the difference of its frequencies in the two bags:</p> $(B1 \cup B2) \# x = (B1 \# x) - (B2 \# x).$
$\text{pbagof } B$	$== \llbracket p : B \mid B \# p > 0 \rrbracket$ <p>The bag with only positive frequency items included.</p>

A.13 Free type definitions

$$X ::= \text{ident1} \mid \text{ident2} \langle\langle S \rangle\rangle$$

Free types allow a new free set X to be introduced as well as defining constructors to generate elements of the type. The constructors may either be an identifier (*ident1*), which is an element of the new type, or a constructor function (*ident2*), which is a function taking an argument of type S and returning an element of the new type. Distinct values of arguments to constructor functions return distinct elements of the free type, and distinct constructors generate distinct elements. The constructors generate all the elements of the type.

Free types are useful for defining recursive structures, such as trees. The following example defines an arithmetic expression tree:

$$\begin{aligned}
 OP &::= \text{plus} \mid \text{minus} \mid \text{times} \mid \text{divide} \\
 EXP &::= \text{const} \langle\langle \mathbb{N} \rangle\rangle \\
 &\quad \mid \text{binop} \langle\langle OP \times EXP \times EXP \rangle\rangle
 \end{aligned}$$

Type OP contains four distinct elements which may be referenced by the identifiers *plus*, *minus*, *times* and *divide*. The type EXP describes an expression tree which is either a constant, natural number, or an expression consisting of a binary operator and two sub-expression trees.

Appendix B

Glossary: Z schema notation

B.1 Schema definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

S	
$x : \mathbb{Z}$	
$y : \mathbb{P}\mathbb{Z}$	
$x \leq \#y$	

and horizontally, for the same example,

$$S \triangleq [x : \mathbb{Z}; y : \mathbb{P}\mathbb{Z} \mid x \leq \#y]$$

As well as explicit declarations of variables, we allow schemas to be used in declarations as a shorthand for the declaration of the the variables in the schema constrained by the predicate of the schema. For example, schemas may be used in the declaration part of \forall , λ , $\{\dots\}$, etc.:

$$(\forall S \bullet y \neq \{\}) \Leftrightarrow (\forall x : \mathbb{Z}; y : \mathbb{P}\mathbb{Z} \mid x \leq \#y \bullet y \neq \{\})$$

$\{S\}$ Stands for the set of objects described by schema S . In declarations we usually write $w : S$ as an abbreviation for $w : \{S\}$, e.g. $w : S$ declares a variable w with components x (an integer) and y (a set of integers) such that $x \leq \#y$.

B.2 Schema operators

Let S be defined as above and $w : S$.

$w.x$ $== (\lambda S \bullet x)(w)$
 Projection functions: the component names of a schema may be used as projection (or selector) functions, e.g. $w.x$ is w 's x component and $w.y$ is its y component; of course, the predicate ' $w.x \leq \#w.y$ ' holds.

θS The (unordered) tuple formed from a schema's variables, e.g. θS contains the named components x and y . ' θ ' is commonly used to equate the before and after state components of an operation, e.g. $\theta S' = \theta S$ is equivalent to $x' = x \wedge y' = y$. For a usage of θS to be well-defined, the components of S must be defined in the enclosing scope.

$\text{pred } S$ The predicate part of a schema, e.g. $\text{pred } S$ is ' $x \leq \#y$ '.

Compatibility

Two schemas are compatible if the declared sets of each variable common to the declaration parts of both the schemas are equal. In addition, any global variables referenced in predicate part of one of the schemas must not have the same name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

Inclusion A schema S may be included within the declarations of a schema R , in which case the declarations of S are merged with the other declarations of R (variables declared in both S and R must be compatible) and the predicates of S and R are conjoined. For example,

R	
S	
$z : \mathbb{Z}$	
$z < x$	

is equivalent to

R	
$x, z : \mathbb{Z}$	
$y : \mathbb{P}\mathbb{Z}$	
$x \leq \#y \wedge z < x$	

The included schema (S) may not refer to global variables that have the same name as one of the declared variables of the including schema (R).

$S[\text{new/old}]$ or $S\left[\frac{\text{new}}{\text{old}}\right]$

Renaming of components: the schema S in which the component *old* has been renamed to *new* both in the declaration and every free occurrence in the predicate. For example,

$S[z/x]$ is $[z : \mathbb{Z}; y : \mathbb{P}\mathbb{Z} \mid z \leq \#y]$, and

$S\left[\frac{y,x}{x,y}\right]$ is $[y : \mathbb{Z}; x : \mathbb{P}\mathbb{Z} \mid y \leq \#x]$.

In the second case above, the renaming is simultaneous. As usual, the renaming in the predicate might entail consequential changes of bound variables. If the renaming leads to two or more previously distinct variables ending up with the same name, the renaming is valid only if the variables all have the same type; the set of variables is replaced by a single variable with the new name.

Decoration Systematic renaming of the variables declared in a schema. Decoration with subscript, superscript, prime, etc. For example, S' is $[x' : \mathbb{Z}; y' : \mathbb{P}\mathbb{Z} \mid x' \leq \#y']$. Multiple decorations of a single schema are allowed, e.g. S'' .

$\neg S$ The schema S with its predicate part negated. For example, $\neg S$ is $[x : \mathbb{Z}; y : \mathbb{P}\mathbb{Z} \mid \neg(x \leq \#y)]$.

$S \wedge T$ The schema formed from schemas S and T by merging their declarations and conjoining (and-ing) their predicates. The two schemas must be compatible (see above). Given

T	
$x : \mathbb{Z}$	
$z : \mathbb{P}\mathbb{Z}$	
$x \in z$	

$S \wedge T$ is

$S \wedge T$	
$x : \mathbb{Z}$	
$y : \mathbb{P}\mathbb{Z}$	
$z : \mathbb{P}\mathbb{Z}$	
$x \leq \#y \wedge x \in z$	

$S \vee T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \vee T$ is

$S \vee T$	
$x : \mathbb{Z}$	
$y : \mathbb{P}\mathbb{Z}$	
$z : \mathbb{P}\mathbb{Z}$	
$x \leq \#y \vee x \in z$	

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking ‘pred $S \Rightarrow$ pred T ’ as the predicate. The two schemas must be compatible (see above). For example, $S \Rightarrow T$ is

$S \Rightarrow T$	
$x : \mathbb{Z}$	
$y : \mathbb{P}\mathbb{Z}$	
$z : \mathbb{P}\mathbb{Z}$	
$x \leq \#y \Rightarrow x \in z$	

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking ‘pred $S \Leftrightarrow$ pred T ’ as the predicate. The two schemas must be compatible (see above). For example, $S \Leftrightarrow T$ is

$S \Leftrightarrow T$	_____
$x : \mathbb{Z}$	
$y : \mathbb{P}\mathbb{Z}$	
$z : \mathbb{P}\mathbb{Z}$	

$x \leq \#y \Leftrightarrow x \in z$	

$S \setminus (v_1, v_2, \dots, v_n)$

Hiding: the schema S with variables v_1, v_2, \dots, v_n hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. For example, $S \setminus (x)$ is

$S \setminus (x)$	_____
$y : \mathbb{P}\mathbb{Z}$	

$(\exists x : \mathbb{Z} \bullet x \leq \#y)$	

$S \upharpoonright (v_1, v_2, \dots, v_n)$

Projection: The schema S with any variables that do not occur in the list v_1, v_2, \dots, v_n hidden – the variables are removed from the declarations and are existentially qualified in the predicate. For example, $(S \wedge T) \upharpoonright (x, y)$ is

$(S \wedge T) \upharpoonright (x, y)$	_____
$x : \mathbb{Z}$	
$y : \mathbb{P}\mathbb{Z}$	

$(\exists z : \mathbb{P}\mathbb{Z} \bullet x \leq \#y \wedge x \in z)$	

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

$\exists D \bullet S$

Existential quantification of a schema. The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S . The predicate of S is existentially quantified over D . For example, $(\exists x : \mathbb{Z} \bullet S)$ is the following schema:

$\exists x : \mathbb{Z} \bullet S$	_____
$y : \mathbb{P}\mathbb{Z}$	

$\exists x : \mathbb{Z} \bullet x \leq \#y$	

The declarations may include schemas. For example,

$\exists S \bullet T$	_____
$z : \mathbb{P}\mathbb{Z}$	

$\exists S \bullet x \in z$	

This is equivalent to

$\exists S \bullet T$	_____
$z : \mathbb{P}\mathbb{Z}$	_____
$\exists x : \mathbb{Z}; y : \mathbb{P}\mathbb{Z} \mid x \leq \#y \bullet x \in z$	_____

$\forall D \bullet S$ Universal quantification of a schema. The variables declared in the schema S that also appear in the declarations D are removed from the declarations of S . The predicate of S is universally quantified over D . For example, $(\forall x : \mathbb{Z} \bullet S)$ is the following schema.

$\forall x : \mathbb{Z} \bullet S$	_____
$y : \mathbb{P}\mathbb{Z}$	_____
$\forall x : \mathbb{Z} \bullet x \leq \#y$	_____

The declarations may include schemas. For example,

$\forall S \bullet T$	_____
$z : \mathbb{P}\mathbb{Z}$	_____
$\forall S \bullet x \in z$	_____

This is equivalent to

$\forall S \bullet T$	_____
$z : \mathbb{P}\mathbb{Z}$	_____
$\forall x : \mathbb{Z}; y : \mathbb{P}\mathbb{Z} \mid x \leq \#y \bullet x \in z$	_____

B.3 Operation schemas

The following conventions are used for variable names in those schemas that represent operations, i.e. are written as descriptions of operations on some state:

- undashed** – state before the operation;
- dashed** – state after the operation;
- ending in ‘?’** – inputs to (arguments for) the operation; and
- ending in ‘!’** – outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

- $\Delta S \quad \triangleq S \wedge S'$
Change of state schema: this is a default definition for ΔS . In some specifications it is useful to have additional constraints on the change of state schema. In these cases ΔS can be explicitly defined.
- $\Xi S \quad \triangleq [\Delta S \mid \theta S' = \theta S]$
No change of state schema.

B.4 Operation schema operators

pre S Precondition: the after-state components (dashed) and the outputs (ending in '!') are hidden, e.g. given,

$$\frac{S}{\begin{array}{c} x?, s, s', y! : \mathbb{N} \\ \hline s' = s - x? \wedge y! = s' \end{array}}$$

pre S is

$$\frac{\text{pre } S}{\begin{array}{c} x?, s : \mathbb{N} \\ \hline \exists s', y! : \mathbb{N} \bullet \\ \quad s' = s - x? \wedge y! = s' \end{array}}$$

Because, given the declarations above,

$$(\exists s', y! : \mathbb{N} \bullet s' = s - x? \wedge y! = s') \Leftrightarrow s \geq x?$$

the predicate can be simplified.

$$\frac{\text{pre } S}{\begin{array}{c} x?, s : \mathbb{N} \\ \hline s \geq x? \end{array}}$$

$$S \oplus T \quad \hat{=} (S \wedge \neg \text{pre } T) \vee T$$

Overriding: for example, given S above and T ,

$$\frac{T}{\begin{array}{c} x?, s, s' : \mathbb{N} \\ \hline s \leq x? \wedge s' = s + x? \end{array}}$$

$S \oplus T$ is

$$\frac{S \oplus T}{\begin{array}{c} x?, s, s', y! : \mathbb{N} \\ \hline s' = s - x? \wedge y! = s' \wedge \\ \quad \neg (\exists s' : \mathbb{N} \bullet s \leq x? \wedge s' = s + x?) \\ \vee \\ s \leq x? \wedge s' = s + x? \end{array}}$$

Because, given the declarations above,

$$(\exists s' : \mathbb{N} \bullet s \leq x? \wedge s' = s + x?) \Leftrightarrow s \leq x?$$

the predicate can be simplified.

$$\begin{array}{c}
\frac{S \oplus T}{x?, s, s', y! : \mathbb{N}} \\
\hline
\begin{array}{c}
s > x? \wedge s' = s - x? \wedge y! = s' \\
\vee \\
s \leq x? \wedge s' = s + x?
\end{array}
\end{array}$$

$S \circ T$

Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation that relates the initial state of S to the final state of T through the intermediate state. To form the composition of S and T we take the pairs of after-state components of S and before-state components of T that have the same basename, rename each pair to a new variable, take the conjunction of the resulting schemas, and hide the new variables. For example, $S \circ T$ is

$$\begin{array}{c}
\frac{S \circ T}{x?, s, s', y! : \mathbb{N}} \\
\hline
(\exists ss : \mathbb{N} \bullet \\
\quad ss = s - x? \wedge y! = ss \\
\quad \wedge ss \leq x? \wedge s' = ss + x?)
\end{array}$$

which simplifies to

$$\begin{array}{c}
\frac{S \circ T}{x?, s, s', y! : \mathbb{N}} \\
\hline
y! = s - x? \wedge s - x? \leq x? \wedge s' = s
\end{array}$$

Note that the composition of two schemas using the specifier's semicolon, $S \circ T$, is not the same as sequential composition in programming languages using the programmer's semicolon. The difference arises if the result produced by the S is not an acceptable input to T , i.e. it does not satisfy $\text{pre } T$. In a programming language this would abort, but with the specifier's semicolon this would only rule out the intermediate state; if there are intermediate states that are acceptable to T then one of them must be used.

$S \gg U$

Piping: this schema operation is similar to schema composition; the difference is that, rather than identifying the after-state components of S with the before-state components of U , the output components of S (ending in '!') are identified with the input components of U (ending in '?') that have the same basename. For example, if U is the schema

$$\begin{array}{c}
\frac{U}{y?, u, u', z! : \mathbb{N}} \\
\hline
y? \neq 0 \wedge u' = u + y? \wedge z! = u'
\end{array}$$

then $S \gg U$ is the following schema:

$$\frac{S \gg U}{\begin{array}{c} x?, s, u, s', u', z! : \mathbb{N} \\ (\exists yy : \mathbb{N} \bullet \\ \quad s' = s - x? \wedge yy = s' \wedge \\ \quad yy \neq 0 \wedge u' = u + yy \wedge z! = u') \end{array}}$$

This simplifies to the following schema:

$$\frac{S \gg U}{\begin{array}{c} x?, s, u, s', u', z! : \mathbb{N} \\ s' = s - x? \wedge \\ s' \neq 0 \wedge u' = u + s' \wedge z! = u' \end{array}}$$

In the above example the state components of the two schemas are disjoint. If the state components are not disjoint then the effect on the state components is that of schema conjunction.

Bibliography

- [1] J.-R. Abrial. Formal programming. Private manuscript, Paris, 1982. 42
- [2] R. Biekert and B. Janssen. The implementation of a file system for the open distributed operating system Amoeba. Informatica rapport, Vrije Universiteit, Amsterdam, 1983. 149
- [3] D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors. *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*. VDM-Europe, Springer-Verlag, Berlin, 1990. xvi, 245
- [4] B.W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981. 190
- [5] Jonathan P. Bowen, editor. *Proc. Z Users Meeting*, Oxford, UK, December 1987. Oxford University Computing Laboratory. xvi
- [6] Jonathan P. Bowen, editor. *Proc. Third Annual Z Users Meeting*, Oxford, UK, December 1988. Oxford University Computing Laboratory. xvi
- [7] S.M. Brien. Z Base Standard – Version 0.5. Oxford University Computing Laboratory ZIP/PRG/92/92, March 1992. 191
- [8] B.P. Collins, J.E. Nicholls, and I.H. Sørensen. Introducing formal methods: the CICS experience with Z. Technical Report TR12.260, IBM Hursley Park, December 1987. 183, 192
- [9] S. Croxall, P.J. Lupton, and J.B. Wordsworth. A formal specification of the CPI Communications. Technical Report TR12.277, IBM Hursley Park, December 1990. 194
- [10] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8): 453–457, August 1975. 186
- [11] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976. 186
- [12] R. B. Gimson. A file package – user manual. Distributed Computing Project working paper, Programming Research Group, Oxford University, 1983. 149
- [13] G. Goos and J. Hartmanis, editors. *VDM – The Way Ahead. Proc. 2nd VDM-Europe Symposium*, volume 328 of *Lecture Notes in Computer Science*. VDM-Europe, Springer-Verlag, Berlin, 1988. xvi

- [14] D. Gries. *The Science of Programming*. Springer-Verlag, Berlin, 1981. 186
- [15] I. J. Hayes. A generalisation of bags in Z. In J. E. Nicholls, editor, *Z User Workshop: Proceedings of the Fourth Annual Z User Meeting, Oxford, December 1989*, Workshops in Computing, pages 113–127. Springer, 1990. 233
- [16] I. J. Hayes and L. P. Wildman. Towards libraries for Z. In *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting, London, December, 1992*. 74
- [17] I.J. Hayes. Applying formal specification to software development in industry. *IEEE Trans. Software Engng*, SE-11(2): 169–178, 1985. Also Chapter 13 of this volume. 183, 198, 199
- [18] I.J. Hayes, He Jifeng, C.A.R. Hoare, C.C. Morgan, J.W. Sanders, I.H. Sørensen, J.M. Spivey, and B. Sufrin. Data refinement refined. Oxford University Computing Laboratory, May 1985. 191
- [19] C. A. R. Hoare. Professionalism. *BCS Comput. Bull.*, 2(29): 2–4, September 1981. Invited talk given at BCS '81. 163
- [20] C. A. R. Hoare. Programming is an engineering profession. Technical Monograph 27, Programming Research Group, Oxford University, 1982. Also published in [22]. 42, 184
- [21] C. A. R. Hoare. Specifications, programs and implementations. Technical Monograph PRG-29, Programming Research Group, Oxford University, 1982. 99
- [22] C. A. R. Hoare. Programming is an engineering profession. In C. B. Jones, editor, *Essays in Computing Science*, pages 315–324. Prentice Hall, Hemel Hempstead, 1989. 244
- [23] Ib Holm Sørensen. Specification of a simple assembler. CICS Project working paper, Programming Research Group, Oxford University, 1982. 136
- [24] I.S.C. Houston and J.B. Wordsworth. A Z specification of part of the CICS file control API. Technical Report TR12.272, IBM Hursley Park, February 1990. 194
- [25] IBM Corporation. *CICS/ESA Application Programmer's Reference*. SC33-0676. 196, 197
- [26] IBM Corporation. *CICS/ESA Application Programming Guide*. SC33-0675. 197
- [27] IBM Corporation. *CICS/ESA General Information*. GC33-0155. 183
- [28] IBM Corporation. *OS PL/I Checkout and Optimising Compilers: Language reference manual*, 1976. 169
- [29] IBM Corporation. *CICS/OS/VS Version 1 Release 5, Application Programmer's Reference Manual (Command level)*, 1980. 168, 179
- [30] IBM Corporation. *CICS/VS General Information*, 1980. 166
- [31] ICL. *Reference Manual for the ICL Data Dictionary System (DDS.600)*, May 1982. ICL Document RPO120. 100

- [32] He Jifeng, C.A.R. Hoare, and J.W. Sanders. Data refinement refined: resumé. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming ESOP86*, volume 213 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1986. 191
- [33] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall, Hemel Hempstead, 1980. 42, 99
- [34] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Hemel Hempstead, 2nd edition, 1990. xvi
- [35] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall, Hemel Hempstead, 1990. xvi
- [36] S. King, I.H. Sørensen, and J.C.P. Woodcock. Z : Grammar and concrete and abstract syntaxes. Technical Report PRG-68, Programming Research Group, Oxford, 1988. 191
- [37] Steve King. Z and the refinement calculus. In Bjørner et al. [3], pages 164–188. xvi
- [38] J. Lions. UNIX operating system source code level 6. Technical report, Department of Computer Science, University of NSW, Sydney, Australia, 1977. 42
- [39] P.J. Lupton. Promoting forward simulation. In Nicholls [48]. 198
- [40] C.C. Morgan. Mailbox communication in Pascal-M. Distributed Computing Project working paper, Programming Research Group, Oxford University, 1982. 67
- [41] C.C. Morgan. Specification of the Cambridge model distributed system name service. Distributed Computing Project working paper, Programming Research Group, Oxford University, 1982. 67
- [42] C.C. Morgan. Specification of a communication system. In Y. Paker and J.-P. Verjus, editors, *Distributed Computing Systems: Synchronisation, Control, and Communication*. Academic Press, 1983. 29
- [43] C.C. Morgan. Using mathematics in user manuals. Distributed Computing Project technical report, Programming Research Group, Oxford University, 1983. 179
- [44] C.C. Morgan. Schemas in Z: A preliminary reference manual. Distributed Computing Project report, Programming Research Group. Oxford University, March 1984. This is a historical reference; for a more up-to-date treatment of Z schemas see [57]. 13
- [45] C.C. Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, 1990. xvi
- [46] C.C. Morgan and B.A. Sufrin. Specification of the UNIX file system. *IEEE Trans. Software Engng*, SE-10(2): 128–142, March 1984. Also Chapter 4 of this volume. 99

- [47] R. Needham and A. Herbert. The Cambridge file service. In *The Cambridge Distributed Computing System*, pages 41–63. Addison-Wesley, Reading, Mass., 1982. 149
- [48] J.E. Nicholls, editor. *Z User Workshop, Oxford 1989*, Workshops in Computing. Springer-Verlag, Berlin, 1990. xvi, 245, 247
- [49] J.E. Nicholls, editor. *Z User Workshop, Oxford 1990*, Workshops in Computing. Springer-Verlag, Berlin, 1991. xvi
- [50] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall, Hemel Hempstead, 1991. xv
- [51] R.A. Radice, N.K. Roth, A.C. O'Hara, and W.A. Ciarfella. A programming process architecture. *IBM Sys. J.*, 24(2/3): 79–90, 1985. 185
- [52] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7), July 1974. 41, 42
- [53] J. M. Spivey. *The fUZZ Manual*. Computing Science Consultancy, Garsington, Oxford, 1988. xv
- [54] J.M. Spivey. Understanding Z: a specification language and its formal semantics. DPhil thesis, Oxford University Computing Laboratory, Oxford, UK, October 1985. Subsequently published as [55]. xv, 191
- [55] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Number 3 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1988. xv, 246
- [56] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Hemel Hempstead, 1989. xv
- [57] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Hemel Hempstead, 2nd edition, 1992. xv, 13, 191, 245
- [58] J. Staunstrup, editor. *Program Specification: Proceedings of a Workshop, Aarhus, Denmark (August 1981)*, volume 134 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1982. 167
- [59] J. E. Stoy and C. Strachey. OS6 – an operating system for a small computer. *Comput. J.*, 15(2): 195–203, 1972. 147
- [60] B. A. Sufrin. Formal specification of a display-oriented text editor. *Sci. Comput. Program.*, 1: 157–202, May 1982. 67
- [61] B. A. Sufrin. Mathematics for system specification. Lecture notes, Programming Research Group, Oxford University, 1983–84. 99
- [62] B. A. Sufrin. Formal specification of an electronic mail system. In *Formal Methods and the Design of Effective User Interfaces, Proceedings of HCI'86*. Cambridge University Press, Cambridge, 1986. 67
- [63] J.C.P. Woodcock. Luptonian triads explained. Oxford University Computing Laboratory, 1989. 200

- [64] J.C.P. Woodcock. Mathematics as a management tool: proof rules for promotion. In B.A. Kitchenham, editor, *Software Engineering for Large Software Systems*. Elsevier, Amsterdam, 1990. 198
- [65] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics*. Pitman, London, 1988. xv, 3
- [66] J.B. Wordsworth. Practical experience of formal specification: a programming interface for communications. In C. Ghezzi and J. McDermid, editors, *ESEC89: 2nd European Software Engineering Conference*, volume 387 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1989. 194, 200
- [67] J.B. Wordsworth. The CICS application programming interface definition. In Nicholls [48]. 183, 192
- [68] B.M. Yelavich. Customer information control system – an evolving facility. *IBM Sys. J.*, 24(2/3), 1985. 183

Index

This index includes all schemas, explicit components of schemas, basic types, axiomatic and generic definitions, disjoint unions and members of disjoint unions used in the case studies. Such entries are set in *italic* type. The entries for components of schemas and members of disjoint unions give the name of the schema or disjoint union, respectively, in which they are defined.

- a!*
 - in *ExceptionCheck* 173
- a?*
 - in *FileAdd* 12
 - in *HandleCondition* 171
- A* 126, 130
- abort*
 - in *ACTION* 170
- ACTION* 170
- Add* 12, 15, 88
- Add_Employee* 124
- Add_Visitor_to_Meeting* 95
- Add_Visitor_to_Meeting_0* 86
- admin*
 - in *DD13* 118
- after 44
- amount?*
 - in Δ *Dining_State* 91
- Append* 207
- Append0* 204
- AppendQ* 209
- AppendQN0* 210
- AppendQN1* 211
- AppendQN2* 211
- ar*
 - in *TSQ* 203
- AS* 52
- Δ *AS* 53
- Φ *AS* 53
- Ascending* 12
- ASSEMBLY* 130, 136
- auth*
 - in *DD1* 102
 - in *DD9* 107
 - in *DD13* 118
- AuthoritarianDD* 103
- authority* 110
- AUTHORITY* 113
- AV* 33, 35
- Δ *AV* 35, 36
- avail*
 - in *AV* 33, 35
 - in *TN* 34, 36
- BAdd* 24
- BAdd0* 22
- BAdd1* 22
- bag 10
 - addition 11
 - items 10
- Balance*
 - in *Response* 122
- basic type 4
- BDelete* 24, 26, 27
- BDelete0* 22
- BDelete1* 22
- BEnd* 24
- BEnd0* 21
- BLocate* 27
- BLocate0* 27
- BlockId* 25
- BLookUp* 24
- BLookUp0* 22
- BLookUp1* 22
- Book_Conf_Room* 95
- Book_Conf_Room_0* 89
- Book_Dining_Room* 95
- Book_Dining_Room_0* 91
- Book_Hotel_Room* 96
- Book_Hotel_Room_0* 84
- Book_Transport* 96
- Book_Transport_0* 84
- Break* 33, 35, 37
- BReplace* 24
- BReplace0* 21, 22
- BReplace1* 22, 23

- BSearch* 24
- BSearch0* 20
- bst*
 - in *BDelete* 26, 27
 - in *BST1* 25
 - in *BST2* 25
 - in Δ *BST* 20
- bst'*
 - in *BDelete* 26, 27
 - in Δ *BST* 20
- bst₀* 20
- BST* 19
- Δ *BST* 20
- Φ *BST* 22
- Ξ *BST* 20
- BST1* 25
- BST2* 25
- BStart* 24
- BStart0* 21
- BYTE* 203, 43
- c?*
 - in *ExceptionCheck* 173
 - in *HandleCondition* 171
 - in *IgnoreCondition* 172
- Call* 31, 35, 36, 37
- canaccess*
 - in *AuthoritarianDD* 103
 - in *NotQuiteSoAuthoritarianDD* 103
- Cancel_Conf_Rooms* 95
- Cancel_Conf_Rooms_0* 89
- Cancel_Conf_Rooms_1* 89
- Cancel_Dining_Room* 95
- Cancel_Dining_Room_0* 91
- Cancel_Dining_Room_1* 91
- Cancel_Hotel_Room* 96
- Cancel_Hotel_Room_0* 84
- Cancel_Meeting* 95
- Cancel_Meeting_0* 87
- Cancel_Meeting_1* 94
- Cancel_Meeting_Arrangements_0* 95
- Cancel_Meeting_Arrangements_1* 96
- Cancel_Meeting_Fail* 94
- Cancel_Transport* 96
- Cancel_Transport_0* 84
- canretrieve*
 - in *DD6* 104
 - in *DD9* 107
 - in *DD15* 119
- canupdate*
 - in *DD6* 104
 - in *DD9* 107
 - in *DD15* 119
- Capacity* 160
- CAVIAR* 93
- Δ *CAVIAR* 94
- CAVIAR_Init* 93
- CHAN* 50, 69
- Δ *CHAN* 51
- ChanErr* 64
- cid!*
 - in *open* 66
 - in *openCS* 52
- cid?*
 - in Φ *AS* 53
 - in *CidErr* 64
 - in *closeCS* 52
 - in *fstat* 55
 - in *read* 65
 - in *readAS* 54
 - in *seekAS* 54
 - in *writeAS* 54
- CID* 52
- CidErr* 64
- client?*
 - in Δ *RS* 156
- Client* 142
- ClockIn* 124
- ClockIn_0* 123
- Δ *Clocking* 122
- ClockOut* 124
- ClockOut_0* 123
- closeCS* 52
- closeFS* 59
- CON* 30
- CONDITION* 170, 207
- Conference_Init* 89
- Conference_State* 89
- Δ *Conference_State* 89
- cons*
 - in *efficientTN* 31
 - in *TN* 30, 34, 36
- console* 216
- contained*
 - in *BST1* 25
- contents*
 - in *FILE* 142
- contents!*
 - in *lsNS* 56
 - in *ReadFile* 144
 - in *UpdateData* 146
- contents?*
 - in *ReadData* 145
 - in *StoreFile* 143
 - in *UpdateData* 146
- core*
 - in *IS* 133
- cost!*

- in *DeleteFile* 144
 - in ΔRS 156
 - in *StoreFile* 143
- cr!*
 - in *Worked* 123
- CR* 73
- Create* 85
- created*
 - in *FILE* 142
- createFS* 58
- createFS0* 58
- Create_Meeting* 94
- Create_Meeting_0* 87
- createNS* 55
- CreateQ* 209
- createSS* 47
- Create_Visitor* 96
- Create_Visitor_0* 92
- CS* 52
- ΔCS 52
- cstore*
 - in *CS* 52
- d?*
 - in *Delete_Visitor_0* 97
 - in *FileUpdate* 8
- data!*
 - in *read* 65
 - in *readAS* 54
 - in *readFILE* 45
- data?*
 - in *writeAS* 54
 - in *writeFILE* 46
- Date* 73
- date_of_session* 73
- date_of_time* 73
- DD0* 102
- DD1* 102
- DD2* 103
- DD3* 103
- DD4* 104
- DD5* 104
- DD6* 104
- DD7* 105
- DD8* 106
- DD9* 107
- DD10* 111
- DD11* 112
- DD12* 113
- DD13* 118
- DD14* 118
- DD15* 119
- DDS* 108
- $\square DDS$ 108
- ΔDDS 108
- $\Delta DDS1$ 113
- Default* 171
- delegates*
 - in *DD3* 103
 - in *DD9* 107
 - in *DD14* 118
- Delete* 16, 6
- DeleteElement* 115
- DeleteFile* 144
- DeleteProperties* 115
- DeleteQ* 209
- DeleteQ0* 209
- Delete_Visitor* 97
- Delete_Visitor_0* 97
- Δ (Delta) 14
- Destroy* 85
- destroyFS* 59, 61
- destroyNS* 56
- destroySS* 48
- Destroy_Visitor* 96
- Destroy_Visitor_0* 92
- dev*
 - in *MapName* 216
- DeviceCapacity* 68
- dialled?*
 - in *Call* 31, 35, 37
- Diary* 87
- $\Delta Diary$ 88
- Diary_Init* 88
- Dining_Init* 90
- Dining_State* 90
- $\Delta Dining_State$ 91
- dir?*
 - in *lsNS* 56
- direncode* 58
- dirformat* 57
- dirstored* 57
- disjoint* 30
- Display* 109
- dnames*
 - in *NS* 56
- domain (dom) 4
- domain exclusion (\Leftarrow) 7
- DR* 73
- E* 101
- efficientTN* 30, 31, 34, 36
- EI* 112
- eid?*
 - in *DeleteElement* 115
 - in *FOR* 110
 - in *InsertNewElement0* 114
- Element* 110

- elementcontext*
 - in *DDS* 108
- elements*
 - in *DD0* 102
 - in *DD1* 102
 - in *DD9* 107
 - in *DD13* 118
- Elements* 110
- Empty* 23
- engaged!*
 - in *Engaged* 32, 35, 37
- Engaged* 32, 35, 36, 37
- Erase* 88
- error*
 - in *CONDITION* 170
- ERROR* 207
- ExceptionCheck* 173
- Exceptions* 170
- Δ Exceptions* 171
- exists*
 - in *Pool* 85
- exists_D*
 - in *Meeting_State* 86
- expires*
 - in *FILE* 142
 - in *RS* 156
- expires!*
 - in *ReadFile* 144
 - in *ReadStoredFile* 148
- expires?*
 - in *StoreFile* 143
- f* 117
 - in *FileAdd* 12
 - in *FileUpdate* 8
- f'*
 - in *FileAdd* 12
 - in *FileUpdate* 8
- fd*
 - in *CHAN* 50, 69
 - in *createNS* 55
 - in *createSS* 47
 - in *destroySS* 48
 - in *open* 66
 - in *Φ SS* 49
- fid'*
 - in *lookupNS* 55, 63
 - in *open* 66
- FID* 47
- file*
 - in *read* 65
 - in *readAS* 54
 - in *readFILE* 45
 - in *Φ SS* 49
 - in *writeAS* 54
 - in *writeFILE* 46
- file'*
 - in *readFILE* 45
 - in *Φ SS* 49
 - in *writeAS* 54
 - in *writeFILE* 46
- FILE* 142, 43, 67
- FileAdd* 12
- FileNumberLimit* 67
- files*
 - in *Δ FS* 143
 - in *FS* 143
- files'*
 - in *Δ FS* 143
- FileSizeLimit* 67
- FileUpdate* 8
- Fix* 33, 35, 37
- Flexi* 122
- Δ Flexi* 122
- Ξ Flexi* 122
- Flexitime_Hours*
 - in *Flexi* 122
- FOR* 110
- Forest* 117
- framing* 21
- from?*
 - in *Append0* 204
 - in *Standard_Append* 205
 - in *Write0* 204
- FS* 143, 57
- Δ FS* 143, 63
- fstat* 55
- fstore*
 - in *SS* 47, 67, 68
- function*
 - application 4
 - empty 4
 - override (\oplus) 5
 - partial 4, 7
- GetNickname* 151
- GuestClient* 152
- handle?*
 - in *ExceptionCheck* 173
- HandleCondition* 171
- Handler*
 - in *Exceptions* 170
- HangUp* 32, 35, 36, 37
- has*
 - in *DD10* 111
- Hotel_Init* 84
- Hotel_State* 83, 84

- HR* 73
- i?*
 - in *Add* 88
- I* 112
- ident!*
 - in *Add_Employee* 124
- ident?*
 - in *ΔClocking* 122
 - in *Unknown* 124
- Ident* 122
- IdUnknown*
 - in *Response* 122
- IgnoreCondition* 172
- Implementation* 134, 135
- in* 127
 - in *ASSEMBLY* 130, 136
 - in *Flexi* 122
 - in *Implementation* 135
 - in *Phase1* 133
 - in *PreASSEMBLY* 131
- in?*
 - in *Sort* 11
 - in *SortNoDup* 12
- In*
 - in *Response* 122
- in1?*
 - in *Merge* 12
- in2?*
 - in *Merge* 12
- ind!*
 - in *ΔClocking* 122
 - in *Unknown* 124
- info*
 - in *Diary* 87
- info_1*
 - in *Diary* 87
- inhibit* 112
- Initial* 171
- inputs* 5
- InsertKey* 124
- InsertNewElement* 114
- InsertNewElement0* 114
- InsertNewProperties* 115
- InsertNewProperties0* 114
- interval?*
 - in *NotAvailable* 160
 - in *Reserve* 158
- Interval* 157
- into!*
 - in *Read0* 205
 - in *Remove0* 204
 - in *Standard_Remove* 205
- in_use*
 - in *Hotel_State* 84
 - in *R_U* 76
- in_use_D*
 - in *Conference_State* 89
- IS* 133
- item!*
 - in *Append0* 204
- item?*
 - in *OutOfBounds* 207
 - in *Read0* 205
 - in *Write0* 204
- ItemErr*
 - in *CONDITION* 207
- items* 10
- K* 112
- Key* 7
- lab* 127, 130
- length?*
 - in *read* 65
 - in *readAS* 54
 - in *readFILE* 45
- level!*
 - in *BLocate0* 27
- Level* 105
- LI* 73
- linkFS* 63
- linkNS* 62
- ln?*
 - in *MapName* 216
 - in *ToSetL* 217
- LName* 215
- lns!*
 - in *ToSetL* 217
- lns?*
 - in *MapSet* 216
- LookUp* 15, 6
- lookupNS* 55, 63
- LReceive* 216
- LSend* 216
- LSendM* 216
- LSendReceive* 216
- lsNS* 56
- ltop*
 - in *LtoP* 216
- LtoP* 216
- \exists *LtoP* 216
- m!*
 - in *NReceive_0* 215
- m?*
 - in *NSend_0* 214
 - in *NSendM_0* 214

- in *Pre* 217
- M* 126, 130, 73
- MapName* 216
- MapName_PreIn* 217
- mapping 4
- MapSet* 216
- master*
 - in *DD8* 106
 - in *DD9* 107
- maxbytes* 68
- max_cap* 90
- maynotupdate*
 - in *DD5* 104
 - in *DD9* 107
- mayretrieve*
 - in *DD4* 104
 - in *DD9* 107
 - in *DD14* 118
- mayupdate*
 - in *DD14* 118
- meeting?*
 - in *Cancel_Meeting_Arrangements_0* 95
- Meeting_Init* 86
- Meeting_State* 86
- Δ *Meeting_State* 86
- Merge* 12
- Message* 213
- minbytes* 68
- mnem* 127, 130
- module 74
 - decoration 92
 - instantiation 74, 79
 - parameters 74
- Money* 142
- M_OP* 94
- moveFS* 63
- moveNS* 62
- M_SYS* 93
- multiset 10
- M_V::r?*
 - in *Cancel_Meeting_Fail* 94
- M_V::t?*
 - in *Cancel_Meeting_Fail* 94
- n!*
 - in *MapName* 216
- n?*
 - in *NReceive_0* 215
 - in *NSend_0* 214
 - in *PreIn* 217
 - in *seekoffset* 69
 - in *ToSet* 214
- n24bit* 69
- name!*
 - in *StoreFile* 143
- name?*
 - in *createNS* 55
 - in *DeleteFile* 144
 - in *destroyNS* 56
 - in *lookupNS* 55, 63
 - in *NameErr* 64
 - in *open* 66
 - in *ReadFile* 144
 - in *ReadStoredFile* 148
- Name* 140, 213
- NameErr* 64
- NDEV* 215
- Δ *NDEV* 215
- newelement?*
 - in *InsertNewElement0* 114
- newname?*
 - in *linkNS* 62
 - in *moveNS* 62
- newposn?*
 - in *seekAS* 54
 - in *seekCHAN* 51
- newprops?*
 - in *InsertNewProperties0* 114
- nickname*
 - in Δ *RS* 156
- Nickname* 151
- nil* 102
 - in *ACTION* 170
- NIN* 214
- Δ *NIN* 214
- Ξ *NIN* 215
- niq*
 - in *NIN* 214
- No*
 - in *Status* 32
- NoDuplicates* 12
- NoFreeCids*
 - in *REPORT* 63
- NonDecreasing* 10
- NoneLeft* 207
- NonExistent* 209
- noq*
 - in *NOUT* 213
- NoSpace*
 - in *CONDITION* 207
- NoSuchCid*
 - in *REPORT* 63
- NoSuchName*
 - in *REPORT* 63
- NoSystem* 210
- Not_Available*
 - in *Report* 156
- NotAvailable* 160

- NotFound* 23
- NotPresent* 17
- NotQuiteSoAuthoritarianDD* 103
- Not_within_any_block*
 - in *Report* 16
- NOUT* 213
- Δ *NOUT* 213
- Ξ *NOUT* 215
- now*
 - in *NotAvailable* 160
 - in *Reserve* 158
 - in *Scavenge* 159
 - in *TooManyClients* 161
- NReceive* 215
- NReceive_0* 215
- ns!*
 - in *MapSet* 216
 - in *ToSet* 214
- ns?*
 - in *NSendM_0* 214
- NS* 56
- NS0* 55
- Ξ *NS0* 55
- NSend* 215
- NSend_0* 214
- NSendM* 215
- NSendM_0* 214
- NSendReceive* 215
- NSendReceive_0* 215
- nstore*
 - in *NS0* 55
- nts*
 - in Δ *NTS* 210
- nts'*
 - in Δ *NTS* 210
- NTS* 210
- Δ *NTS* 210
- Φ *NTS* 210
- null* 108
- num* 127, 130
- number?*
 - in *ReadData* 145
 - in *ReadStoredFile* 148
- Number* 110
- offset?*
 - in *readFILE* 45
 - in *seekoffset* 69
 - in *writeFILE* 46
- Ok*
 - in *REPORT* 63
- OK*
 - in *Report* 16
- oldname?*
 - in *linkNS* 62
 - in *moveNS* 62
- op* 127, 130
- opcode* 127, 130
- open* 64, 66
- openCS* 52
- openFS* 59
- operand* 127, 130
- operation* 60
- OPSYM* 126, 130
- other!*
 - in *Engaged* 32, 35, 37
- out* 129
 - in *ASSEMBLY* 130, 136
 - in *Implementation* 135
 - in *Phase2* 134
- out!*
 - in *Merge* 12
 - in *Sort* 11
 - in *SortNoDup* 12
- Out*
 - in *Response* 122
- OutofBounds* 207
- OutOfSpace* 207
- outputs* 5
- override*
 - distributed 20
- owner*
 - in *DD1* 102
 - in *DD9* 107
 - in *DD13* 118
 - in *FILE* 142
- p*
 - in *TSQ* 203
- p?*
 - in *seekoffset* 69
- PATH* 25
- Period* 121
- ph?*
 - in Δ *TN* 31, 34, 36
- Phase1* 133
- Phase2* 134
- phone?*
 - in *Break* 33, 35, 37
 - in *Fix* 33, 35, 37
- PHONE* 29
- pieces!*
 - in *ReadData* 145
 - in *ReadStoredFile* 148
- pieces?*
 - in *UpdateData* 146
- Pool* 85
- Δ *Pool* 85

- Ξ Pool 85
- Pool_Init 85
- posn
 - in CHAN 50, 69
- powerset 8
- Pre 217
- PreASSEMBLY 131
- PreIn 217
- Present 17
- priv
 - in DD7 105
 - in DD9 107
- privacy 110
- promotion 21
- properties?
 - in Display 109
- props?
 - in DeleteProperties 115
- q
 - in ΔQ 205
 - in QLike 206
- q'
 - in ΔQ 205
- Q 205
- ΔQ 205
- QIdErr
 - in CONDITION 207
- QLike 206
- queue!
 - in TSRemote 211
- queue?
 - in CreateQ 209
 - in DeleteQ0 209
 - in NonExistent 209
 - in ΦTS 208
 - in TSRemote 211
- r 212
- r'
 - in R_U_Book 77
 - in R_U_Cancel 77
 - in R_U_Del_Res 78
- range (ran) 4
- read 64, 65
- Read 207
- Read0 205
- readable!
 - in Display 109
- readAS 53, 54
- readCHAN 51
- ReadData 145
- readFILE 45
- ReadFile 144
- readFS 59
- ReadOut 124
- ReadQ 209
- ReadQN0 210
- ReadQN1 211
- ReadQN2 211
- readSS 49
- ReadStoredFile 147, 148
- Rebate 144
- Record 7
- recorded
 - in Diary 87
- $R \equiv U$ 81
- $R \equiv U_Book$ 81
- $R \equiv U_Cancel$ 81
- $R \equiv U_Init$ 81
- ref 127, 130
- RelMinutes 123
- rem 212
- remote 211
- Remove 207
- Remove0 204
- RemoveQ 209
- RemoveQN0 210
- RemoveQN1 211
- RemoveQN2 211
- Remove_Visitor_from_Meeting 95
- Remove_Visitor_from_Meeting_0 87
- rep!
 - in BDelete 26, 27
 - in BReplace0 21
 - in BReplace1 23
 - in Empty 23
 - in NotFound 23
 - in NotPresent 17
 - in Present 17
 - in STDelete 26
 - in STReplace 26
 - in Success 17
- repeat 212
- Replace 12, 16
- report!
 - in DeleteQ0 209
 - in ERROR 207
 - in ΔFS 63
 - in NonExistent 209
 - in NoSystem 210
 - in open 66
 - in read 65
 - in ΔRS 156
 - in ServiceError 160
 - in Successful 207
- Report 156, 16
- REPORT 63

- reqs*
 - in *efficientTN* 31
 - in *TN* 30, 34, 36
- ReservationCost* 158
- Reserve* 158
- Response* 122
- retrieval* 110
- $R \ll U$ 80
- $\Delta R \ll U$ 80
- $\exists R \ll U$ 80
- $R \ll U_Book$ 80
- $R \ll U_Cancel$ 81
- $R \ll U_Del_Res$ 81
- $R \ll U_Del_User$ 81
- $R \ll U_Init$ 80
- robust 16
- RootFid* 57
- $R \gg U$ 79
- $\Delta R \gg U$ 80
- $\exists R \gg U$ 80
- $R \gg U_Book$ 80
- $R \gg U_Cancel$ 80
- $R \gg U_Del_Res$ 80
- $R \gg U_Del_User$ 80
- $R \gg U_Init$ 79
- RS* 156
- ΔRS 156
- rsvd*
 - in *Dining_State* 90
- rt*
 - in *IS* 133
- ru*
 - in *Hotel_State* 84
 - in *R_U* 76
- R_U* 76
- ΔR_U 76
- R_U_Book* 77
- R_U_Cancel* 77
- R_U_Del_Res* 78
- R_U_Del_User* 78
- R_U_Init* 76
- s?*
 - in *Add* 12, 15
 - in *BDelete* 26, 27
 - in *BLocate0* 27
 - in *BReplace0* 21
 - in *BReplace1* 23
 - in *BSearch0* 20
 - in *Delete* 16, 6
 - in *LookUp* 15, 6
 - in *NotFound* 23
 - in *NotPresent* 17
 - in *Present* 17
 - in *Replace* 12, 16
 - in *STDelete* 26
 - in *STReplace* 26
 - in *Update* 5
- Scavenge* 159
- schema
 - conjunction 17
 - disjunction 18
 - horizontal 14
 - inclusion 15
 - operation 5
- seekAS* 53, 54
- seekCHAN* 51
- seekoffset* 69
- sequence (seq) 9
- Service_Error*
 - in *Report* 156
- ServiceError* 160
- Session* 73
- SetShutdown* 159
- shift 46
- shutdown*
 - in *RS* 156
- shutdown?*
 - in *SetShutdown* 159
- SI* 73
- size*
 - in *seekoffset* 69
- size!*
 - in *fstat* 55
- Size* 143
- Sort* 11
- SortNoDup* 12
- SS* 47, 67, 68
- ΦSS 49
- st*
 - in *Add* 12
 - in *Delete* 6
 - in *IS* 133
 - in *LookUp* 6
 - in *Replace* 12
 - in ΔST 14
 - in *STDelete* 26
 - in *STReplace* 26
 - in *Update* 5
- st'*
 - in *Add* 12
 - in *Delete* 6
 - in *LookUp* 6
 - in *Replace* 12
 - in ΔST 14
 - in *STDelete* 26
 - in *STReplace* 26
 - in *Update* 5

- sto* 14
- ST* 14
- ΔST 14
- ΞST 14
- STAdd* 17
- Standard_Append* 205
- Standard_Hours*
 - in *Flexi* 122
- Standard_Remove* 205
- start?*
 - in *ReadData* 145
 - in *ReadStoredFile* 148
 - in *UpdateData* 146
- state* 14, 5
- Status* 32
- STDelete* 17, 26
- STLookUp* 17
- store*
 - in *DD0* 102
 - in *DD1* 102
 - in *DD9* 107
 - in *DD13* 118
- StoreFile* 143
- STReplace* 17, 26
- success* 63
 - in *CONDITION* 170
- Success* 17
 - in *CONDITION* 207
 - in *Report* 156
- Successful* 207
- SYM* 126, 130, 13, 4
- symbol table 3
- Symbol_not_found*
 - in *Report* 16
- Symbol_not_present*
 - in *Report* 16
- Symbol_present*
 - in *Report* 16
- symtab* 127
- sysid!*
 - in *TSRemote* 211
- sysid?*
 - in *NoSystem* 210
 - in ΦNTS 210
 - in *TSRemote* 211
- SysId* 210
- SysIdErr*
 - in *CONDITION* 207
- system*
 - in *ACTION* 170
- t?*
 - in *Create* 85
 - in $\Delta Clocking$ 122
 - in $\Delta Diary$ 88
 - in ΔR_U 76
 - in *Destroy* 85
- Tariff* 143
- threatens!*
 - in *SetShutdown* 159
- Time* 73, 121, 142
- TN* 30, 34, 36
- ΔTN 31, 34, 36
- Too_Many_Clients*
 - in *Report* 156
- TooManyClients* 161
- ToSet* 214
- ToSetL* 217
- ToSetL_LSendM* 217
- ToSet_NSendM_0* 214
- TR* 73
- Transport_Init* 84
- Transport_State* 84
- ts*
 - in ΔTS 208
- ts'*
 - in ΔTS 208
- TS* 208
- ΔTS 208
- ΦTS 208
- TSElem* 203
- TS_Initial* 208
- TSQ* 203
- ΔTSQ 204
- TSQ_Initial* 203
- TSQName* 208
- TSRemote* 211
- Types* 112
- u?*
 - in *FileUpdate* 8
 - in *R_U_Book* 77
 - in *R_U_Cancel* 77
 - in *R_U_Del_User* 78
- Unknown* 124
- unlinkFS* 59
- unlinkFS0* 59
- until!*
 - in *NotAvailable* 160
 - in *Reserve* 158
- Update* 5
- UpdateData* 146
- UpdateStoredFile* 148
- ur*
 - in *Hotel_State* 84
 - in *R_U* 76
- usedfids*
 - in *FS* 57

- user*
 - in *DDS* 108
 - in *Destroy* 85
 - in *Erase* 88
- users*
 - in *Hotel_State* 84
 - in *R_U* 76
- users_D*
 - in *Conference_State* 89
 - in *Meeting_State* 86
 - in *Transport_State* 84
- v!*
 - in *BLocate0* 27
 - in *BSearch0* 20
 - in *LookUp* 15, 6
- v?*
 - in *Add* 12, 15
 - in *BReplace0* 21
 - in *BReplace1* 23
 - in *Delete_Visitor_0* 97
 - in *Replace* 12, 16
 - in *STReplace* 26
 - in *Update* 5
- V* 73
- VAL* 13, 4
- Visitor_Init* 92
- Visitor_State* 91
- V_OP* 96
- V_SYS* 93
- ΞV_SYS 94
- wait*
 - in *ACTION* 170
- when*
 - in ΔFS 143
- who*
 - in ΔFS 143
- worked*
 - in *Flexi* 122
- Worked* 123
- Write* 207
- Write0* 204
- writeAS* 53, 54
- writeCHAN* 51
- writeFILE* 46
- writeFS* 59
- WriteQ* 209
- WriteQN0* 210
- WriteQN1* 211
- WriteQN2* 211
- writeSS* 49
- x?*
 - in *Add* 88
 - in *Create* 85
- Xi* (Ξ) 14
- Yes*
 - in *Status* 32
- zero* 46
- ZERO* 46