To
my wife Ulla

for helping me through difficult times
and for all the support and encouragement
which she has given me.

Timor:

An Object- and Component Oriented Language

J. Leslie Keedy

Preface

This book records one of the results of an Odyssey which has lasted for more than fifty years of my life, beginning with my work in the design team of the VME operating system for the ICL 2900 Series of computers in Kidsgrove, England. This was followed by my founding the MONADS operating system group at Monash University in Melbourne Australia, with follow up work on MONADS in the groups which I later led at the University of Darmstadt in Germany, the University of Newcastle, N.S.W., Australia and the University of Bremen in Germany. My final professional move was to the University of Ulm in Germany, where I founded the SPEEDOS project and the Timor Project in the Department of Computer Structures.

At heart I am an operating system designer, but I have also long been interested in programming language design. This is inevitable because operating system designers often develop new concepts which cannot easily be programmed in existing high level languages. From the programming language perspective I have been fortunate to have been accompanied through most of my career by Dr. Mark Evered, who started his PhD work at Monash and then accompanied me to Darmstadt, where he produced an excellent PhD based on the LEIBNIZ programming language [1], which laid particular emphasis on information hiding module structures and on an abstract concept for collections of items in programming languages, both of which subsequently played an important role in Timor. Later when I returned from Australia to Germany he re-joined the team in Bremen and subsequently moved with me to Ulm, where he informally led our programming language research work, which since Bremen included Dr. Gisela Menger, whose PhD work [2] concentrated mainly on developing concepts associated with collections, and Dr. Axel Schmolitzky, whose PhD work [3] laid the foundations for the Timor idea of co-types. Soon after Dr. Evered and Dr. Schmolitzky had left the group we were joined by Dr. Christian Heinlein, whose knowledge of other programming languages greatly helped in formulating Timor concepts, despite continuing his work on a quite different dissertation. Unfortunately Dr. Evered had left to take a position in Australia before the work specifically concerned with Timor began.

The main aim of Timor at that time was to provide a suitable programming language for SPEEDOS [4], which was our operating system research project in Ulm, as a successor to the MONADS project. A new language was necessary because key features of SPEEDOS could not be programmed in conventional languages. A fundamental feature of SPEEDOS (as previously of MONADS) was that it eliminated a conventional file system by providing a persistent virtual memory which was populated by information hiding modules [5] as the basic software objects visible to applications. An important consequence of this idea was that Timor had to support a much more rigorous concept of objects than is common in other object-oriented languages, e.g. by separating type definitions from their potentially multiple implementations and by separating n-ary and similar operations from instance methods. Furthermore SPEEDOS required linguistic support for its solution of the confinement problem (which is still the most serious security problem in operating systems[1]); this allows each module to have its own specialised, user programmed 'firewalls' using a new type which we called qualifying types.

Following my retirement I continued to develop the ideas from the MONADS and SPEEDOS projects, considerably extending and improving on the original versions and working out how to implement some of the wilder concepts efficiently, such as the worldwide unique virtual memory and addressing. The final results of that work include the formulation of a new model for secure operating system design. This is described in a two volume book entitled "Making Operating Systems Secure". The first volume develops an overview of this new model (which I call ModelOS), while the second volume provides a detailed explanation of how the model can be implemented. ModelOS currently exists only as a design, since an implementation would be extremely expensive, involving a new hardware design for a CPU (competition for Intel) and an operating system comparable in its scope with those of Microsoft and Apple systems.

The work on Timor has been adapted to suit the ModelOS design, and the resultant language is described in this book. Consequently Timor is an object-oriented persistent programming language. The basic software units seen by users of ModelOS are not directly comparable with files in conventional systems but are modules which strictly conform to the information hiding principle. Timor allows its programmers to separate the definition of these units from their implementations, such that a type can have multiple implementations. This

1 The confinement problem is the issue of how to prevent information from escaping from a module, i.e. the problem which we hears about every day as hackers penetrate systems and steal or alter information.

means for example that collection types can serve a role similar to data files in conventional systems where the different implementations might correspond to the different implementation approaches in current systems, e.g. as sequential, indexed sequential files or as B-Trees, etc. But Timor (and ModelOS) modules can serve quite different purposes, e.g. as multiple entrypoint programs, as mathematical libraries, as module directories, as operating system modules, etc.

For a user or application programmer, the code of a module is inseparable from the data structures which it uses, and this has advantages for the security of a system. A hacker cannot simply write a program to access a data file, as he can in conventional systems. The ModelOS kernel ensures that the data can only be accessed via the information hiding routines as defined in the various implementations of its type definition.

The main purpose of this book is to introduce the Timor programming language, and in particular the new and/or unconventional ideas which it contains, such as the separation of types from implementations, its unusual approach to inheritance, attribute types, qualifying types, co-types, its support for genericity (including generic functions) and its approach handling collections, as well as some special features for supporting the ModelOS operating system.

Leslie Keedy
BREMEN 2021

Table of Contents

Preface .... **i**

Table of Contents .... **iv**

List of Figures .... **xiii**

Chapter 1 Introduction **1**

1 The Aims of Timor 1

1.1 Support for ModelOS Applications 1

1.2 Support for the Development of a Genuine Components Industry 5

1.3 Research into Object-Oriented Programming 5

1.4 Support for Modelling and Implementing Database Applications 5

2 An Overview of Timor 6

3 Designing and Implementing Systems 6

Chapter 2 Control Structures **8**

1 Iteration Statements 8

2 Conditional Statements 8

3 `With` Blocks 9

3.1 What can be Nominated in a Timor Block 9

3.2 Interpreting Identifiers in a `With` Block 10

3.3 The Scope of a With Block 11

4 Exception Handling 11

5 Further Syntax 12

6 Pragmas 12

Chapter 3 The Basic Structure of Timor Programs 13

1 Identifiers 13

2 Enumeration Types 14

2.1 Simple Enumeration Types 14

2.2 Sequences 15

2.3 Circular 15

2.4 Lists of Enumeration Values 16

3 Type Definitions 16

4 Implementations.. 17

4.1 Retained Data.. 18

4.2 Multiple Implementations.. 18

5 Instance Methods 19

6 Open and Close Methods 20

7 Callback Methods 21

8 Parameters.. 21

8.1 Parameter Declarations.. 21

8.2 Default Parameter Values.. 21

9 Protected Methods 22

10 Internal Methods 22

11 An Example Type Definition 22

12 An Example Implementation.. 23

Chapter 4 Instances, Values, Objects and Modules...................................... 26

1 The Basic Types.. 27

2 Value Variables... 27

3 References and Objects... 28

4 Capabilities and File Modules 29

5 Library Modules 30

6 Conversion between Modes.. 31

7 Numerical Representation 31

8 Instance Records.. 31

9 Shared Objects and Collections... 31

Chapter 5 Abstract Variables and Records **33**
1 Standard Implementations of Abstract Variables 34
2 Records 35
3 Using the Methods 36
4 Final and Constant Values 36
5 Fixed and Constant References 37
Chapter 6 Inheritance in Conventional Object Orientation **38**
1 Subtyping and Subtype Polymorphism 38
2 Diamond Inheritance 40
3 Multiple and Repeated Inheritance 41
4 Method Redefinition 41
5 Subclassing and Code-Reuse 41
6 Overriding Methods 42
7 Conclusion 42
Chapter 7 Type Inheritance in Timor **43**
1 Single Inheritance 43
2 Method Redefinition 43
3 Views 44
4 Multiple and Repeated Inheritance 45
5 Diamond Inheritance 47
6 Inheriting from a Common Abstract Ancestor 48
7 Type Rules 50
8 Handles 50
9 Cast Statements 51
10 Comparison Operators and Subtyping 51
Chapter 8 Implementations and Code Re-Use in Timor **53**
1 Re-Use Variables 53
2 Clashing Methods in Re-Use Variables 54
3 Reversing the Re-Use Relationship of Subtypes 55
4 Re-Use of Independent Types 55
5 Overriding Code 56
6 Implementing Views 56

7 Implementing Multiple and Repeated Inheritance 56
8 Implementing Diamond Inheritance 57
9 Implementing Types with a Common Abstract Ancestor 58

Chapter 9 Attribute Types 59

1 Defining and Implementing Attribute Types... 59
2 Static Use of Attributes... 61
3 Instantiating Attribute Types 61
4 Attaching Attributes Dynamically to Objects 61
5 Removing Attributes from Objects... 62
6 Casting with Attributes 62
7 Final Remark... 63

Chapter 10 Qualifying Types 64

1 Qualifiers: The Basic Idea 65
1.1 Call-In Bracket Methods 65
1.2 The Body Statement 66
1.3 Augmenting Bracket Routines 66
1.4 Testing Bracket Methods.. 66
1.5 Replacing Bracket Methods 67
1.6 Multiple Qualifiers 67
1.7 Call-Out Bracket Methods.. 67
2 Qualifying All the Methods of Any Type in the Same Way..................... 69
3 Distinguishing Between Reader and Writer Methods 70
4 Qualifiers with Instance Methods... 71
5 Qualifying Specific Methods... 73
6 Call-Out Methods 75
7 Combining Call-Out and Call-In Brackets 76
8 Instantiating and Using Qualifiers.. 79
8.1 Qualifying Target Objects Dynamically 79
8.2 Qualifying Target Objects Statically 80

Chapter 11 Co-Types 83

1 The Basic Structure of a Co-Type 83
2 The Maker Section... 84

3 The Instance Section 85
4 The Binary Section 86
5 A Simple Co-type Implementation 86
6 Accessing Parameters at the Implementation Level 87
7 Derivation and Adjustment of Co-Types 89
8 Syntactic Features Limited to Co-Types in an Adjustment Hierarchy 90
9 Covariant Makers 91
10 Covariant Instance Methods 91
11 Modifying Co-Type Definitions 91
12 Merging Co-Type Methods which Result from Diamond Inheritance 91
13 Merging Multiply Adjusted Co-Types for Parts 92
14 Implementing Adjustment Hierarchies 92
15 A Further Example of Co-Types 93
15.1 Co-types for Standard Input-Output Operations 93
15.2 Adjusting Input-Output Methods in Co-types 96
15.3 Using Inout Methods 97
16 Access to Co-Types and other Components 99

Chapter 12 Generic Types and Implementations 101

1 Generic Templates 101
2 Type Templates 102
3 Implementation Templates 103
4 Actualising Templates 103
5 Deriving Templates by Single Inheritance 103
5.1 Deriving Templates by Extension 103
5.2 Deriving Templates by Inclusion 105
6 Deriving Type Templates by Multiple Inheritance 106
6.1 Deriving from Templates with a Common Ancestor Template 106
6.2 Deriving from Templates without a Common Ancestor Template 106
7 Generically Defined Views 107
8 Implementing Generic Types and Code Re-Use 107
9 Implementing Generic Co-Types 107
10 Generic Function Parameters 108

Chapter 13 The Basic Timor Collection Types 109

1 The Collection Types 111

1.1 The Abstract Type Collection 111

1.2 The Concrete Type Bag 112

1.3 The Ordered Types 113

1.4 The Unordered Duplicate Free Types 115

1.5 The Ordered Duplicate Free Types 116

2 The Collection Implementations 118

2.1 A Consolidated Definition of the Type List 118

2.2 An Array Implementation of the Generic Type List 119

2.3 Implementing the Remaining Types 122

3 Co-Types for the TCL Collection Hierarchy 129

3.1 A Co-Type for the Base Type Collection 129

3.2 Implementing the Co-Type Adjustment Hierarchy 132

4 Collection Syntax 137

4.1 Collection Literals: 137

4.2 Subcollection Selection 137

4.3 Element Selection 138

4.4 Type Conversion 139

4.5 Collection Operators 139

4.6 Boolean Expressions 139

4.7 Iteration 140

4.8 Implementing the `for` Statement 141

Chapter 14 Generic Function Parameters 145

1 Function Clauses 146

2 Motivation for Static Function Parameters 146

3 Defining Static Function Parameters in Type Templates 147

4 Using Static Function Parameters in Implementations 148

5 Actualising Static Function Parameters 150

5.1 A Suggested Implementation 150

Chapter 15 Support for ModelOS 152

1 Returning Values of User-Defined Types 152

2 Handling ModelOS Access Rights ... 153
3 Calls to the ModelOS Kernel ... 154
3.1 Executing Simple Kernel Instructions ... 154
3.2 Executing Kernel Instructions involving Access and Control Rights ... 155
3.3 Normal Execution of Inter-Module and Similar Calls ... 156
3.4 Callback Calls ... 156
3.5 Direct Execution of Inter-Module and Similar Calls ... 156
4 Synchronisation and Semaphores ... 158
4.1 General Semaphores ... 158
4.2 Resource Set Semaphores ... 158
4.3 Reader-Writer Exclusion ... 159
4.4 Access to Basic Semaphore Variables ... 159
4.5 Higher Level Synchronisation ... 160
Chapter 16 Why Timor Does Not Need Wildcards ... 161
1 Upper Bounded Wildcards ... 161
1.1 Why Is a List of Students not a Subtype of a List of Persons? ... 162
1.2 The Wildcard Solution ... 163
1.3 The First Timor Solution: Generic Co-Types ... 163
1.4 Using Restricted Variables and Parameters as an Alternative to Upper Bounded Wildcards ... 165
1.5 The Restrictor OOPS ... 165
2 Unbounded Wildcards ... 166
2.1 Handles and Unbounded Wildcards ... 166
2.2 Further Aspects of Working with Handles ... 168
2.3 Casts Involving Co-types ... 169
3 Lower Bounded Wildcards ... 170
3.1 Comparators and Co-Types ... 170
3.2 Type Matching and Casts ... 171
3.3 Complementing Casts with Restrictions ... 172
Chapter 17 Concluding Remarks ... 175
1 Support for ModelOS Applications ... 175

2 Flexible Support for the Design and Development of Software Components ... 176

3 Improving the Object Oriented Paradigm ... 177

4 Support for Database Applications ... 178

4.1 Remote Databases ... 179

4.2 Separating Types from Implementations ... 179

4.3 Kinds of Database ... 179

4.4 Flexible Database Record Entries ... 180

4.5 Persistence ... 180

4.6 Big Data ... 180

APPENDIX I A Timor Object Model ... 182

1 Instance Records ... 182

2 The Object Table ... 183

3 Capabilities ... 184

APPENDIX II The Timor Operators ... 186

1 The Null Value and Null Exceptions ... 186

2 Arithmetic Operators ... 186

2.1 Operations on Numerical Values ... 186

2.2 Collection Operators ... 186

2.3 Operations on Dynamic Attributes ... 186

3 Assignment Operators ... 187

3.1 Value Assignments ... 187

3.2 Reference and Capability Assignments ... 188

3.3 Assignments Involving Restrictions ... 188

3.4 Shorthand Assignment Operators ... 188

4 New, Create, Delete and Dereferencing Operators ... 188

5 Comparison Operators ... 189

5.1 Comparing Values ... 189

5.2 User-Defined Types: Comparing Using the Comparison Operators ... 190

5.3 Comparing Subtypes via the Value Operators ... 190

5.4 Comparing References and Capabilities ... 190

6 Logical Operators ... 191

7 The Conditional Operator 191
8 Bit Manipulation Operators 191
9 Defining/Redefining Operators in Co-Types 191
9.1 Associating Binary Comparison Methods with Operators 191
9.2 Associating Binary Operations with Operators 192
9.3 Definitions in Multiple Co-Types 192
Appendix III: EBNF for the Timor Programming Language 193
References 197
Bibliography 201
Acknowledgements 205

List of Figures

Figure 1.1: A Bank Accounts Module 2
Figure 1.2: A Compendium of Games 3
Figure 1.3 Access Rights expressed as Semantic Operations 4
Figure 6.1: Diamond Inheritance 40
Figure 6.2: Multiple and Repeated Inheritance 41
Figure 7.1: Structure of the TCL Collection Types 49
Figure 10.1: A Normal Method Invocation 65
Figure 10.2: A Qualifying Type with a Call-In Bracket Method 65
Figure 10.3: An Augmenting Bracket Method 66
Figure 10.4: A Testing Bracket Method 67
Figure 10.5: A Replacing Bracket Method 67
Figure 10.6: A Qualifier with Call-In and Call-Out Bracket Methods 68
Figure 10.7: A Client with Call-Out and a Target with Call-In Brackets 68
Figure 11.1: Parallel Hierarchies for Subtypes and their Co-Types 89
Figure 13.1: Structure of the TCL Collection Types 110
Figure AI.1: An Instance Record for a Structured Type 183
Figure AI.2 Sharing an Object 184

Chapter 1
Introduction

Timor[2] has the following four primary aims:

- support for applications designed for the ModelOS system [6],
- support for the development of a genuine components industry,
- research into object-oriented programming, and
- support for modelling and implementing database applications.

After briefly introducing these aims, an initial overview of how they have affected the design of Timor is provided.

1 The Aims of Timor

1.1 Support for ModelOS Applications

The first motivation for developing Timor was to support the development of application programs, initially for SPEEDOS [4] but later for the ModelOS system [6]. The reason for this is that ModelOS provides a fundamentally different computer operating system architecture from that of conventional system architectures. Here are a few examples.

First, it is a *persistent* virtual memory system, which means that the conventional distinction between a temporary computational virtual memory and a persistent file system disappears[3]. In ModelOS all applications execute directly in a persistent virtual memory. This has the great advantage that no separate file system is needed. Consequently the fundamental difference in conventional sys-

---

2 Timor is an acronym for "**T**ypes, **I**mplementations and **MOR**e".

3 A forerunner of this idea (known as "direct addressability") was pursued by the designers of the famous MULTICS system developed at MIT in the 1960s [38, 37] but this was unfortunately not achievable in a satisfactory way at that time because of the inadequacies of the available hardware. For more details, see chapter 12 of [6].

tems between data structures for

(a) temporary data items in programs executing in conventional virtual memory and

(b) for persistent files held in a separate file system

does not exist in ModelOS[4]. This has many advantages, the most important of which from the Timor viewpoint is that Timor does not need to provide a bridge between two ways of programming data structures. The data structures provided by a Timor application (in a ModelOS environment) are automatically persistent.

This means in practice that those kinds of data which typically are directly supported in programming languages (e.g. individual variables such as integers and boolean values as well as arrays and linked lists, etc.) are held in ModelOS in the persistent virtual memory. This results in a much simpler Timor design.

The content of a ModelOS persistent virtual memory is not structured like conventional systems. The latter distinguish between application programs and files, whereas ModelOS supports a single major structure in the form of information hiding modules with multi-entrypoints [5], which are known in Timor and ModelOS as semantic routines. These can in practice be used both as persistent files (together with their semantic routines) (see Figure 1.1) and as application modules (see Figure 1.2). For a more detailed discussion of information hiding see chapter 13 of [6].

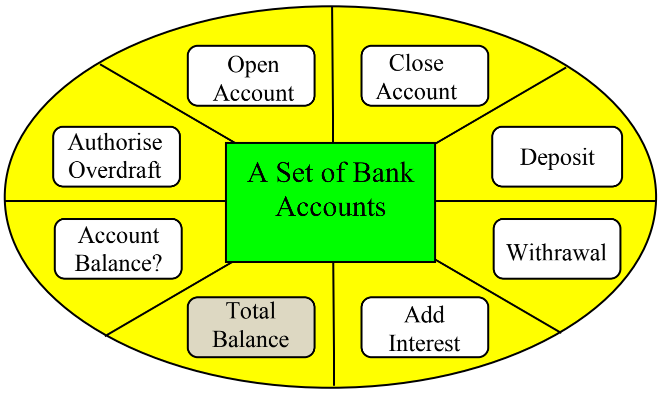

Figure 1.1: A Bank Accounts Module

[4] Persistent programming was initially the focus of work at the University of Glasgow (under M. P. Atkinson) and the University of St. Andrews (under R. Morrison) in the 1980s based on an idea called orthogonal persistence [35]. Whereas their aim was to provide mechanisms which could be implemented using conventional hardware for their languages PS-Algol [39] and Napier [40], Timor simply presupposes that appropriate hardware exists (via the ModelOS environment as described in chapter 12 and in the appendix to volume 1of [6]).

A further advantage of this organisation is that it eliminates the need for a special way of starting applications, such as Java's public `static void main (String[] args)`. In Timor any semantic routine of any module can in principle be invoked from any other module (subject to a ModelOS check that the caller has appropriate access rights). In order to invoke a semantic routine a ModelOS/Timor thread must present a *capability* (which is a ModelOS-protected reference) for the module. Such a capability can be created in Timor in a manner similar to the way it creates internal values and object references within a module (see chapter 4). Capabilities contain access rights, which can also be changed (i.e. reduced) in Timor. Normal parameters can be directly passed from one module to another as part of an *inter-module call*.

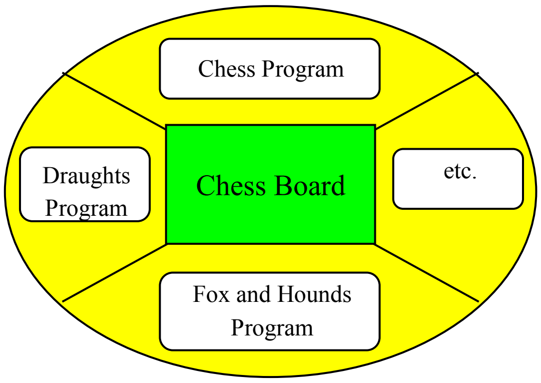

Figure 1.2: A Compendium of Games

For a user or application programmer, the code of a module is inseparable from the data structures which it uses, and this has advantages for the security of a system. A hacker cannot simply write a program to access a data file, as he can in conventional systems. The ModelOS kernel ensures that the data can only be accessed via the information hiding routines as defined in the various implementations of its type definition.

Supporting modules with semantic routines has a further security advantage. The interface routines of a module can be expressed at a much higher level than in conventional systems, e.g. for a bank account module routines such as deposit, withdraw, add interest, transfer, authorize overdraft, etc. can be defined as an information hiding module. The ModelOS kernel ensures that these routines can be individually protected and that threads can only invoke those routines of a module for which it has permission. This provides a much more appropriate implementation of access rights than in conventional systems, for example by providing bankers with only the access rights to bank accounts which their work and positions entitle them, as the following figure shows.

	Teller	Branch Manager	H.O. Accountant	H.O. Auditor
Open Account	√	√	x	x
Close Account	√	√	x	x
Deposit	√	√	x	x
Withdraw	√	√	x	x
Transfer	√	√	√	x
Add Interest	x	x	√	x
Authorise Overdraft	x	√	x	x
Customer Number	√	√	x	√

A tick indicates that the subject at the head of the column may carry out the operation in the corresponding row.

Figure 1.3 Access Rights expressed as Semantic Operations

It may come as a surprise to some readers that Timor does not provide a particular model for parallel processing, multithreading, etc. The reason for this is quite simple. ModelOS provides several basic mechanisms based on semaphores (see chapter 15 section 4 and also chapters 20 to 22 of [6]) which allow threads to synchronise their activities. At a higher level the basic operating system provides mechanisms which allow processes and their threads[5] to be created and managed, see chapter 31 of [6]. This is all achieved via inter-module calls, which in Timor simply appear as normal calls to other modules, as is described in chapter 4 below.

As a result of the above approach Timor supports any model for organising parallel processing, provided that this is based on "in-process" principles[6] of process/thread cooperation.

It is perhaps equally surprising for some that no Timor mechanisms are provided to support the Internet. This is made superfluous by the fact that inter-module calls in ModelOS (and therefore Timor) can function in appropriate cases in a manner similar to conventional remote procedure calls. The destination of a remote inter-module call is located by ModelOS automatically, using in-

5 The ModelOS/Timor concept of processes and threads differs substantially from that found in most operating systems and programming languages. It is based on a rigorous "in-process" model (see chapter 8 sections 7ff and chapter 15 of [6]).

6 If you do not know what the 'in process' model is, see chapters 8 and 15 of [6].

formation (managed by ModelOS) in the capability used to make the call. Timor is totally unaware of the fact that calling some modules involves a remote call (see chapters 27 to chapter 29 of [6]). For security reasons ModelOS users normally only use this remote inter-module call facility when using the Internet; however in exceptional situations a mechanism (implemented via ModelOS modules) is provided to allow users to use conventional email, websites, etc. (see chapters 34 and 35 of [6]).

ModelOS is described in detail in the two volume book "ModelOS – Making Computers Secure" [6].

1.2 Support for the Development of a Genuine Components Industry

The second aim was to design a language which can easily support the idea of components (in the sense of components and component industries, as found for example in the car industry). In contrast with the currently established view of software components, Timor aims to realise McIlroy's vision [7] that software components need not be large, but can be quite small (e.g. a `Person` object or a `Date` object); such components can then be built up into larger components. In our view this philosophy is best realised in an object oriented style, with the help of a strict interpretation of the information hiding principle [5]. An important aspect of the module concept in Timor is that type definitions for modules can be used both as major modules at the operating system level, as described in section 1 above, and as internal components of such a module, as we shall see later.

1.3 Research into Object-Oriented Programming

The third aim was to carry out research into the structures of object-oriented programming languages in order to address certain problems which arise in practice (e.g. with respect to the relationship between subtyping and subclassing), and to examine why binary methods can be troublesome, in the hope of designing a language which does not have such problems.

1.4 Support for Modelling and Implementing Database Applications

The final aim was to provide strong support for modelling and implementing database applications. This is not usually seen as a central aim for object-oriented programming but it becomes essential in the ModelOS/Timor context, where there is no extra file system in which databases (specially for business applications) can be modelled and developed. In Timor a module can be viewed as a file which is protected by its semantic routines, but it can also be viewed as a collection of programs or library routines.

2 An Overview of Timor

Pursuing the above aims has led to the design of a somewhat unconventional object oriented programming language, which

- replaces the class construct by a type definition that can potentially have a number of different implementations [8, 9], each with a single constructor which can have *implementation*-oriented parameters that can differ in different implementations of the same type;
- supports inheritance in the case of subtype hierarchies which derive from a common *abstract* ancestor, where the subtypes primarily vary the behaviour of their supertypes rather than add new methods (although new methods can also be added), e.g. as in the case of a collection hierarchy [8];
- adds the concept of *views*, which are incomplete types (with implementations), that can be usefully incorporated into different type definitions [8];
- supports diamond inheritance [10], and multiple and repeated inheritance from separate types, using a technique known as parts inheritance [11];
- replaces subclassing by a flexible new implementation technique based on *re-use variables* [12, 9];
- introduces a new kind of component, known as a *qualifying type* [13, 14], which contains *bracket methods* that allow instance methods of other objects to be "qualified" in a modular way, e.g. to protect or synchronise them, thus supporting the separation of concerns;
- provides uniform support for distribution and persistence in the form of persistent objects [15] and persistent processes [16];
- introduces an unusual way of handling *makers* (the Timor name for *application*-oriented constructors), binary methods, and class (static) variables and methods, in a new kind of type, known as a *co-type* [17], which can be automatically adjusted covariantly to reflect a subtype hierarchy [18].
- supports genericity in forms which reflect the unusual features of Timor, adding *function parameters* which allow programmers considerable flexibility, for example by allowing a programmer to redefine what is meant by such issues as equality.

These and other aspects of Timor will be described in the following chapters.

3 Designing and Implementing Systems

Timor can be used both for 'programming in the large' and for 'programming in the small', to use the terminology coined by Frank DeRemer and Hans Kron [19]. However, in contrast with their view that a separate module interconnection language is needed for programming in the large, the strict use of information hiding and of the separation of types and their implementations in Timor allows system designers and/or programmers both to obtain an overview of how modules can interact with each other and to concentrate on the implementation(s) of an individual module.

The design of a system can begin by specifying the purposes of individual modules and their functionality in type definitions of major information hiding modules which comprise a system. The designer(s) need not be concerned with individual implementation decisions and can in this way carry out walk-throughs of system activities before (or in parallel with) the programming of individual modules. On the other hand individual modules can be implemented and tested without reference to their interactions with other modules except via their procedural interfaces, thanks to their strict adherence to the information hiding principle. These are some of the advantages of a strict adherence to the information hiding principle and of a strict separation of type definitions from their implementations.

For the same reasons Timor can be considered a component-oriented language, since the components of systems and the components of the individual programs in a system can be separately programmed and implemented in different ways. In this way Timor can be used as a basis for developing a software industry for software components, just as in other industries (e.g. the car industry) components can be individually mass-produced and sold, as Doug McIlroy already envisaged in the early days of computing [7].

In the following chapters the above concepts are explained in detail. But we begin with some of the more mundane concepts in Timor.

WARNING: The published papers which describe various aspects of Timor were prepared before the definition of the language in this book was completed. Readers are therefore warned that some details in these papers do not reflect the latest definition of the language.

Chapter 2
Control Structures

Many of the statement constructs in Timor are similar to Java or C++ statements, and will be immediately obvious to the reader in the examples, but there are some exceptions, as now described:

1 Iteration Statements

The following forms of repetition are supported:

```
while Boolean expression {...}
repeat {...} [until Boolean expression]
```

The **`until`** clause is optional. If it is omitted, the statements following the keyword **`repeat`** are executed "for ever". (In this form it can be used to execute some ModelOS processes).

```
for dummy-variable in collection expression {...}
[else {...}]
```

The optional **`else`** clause is executed if `collection expression` is an empty set. The **`for`** statement is explained in more detail in Chapter 13 sections 4.7 and 4.8.

2 Conditional Statements

The **`if`** conditional statement supports **`elsif`** and **`else`** as follows:

```
if Boolean condition {...}
[elsif Boolean condition {...}]
[elsif Boolean condition {...}]
etc.
[else {...}]
```

A **`case`** statement is also supported, e.g.

```
case (x) of { // where x is a control variable
  (value1) {statements} // and value1 etc. are values
  (value2) {statements} // of the type of the variable
  ...
 [else {...}]
 }
```

The statements following the **`else`** clause are optionally executed if none of the listed values matches the value of the control variable.

There is also a **`cast`** statement, which is explained in chapter 7 section 9 and a **`cocast`** statement which is explained in chapter 16 section 2.3.

3 `With` Blocks

Timor supports a "`with`" block which enables the internal identifiers of a variable to be accessed without repeating the variable name. It has the following form.

```
with (variable) [as identifier] {...}
```

This was inspired by the Pascal "with" statement, but has been adapted to the needs of Timor. The aim is to make code less tedious to write and easier to understand. It is particularly useful in code which overrides methods of re-use variables, but is not confined to this.

Unlike Pascal only a single variable can appear following the **`with`** keyword. The optional **`as`** clause allows the nominated variable to be "renamed" by an identifier. The reason for these changes is to make programs more intelligible.

A **`with`** block is interpreted entirely statically by the compiler when compiling an implementation; it plays no part whatsoever in the dynamic execution of a program. Its sole purpose is to resolve shortened identifiers which appear in the block.

3.1 What can be Nominated in a Timor Block

In Timor any variable identifier, including re-use variable identifiers, can be nominated as a **`with`** variable.

Here is an example, which we used in the paper "Inheriting Multiple and Repeated Parts in Timor" [11]:

```
state:
 ^ArrayQueue1 aq; // a re-use variable
instance:
 op void insertAtFront(ELEMENT e) throws FullEx {
  with (aq) {
   if (size < maxSize)
```

```
    {front--; if (front < 0) front = maxSize - 1;
     theArray[front] = e; size++;}
    else throw new FullEx();
   }
  }
```

This is equivalent to

```
 state:
  ^ArrayQueue1 aq;
 instance: // the methods not coded in ArrayQueue1
  op void insertAtFront(ELEMENT e) throws FullEx {
   if (aq.size < aq.maxSize)
    {aq.front--; if (aq.front < 0) aq.front = aq.maxSize - 1;
     aq.theArray[aq.front] = e; aq.size++;}
   else throw new FullEx();
 }
```

with blocks can be nested within each other, which might be useful in special cases.

3.2 Interpreting Identifiers in a With Block

In the first instance each identifier which is used in a **with** block is treated as if it were a complete identifier (i.e. as if it were not in a **with** block). This means that it is possible explicitly to use the **with** variable identifier itself within a **with** block which nominates it. (This can be important, for example, if a method has to be called recursively, and within it there is a **with** block where the **with** variable has a method with the same identifier.)

If an identifier remains unresolved, the compiler prefixes it with the identifier of the **with** variable of the **with** block in which it directly appears, i.e. the innermost with block at that point in the text (separated by a dot), and attempts again to resolve it. If it is still unresolved, the compiler replaces the prefix of the innermost block with the identifier of the **with** variable of the surrounding with block and tries to resolve it. This process is repeated until either the identifier is resolved (or it cannot be resolved by the outermost with variable, in which case the compiler raises a compile time error).

A **with** block can itself contain a variable name using the dot notation, e.g.

```
 with (a.b) {...}
```

In this case identifiers in the code are where appropriate prefixed by `a.b.`, but not by `a.` or by `b.` alone!

3.3 The Scope of a With Block

The construct has been deliberately called it a *block*, rather than a *statement*, because it can have a wider scope than a statement. For example, if the **`with`** variable is a state variable, the **`with`** block can (but need not) be placed directly after the state section in which it is declared, and can, for example, continue until the end of the implementation. However, its terminating bracket cannot be placed where it would cause the compiler to confuse this with some other terminating curly bracket. A **`with`** block can of course also be placed anywhere where a normal statement is possible, but not nested within a conditional statement.

This allows several methods (even all methods) of an implementation to be enclosed in a single **`with`** block, as is illustrated in the implementation of `OrderedSet::ArrayImpl` (see chapter 13 section 2.3.4).

4 Exception Handling

All exceptions are unchecked, in the sense of Java unchecked exceptions.

A method in a type definition must list in its heading all the exceptions which it explicitly throws, using the keyword **`throws`**. It can, but need not, define other exceptions (e.g. those of the methods which it invokes[7] or arithmetic and other typical run-time exceptions).

New exceptions can be added in a subtype (or a qualifying type).

Any exception which is not specifically defined as a type is a subtype of the general type 'Exception', which has no methods.

An exception can be defined in a **`throws`** clause. This is indicated in the heading of the method. The actual handling takes place in the **`catch`** section of a Java-like **`try-catch`** block. This takes the form

```
try {...} // the statements to be tested
catch (ExceptionType exceptionId) {exception handling code}
... further catch blocks to handle different exceptions
[finally {optional code executed regardless of result, e.g.
to clean up before exiting}]
```

The **`catch`** block can specify multiple exceptions types which use the same exception handling code. These are separated by a bar, e.g.

```
catch (ExceptType1 | ExceptType2 | ExceptType3 excId)
{exception handling code}
```

[7] The information hiding principle implies that at the level of a type definition the designer does not know what components will be used and therefore cannot be sure which exceptions can be thrown!

If the method does not handle an exception, this is passed to its calling method. If the module's semantic routine (i.e. the module's outermost routine which was invoked to enter the module) does not handle an exception, this is passed to the ModelOS exception handler.

5 Further Syntax

Further kinds of statements are described in connection with the structures for which they are used, e.g. **`case`** statements in connection with enumeration types.

Attention is also drawn to the existence of additional syntax for programming with collections, see chapter 13 section 4.

A list of operators is defined in Appendix II.

6 Pragmas

Pragmas are not part of the Timor language, but are compiler directives (or advice) aimed at improving the efficiency of compiled programs; there is no defined list of pragmas for the language but pragmas can be defined for particular Timor compilers. Each compiler can support its own list of pragmas. The only Timor extension is that a pragma is introduced into the text of programs on a separate line which begins with the text `#pragma`.

Chapter 3
The Basic Structure of Timor Programs

The OO idea, as it appears in conventional programming languages, suffers from the restriction that it is applied only to small objects within a program. The result is that conventional OO (and other) languages faithfully reflect the conventional but harmful[8] idea of separating

- *programs*, which typically have a single entry point (parameterless in the sense that routines have parameters) and only temporary data, from
- *persistent data* (in the form of files organised by a file system).

According to that model access to persistent data files is handled by special interfaces to the file system. This approach reflects the dependence of programming languages on conventional operating system structures.

ModelOS and Timor rectify this deficiency by taking a uniform approach to the definition and implementation of both small objects within a program and large objects which form major modules known to the operating system. On the other hand the Timor approach does not prevent the language from being used with conventional operating systems, since access to files in a conventional file system can be hidden behind a Timor module interface, and a single entry point information hiding module can be used to start a program.

1 Identifiers

The following rules are defined for identifiers.

- A type or view identifier starts with a capital letter and may be followed by

[8] This approach is harmful in the sense that it opens up a simple way for any hacker to write programs which can relatively easily access files that are inadequately protected by file systems. It is also harmful in that it discourages the software engineering idea of developing major modules as information hiding modules.

a combination of small and capital letters, including at least one small letter. Other symbols are permitted as normal, except as follows.

- The ampersand character (&) is permitted only in co-type identifiers.
- The double colon (::) is permitted only in implementation identifiers.
- A single capital letter is a type identifier.
- A variable or method identifier begins with a small letter and may be followed by any combination of small and capital letters. Other symbols (except ampersand and double colon) are permitted as normal.
- An identifier for a co-type variable consists of the corresponding type name followed by the & symbol and a co-type suffix.
- A generic identifier (regardless which kind) must begin with at least two capital letters. Small letters are not permitted in a generic identifier but additional further symbols are permitted as normal.

The scope of type, co-type and implementation identifiers (including enumeration types) is the module in which they are used.

2 Enumeration Types

Timor supports enumeration types, in three forms:

a) simple enumerations, using the keyword **`enum`**;

b) sequences, using the keyword **`seq`**;

c) circular types, using the keyword **`circ`**.

2.1 Simple Enumeration Types

An enumeration is declared as

```
enum Colour {red, blue, green, yellow, black, white}
```

Unlike C++ enumerations, the values are not equivalent to integers and cannot be coerced to integers (or any other type). Nor can a C-like "sizeof" operator be applied to them. The only applicable operators are assignment, equality (==) and inequality (!=).

An enumeration value can be used in a **`case`** statement, e.g.

```
case (thisColour) of {
  (red) {statement}
  (blue) {statement}
  [else statement]
}
```

Enumerations can have sub-ranges, e.g. `red..yellow`. These cannot be reversed, and no order is implied. However they can be used as array indices, e.g.

```
Drawer[] cottons = Drawer[green..white]::Impl();
```

Note on arrays: Timor's array indices also use sub-ranges of integers and other types can have literal sub-ranges, e.g.

```
Rainfall[] rains = Rainfall[2000..2011]::Impl();
```

2.2 Sequences

A sequence is declared as follows:

```
seq Numbers {one, two, three}
```

It differs from an enumeration in that its values have an ascending order, and can be compared with each other using all the normal comparators (<, <=, ==, `!=`, >, >=), e.g.

```
one <= three // returns true
three <= one // returns false
```

There are the following additional operators:

```
succ(one) // returns "two"
succ(three) // throws exception OutOfRange
pred(one) // throws exception OutOfRange
pred(three) // returns "two"
Numbers.range() // returns the integer value 3
Numbers.min() // returns one
Numbers.max() // returns three
Numbers.add(one, 2) // returns "three";
Numbers.add(one, 3) // throws OutOfRange
Numbers.diff(one, three) // returns -2;
Numbers.diff(three, one) // returns 2;
```

Sequences can have sub-ranges, e.g. `two..three` and these can be reversed, e.g. `three...two`. (Note: two dots for forward sub-range, three dots for reverse sub-range.) Both forward and reverse sub-ranges of sequences can be used to index arrays. They can be used in **`case`** statements.

2.3 Circular

A circular type is declared as

```
circ Days {monday, tuesday, wednesday, thursday, friday,
      saturday, sunday}
```

It differs from a sequence in that its values have a circular ordering. They can be compared with each other using all the normal comparators (<, <=, ==, `!=`, >, >=), whereby the result is similar to that which would arise for a sequence, e.g.

```
monday <= sunday // returns true
monday == succ(sunday) // returns true
monday > sunday // returns false
```

There are the following additional operators, which illustrate the circular nature of the types:

```
succ(monday) // returns "tuesday"
succ(sunday) // returns "monday"
pred(monday) // returns "sunday"
pred(sunday) // returns "saturday"
Days.range() // returns the integer value 7
Days.add(tuesday, 9) // returns "thursday"
Days.add(monday, -8) // returns "sunday"
Days.diff(sunday, monday) // returns 1
Days.diff(monday, sunday) // returns -1
Days.range() // returns 7
```

There are no `min` and `max` methods.

Circular types can have sub-ranges, e.g. `two..three` and these can be reversed, e.g. `three..two`. These can be used for example to index arrays. (Note: two dots always apply.)

2.4 Lists of Enumeration Values

A list of enumeration values can be created (as for other lists), by using the curly bracket notation `{}`. The type of the list is the enumeration type (followed by a semi-colon) and the values are separated by commas, e.g.

```
{Colour: red, green, red, blue, red}
```

This might for example appear in the following statement:

```
Set<Colour> colourset = {red, green, red, blue, red};
```

This is converted in the usual way to a set consisting of the values `red`, `green`, and `blue`.

3 Type Definitions

The definition of a Timor type, whether it is intended as a small object within a program or as an independent module organised by the operating system, has the following basic form:

```
type typeName {
instance:
instance method definitions /* These define the interface
                  methods (semantic routines) of the type */
protected:
protected method definitions  /* protected methods appear in
   type definitions but can only be called by implementations
   of derived types and co-types (see chapter 11) */
callback:
call back method definitions /* call back methods are
   instance methods which can only be called
   from a module which itself was invoked directly from
   the current module. */
}
```

The words and symbols marked in bold are fixed parts of the Timor language. The rest is supplied by the programmer.

Type definitions can be used to define both small and large objects (e.g. separate modules). The most significant point is that they are *type* definitions. In accordance with the information hiding principle, and in contrast with the conventional OO languages, it provides a natural way of allowing types to be defined and specified separately from their implementations.

This is the simplest form which a type definition can take. There are several possible variations on the basic form, reflecting further structural properties that a type may have. For example, as in standard OO, it is possible to define abstract types, in which case the keyword **abstract** precedes the keyword type. The following keywords can precede the keyword **type**.

a) **abstract:** indicates that an abstract type is being defined;

b) **singleton:** in a module only one instance of the type can be created;

c) **library**: the type can be implemented in ModelOS as a library module – see section 4.3 below and chapter 18 section 6 of [6];

d) **comod**: the type is implemented in ModelOS as a module which *can* accept and return reference parameters[9];

e) **callback**: the type must be implemented in ModelOS as a call-back module (see [6] chapter 18 section 9, chapter 20 section 8.5 and chapter 28 section 7).

4 Implementations

An implementation of a type has the following basic structure:

9 Normally the type definitions of independent modules (in the ModelOS sense) cannot include interface methods which pass and/or return parameters by reference, as is explained in detail in chapter 18 section 7 of [6].

```
impl implementationName {
state:
   /* internal data declarations which define the state
      variables of the module */
retained:
   /* data which allows a thread to retain information
      relating to a sequence of calls between an open call
      and a close call (see section 6 below). This data is not
      accessible to other threads. */
constr:
   // an implementation-specific constructor
instance:
   // instance method implementations
protected:
   /* protected methods appear in type definitions but can
      only be called by implementations of derived types (see
      chapters 6 to 8) and of co-types (see chapter 11) */
callback:
   // callback method implementations
internal:
   /* internal methods (which do not appear in type
      definitions and cannot be invoked from outside the
      module), i.e. subroutines */
}
```

The various sections (except the `constr` section) can appear more than once in an implementation and their order is irrelevant.

4.1 Retained Data

Retained data is an idea taken over from the ModelOS operating system (see chapter 18 section 1.4 of [6])[10]. The basic idea is that when a thread opens a module (e.g. a ModelOS module) the module can store information about that *open* call and the subsequent instance method calls which the thread makes to the module until it calls the *close* routine (see section 6 below).

4.2 Multiple Implementations

The same type can have several different implementations, regardless whether it implements small objects within a module (e.g. an employee record) or major modules (e.g. a file of employee records). Since the interface is defined entirely

10 which was in turn taken over from MONADS.

in procedural terms, users of objects of the type do not have to be concerned about the data structures which appear in the **`state`** and **`retained`** sections. In fact these may differ in different implementations of the same type. All the implementations of a type must produce the "same" results.

An implementation name is the name of the type being implemented followed by a double colon (`::`) then an implementation identifier. Each concrete type must have at least one implementation, with the suffix "`::Impl`". This is the name used for default implementations, which are automatically selected when another implementation name is not explicitly provided.

An explicitly programmed constructor can be omitted if the variables in the state section are all explicitly initialised or have default values[11]. In this case the compiler creates a parameterless constructor, which can be invoked in the usual way, e.g. `typename::Impl()` or in the case of an explicitly defined implementation using the implementation name, e.g. `Stack::LinkedStack()`.

Since different implementations of a type may require constructors with different parameters, constructors do not appear in type definitions, but only in implementations.

A type can be implemented in several ways [9]:

1) It can be complete self-contained, i.e. it is freshly coded independently of all other implementations. This remains true whether the type is a base type or is derived by inheritance from some other type.
2) An implementation can re-use code from some other implementation which has matching methods (regardless whether or not the types stand in an inheritance relationship).
3) It is also possible to create implementations which are typeless and to use these in implementations of types. Such free-standing typeless implementations have no significance for the type system. A typeless implementation is identified by a double colon (`::`) then an implementation identifier (without a preceding type name), e.g. `::usefulCode`.

These possibilities will be discussed in more detail in chapter 8.

5 Instance Methods

An instance method is a method which operates on a single instance of a type. In contrast with more conventional OO languages, a Timor type only has instance methods (and possibly *open* and *close* methods, see next section). It will be ex-

11 Default values for state variables in Timor are similar to those in Java, i.e. for numbers 0, for booleans `false` and for references and capabilities `null`. This should not be confused with default parameter values, see chapter 3 section 8.

plained in chapter 11 how other kinds of methods which appear in conventional OO languages (e.g. static methods, binary methods and application oriented constructors) are supported.

The instance methods which appear in a type definition specify the interface routines of the type[12]. There are two kinds of instance methods.

- *Enquiries*, which are indicated by the keyword **enq**, normally[13] return a value to the caller but are not allowed to modify the state data associated with the object on which they operate[14].
- *Operations*, which are indicated by the keyword **op**, may but need not return a value to the caller. They modify the state data associated with the object on which they operate.

This distinction is important for ModelOS, since it allows compilers and programmers to distinguish between reader methods (**enq**) and writer methods (**op**). This greatly simplifies reader-writer and more advanced forms of synchronisation, and it also allows users to specify their protection requirements more flexibly (see chapter 15 section 2).

6 Open and Close Methods

In addition to the general methods (operations and enquiries) Timor supports two special methods (*open* and *close*). As in conventional operating systems and file systems an *open* method signifies that a thread intends to use a module via a number of method invocations and *close* signifies that the sequence of method calls has now come to completion. Not all types have open and close methods, and in fact such routines can be added to type descriptions via a separate add-on mechanism (see Chapter 10 section 5). They are regarded neither as operations nor as enquiries. They are identified by the reserved names `open` and `close`. The first (and possibly only) parameter of `open` allows a thread to specify whether it intends to access the object being opened in read mode or write mode.

```
enum OpenMode {closed, read, write}
open void open(OpenMode mode) throws OpenError;
close void close() throws CloseError;
```

[12] Individual implementations of a type can also have internal instance methods, which can vary from implementation to implementation.

[13] There are cases where an enquiry returns a `void` value (see e.g. the method `sublist` chapter 14 section 4.2).

[14] Sometimes programmers include auxiliary features in enquiries (e.g. variables which count the number of calls). In Timor such features should be provided in qualifiers (see chapter 10), thus ensuring that methods which logically are enquiries do not contain write operations. This is important to allow the methods to be properly categorised for synchronisation and security reasons, as will become evident later.

Both methods can have further application specific parameters. As the above example shows, these methods are indicated by the keywords **`open`** and **`close`**, which are reserved for this purpose. The names of these methods follow the normal rules, but in contrast with other methods they may (but need not) be called `open` and `close` respectively.

Apart from the synchronisation aspect, these methods are useful in ModelOS, e.g. for determining whether a removable disc can safely be dismounted, and they play a role in iterating through a collection, as is explained in Chapter 13 section 4.8.

7 Callback Methods

These are instance methods (which are defined in a separate **`callback`** section of type and implementation definitions. When a method of a call back module A invokes a method of a further module B, this method can invoke a call back method of A by using the pseudo-variable **`callback`**, e.g.

```
callback.aMethod(params);
```

8 Parameters

8.1 Parameter Declarations

As in other languages, methods have parameters. In most languages individual parameters in method *declarations* are separated by commas. By contrast, method parameters in Timor are treated syntactically as a set of data declarations similar to those in other sections, e.g. in state sections, and are therefore separated via semicolons.

```
enq Boolean equal (Int i1; Int i2);
```

However, as in a Timor state section, a number of parameters of the same type can be separated by commas following the type name, e.g.

```
enq Boolean equal (Int i1, i2);
```

In his case the individual parameters must have *exactly* the same type and exactly the same mode[15].

Method *invocations* separate parameters using commas, as in other languages.

8.2 Default Parameter Values

It is possible in Timor method declarations to provide parameters with default parameter values, which can of course be overridden in actual method invoca-

[15] Modes are described in chapter 4.

tions. In this case each declared parameter which has a default value must be separated by a semicolon from other parameters, e.g.

```
enq Boolean equal (Int i1 = 20; Int i2);
// i1 has a default value; i2 has no default value
```

In this case `i1` has a default parameter value 20, which must also be of type `Int`. This can be overridden in an actual call, e.g.

```
Boolean b = equal (n, m); // m and n must be of type Int
```

To make use of a default parameter value the programmer replaces it with an asterisk, e.g.

```
Boolean b = equal (*, m);
```

9 Protected Methods

These are instance methods which are declared in type definitions, but which cannot be called by normal clients. They can only be invoked by the implementations of derived types and of co-types (see chapter 11).

10 Internal Methods

Internal instance methods can appear in an **`internal`** section of an implementation. These are methods which are not visible to and cannot be called by clients of the module. Such methods do not appear in a type definition, but appear only in implementations. Each implementation of a type can have different internal methods.

11 An Example Type Definition

Here is an example of a simple type which defines stack instances that hold integers[16]:

```
type Stack {
instance:
  op void push(Int i) throws StackFull;
       /* puts an integer on the stack top
          or signals that the stack is full */
  op Int pop() throws StackEmpty;
       /* removes and returns the integer at the stack top
          or signals that the stack is empty */
  enq Int top() throws StackEmpty;
       /* returns the integer value at the stack top
          without modifying the stack */
```

[16] This should not to be confused with the ModelOS kernel's thread stack.

```
    enq Int length();
         /* returns an integer indicating
            the current stack length */
    enq Boolean contains(Int i);
        /* indicates whether the integer i
           is currently on the stack */
  protected:
    enq Int getEntryAtPos(Int position)
                       throws Invalid Param;
       // position 0 signifies first position
  }
```

Notes:

1) Type names always begin with an upper case letter followed by a non-uppercase character; this includes the basic types such as `Int` and `Boolean`.
2) The method `push` has an integer parameter which is referred to in implementations of the method as `i`. It returns a `void` result.
3) The method `pop` has no parameters but returns an integer as its result.
4) The first two methods are defined as operations (**op**) because they change the state data to carry out their tasks.
5) The remaining methods are defined as enquiries (**enq**) because their tasks do not require them to change the state data of implementations.
6) The users of stack objects are unaware of the data structures used in implementations.
7) The example illustrates a stack of integers. Timor has a generic facility, which can be used to define container types (including stacks) that hold elements of any types, defined generically[17]. At this tutorial stage we prefer not to introduce the additional features needed to support genericity; these are explained in chapter 12.
8) The protected method can be used by a co-type (to be discussed in chapter 11) to gain efficient access, e.g. for a copy operation.

12 An Example Implementation

Here is a possible implementation of the type `Stack`.

[17] Not to be confused with containers in ModelOS. In Timor a container type is a type which stores instances of other types. In addition to stacks (illustrated here) different container types can be defined, e.g. lists, sets, tables, queues. They are more fully discussed in chapters 12 to 14.

```
impl Stack::ArrayStack {
state:
   Int[] stack = null; // an array of integers
   Int maxlength, length = 0;
constr: /* constructs the array using a Timor array
           constructor */
   Stack::ArrayStack(Int maxlength) throws BoundsEx {
    if (maxlength < 1 || maxlength > 1000000)
                            throw new BoundsEx();
    this.maxlength = maxlength;
    stack = Int[]::Impl(maxlength); // an array constructor
   }
instance:
  op void push(Int i) {
   if (length == maxlength) throw new StackFull();
   stack[length] = i;
   length++;
  }
  op Int pop() {
   if (length == 0) throw new StackEmpty();
   length--;
   return stack[length];
  }
  enq Int top() {
   if (length == 0) throw new StackEmpty();
   return stack[length-1];
  }
  enq Int length() {return length;}
  enq Boolean contains(Int i) {
   for (next in {0..(length-1)})
    if (stack[next] == i) return true;
   return false;
  }
protected:
  enq Int getEntryAtPos(Int position) {
   if (position < 0 || position ≥ length)
                    throw new InvalidParam();
   return stack[position];
  }
}
```

Notes:

1) The **`state`** section contains data declarations which appear in each instance of the type being implemented as an `ArrayStack`.
2) Constructors are not defined in type definitions because they may have different implementation-dependent parameters. The name of an implementation constructor is the same as that of the implementation.
3) There is only one constructor per implementation. In chapter 11 it is shown why this is not a limitation compared with other OO languages.

Chapter 4
Instances, Values, Objects and Modules

When an implementation of a type is compiled, the compiler creates a pattern for an *instance record*, which contains an internal form of the state declarations and an internal pointer to the corresponding list of instance methods. When a constructor is invoked, this creates and initialises an actual instance record.

Some OO languages, such as C++, draw a distinction between values and pointers to values. Others, including Java, have avoided the explicit use of pointers, presumably with the well-intentioned aim of eliminating certain potentially harmful programming tricks, such as allowing arithmetic operations to be used to change pointers. Unfortunately this also removes from the programmer the ability to determine precisely when underlying pointers should be used, which is obviously disadvantageous in the ModelOS context. Consequently Timor explicitly promotes the ability to distinguish between pointers and values.

The C notation for pointers has been partially re-introduced in Timor, but C and C++ programmers are warned that many of the "features" of pointers in those languages have not been included, especially the ability to carry out pointer arithmetic. In Timor, pointers are usually called *references*, which can be implemented as ModelOS pointers.

Timor also introduces a notation for *capabilities*, which are pointers to major modules known at the operating system level[18]. These correspond to ModelOS capabilities. This eliminates the need for special constructs to access persistent files, necessary in other programming languages.

[18] In ModelOS a capability is a protected data structure which contains a unique module identifier (for modules known to ModelOS, which are equivalent to persistent files in conventional file systems) and an extensive set of access rights and restrictions. Capabilities can only be accessed via ModelOS privileged kernel instructions.

The instance record produced by a constructor represents a *value* for that type. This can, for example, be assigned to a *value variable*, or it can be transformed into an *object*, which can then be assigned to a *reference variable*, or it can be transformed into a *persistent file*, which can be assigned to a *capability variable*.

1 The Basic Types

The basic types (e.g. integers, booleans, characters) are similar to those in Java. These form the basis for building structured types and are normally provided by the computer architecture. Wherever possible, basic types and structured types are handled uniformly in Timor. Thus in contrast to most other languages they have type names which begin with a capital letter, e.g. `Int`, `Boolean`. As will be seen later, they can be statically instantiated in the normal way, but they can also be dynamically instantiated as (shareable) objects accessible via references (see sections 2 and 3 below).

Although a high degree of uniformity exists in Timor for the handling of the basic types and structured types, there are some inevitable small differences, which arise from the fact that basic types represent the level of "real" implementation, i.e. they provide the starting point for defining and implementing structured types. These differences can be summarised as follows:

a) There is only a single (implicit) implementation for a basic type, whereas a structured type can have more than one implementation.

b) A basic type has no normal instance methods. Its operations all fall formally into the Timor categories *binary* or *constructor* (including constructors such as the negation of an integer, or binaries which sum of two integers).

c) Basic types always have a valid value by default, whereas structured types have a special value `null`.

There is a special type `Array`, which is also similar to a Java array. This is not primarily intended for direct use in programs (though it can be so used). Its primary purpose is to provide an implementation of the Timor Collection Library (see chapter 13 section 2.2), which can then be used to provide an equivalent of arrays (see chapter 13 section 4.3, 'Selection by Position').

2 Value Variables

A value variable is declared as a type name followed by an identifier for the variable. Its value can be initialised by assigning a value of the same type to it, which in the case of the built-in types is often a literal value, e.g.

```
Int i = 3;
String s = "This is a string";
```

In the case of user-defined types a constructor defined in an implementation of the type can be called, e.g.

```
Person p = Person::Impl();
```

or an existing value of the type might be assigned to it, e.g.

```
Person p2 = p; // This is a copy operation
```

Different implementations of the same type can co-exist in the same program.

3 References and Objects

A fundamental difference between a value and an object is that values are not shareable, i.e. a value can only be addressed (in accordance with the information hiding principle) by the methods of an implementation of the type in which the value variable is defined when operating on the appropriate instance. In contrast, objects are shareable if they can be reached via an appropriate reference variable. Reference variables are stored as pointers[19] in the state section of an instance record.

A reference variable is declared using a type name with a single asterisk as a suffix, followed by an identifier for the reference, e.g.

```
Person* spouse;
```

References point to objects of the appr**op**riate type. In order to create an object (as distinct from a value within an object) the operator **new** is used, e.g.

```
Person* spouse = new Person::Impl();
```

The **new** operator is followed by a value of the appropriate type. The example shows how this value might be created by a constructor, but an existing value of the appropriate type can also be used to create an object, e.g.

```
Date* dob = new today;
```

A reference assignment, i.e. the assignment of a reference variable to another, causes both references to point to the same object[20], e.g.

```
Person* myBrother = new Person::Impl();
Person* hisCousin = myBrother;
```

The value of an object can be retrieved as a value by using the dereferencing operator, which creates a copy of the content of the object:

```
Person personDetails = *myBrother;
```

[19] References are in fact indirect. A reference selects an entry in a Timor (run-time) object table, which in turn points to the object in question. The indirection is not visible in Timor. In this way objects can be shared and can be relocated and deleted without causing problems. For more details see Appendix I.

[20] in the (hidden) object table.

The built-in types can also be created as objects, e.g.

```
Int* intObject = new Int::Impl(5);
```

or more simply

```
Int* intObject = new 5;
```

Both these declarations create an integer object (with a value 5), which can be shared by other objects.

There is an explicit **`delete`** statement for deleting objects, e.g.

```
delete (hisCousin);
```

This causes the referenced object to be deleted[21]. Subsequent attempts to access the object by any reference cause an exception.

If an object becomes unreachable, it can be deleted by a garbage collector.

4 Capabilities and File Modules

When Timor is used in the ModelOS environment it does not require a special interface to give programs access to a conventional file system. Instead it must take into account the difference between types/implementations which serve as independent ModelOS modules (see chapter 1 section 1) and objects which are internal structures within a ModelOS module. This is achieved in Timor via features to support ModelOS capabilities, which are protected ModelOS data structures that provide access to (and protection for) ModelOS modules.

Just as reference variables point to the objects within a module, so capabilities point to file modules[22]. The notation used to signify a capability variable for a file module is a type name followed by two asterisks, e.g.

```
TextFile** myFile;
```

Just as **`new`** is used to create internal objects, so the keyword **`create`** is used to create a new file module, e.g.

```
TextFile** myFile = create TextFile::Impl();
```

Similarly the other operations involving objects described above apply analogously to capabilities, though it is unlikely that these will be used frequently. For example a new module can be created by nominating an existing value or a literal, e.g.

```
TextFile myFile = TextFile::Impl();
```

[21] In contrast with some OO languages, explicit deletion is supported, because Timor *inter alia* aims to be a realistic language for database applications.

[22] In contrast with objects, there is no "capability table" within a module. The operating system manages capabilities, which can be passed to the instance methods of a module as parameters and can be stored internally in capability variables.

```
// This creates an instance record
myFile.insert("This is my file");
// This inserts text into the instance record
TextFile** hisFile = create myFile;
/* This creates a file from the instance record and assigns
   it to a capability */
```

A capability assignment copies a capability, e.g.

```
TextFile** myFile = create TextFile::Impl();
// This creates a file and assigns it to a capability
Textfile** anotherCapability = myFile;
// This is a capability copy operation
```

Copying a capability does not create a new module, but simply a new capability for the same module. (In ModelOS systems this operation is of course only permitted if the "copy" right is set in the capability to be copied.[23]) Similarly, a file can be deleted using the **`delete`** operator, but only if the rights in the capability allow this.

Violations of the rights in capabilities result in exceptions which are handles as ModelOS exceptions.

The different possibilities for organising instances as values, references and capabilities are known as different *modes*.

5 Library Modules

Library modules are usually considered to be modules which provide commonly used algorithms and data structures. They are sometimes implemented in ModelOS as separate modules which can be used independently of other modules. For example a library of trigonometrical functions can be implemented in this way. Such modules should not be designated as **`library`** types in Timor.

Other library modules, such as synchronisation modules (see chapter 15 section 4) are closely related to the data of application modules and it would be extremely inefficient to treat these as normal modules which are accessed via inter-module calls (which in ModelOS are the main mechanism for implementing protection, and are consequently quite slow when compared with calls between the objects in a single module). Consequently these are designated in type definitions as **`library`** modules and the compiler can treat these almost like other internal data structures, with a root data segment which is integrated into the other data structures of a program. How this happens in detail in ModelOS is explained in [6] chapter 18 section 6. A further reason for this is that unlike calls

[23] For more details about copying capabilities see [6] vol.2 chapter 26 section 3.3.

to independent ModelOS modules, such library modules can have references as parameters and return values.

There is a third group of library routines which can both act as separate modules and can be integrated into other modules. This applies especially in the case of collection modules. These can conveniently serve as data structures within an application (e.g. for temporary arrays, queues, etc., as supported in OO languages) but also as persistent separate file modules (equivalent to file system files in conventional operating systems. From the Timor language definition viewpoint there is no significant difference (provided that they follow the rule that their interface methods do not pass or return references). But because ModelOS distinguishes the two cases the Timor compiler must provide a parameter which indicates whether such a module is to be implemented as a separate module or as a unit integrated into some other module.

6 Conversion between Modes

Mode coercion (i.e. implicit conversion between modes) is not supported, but casting between modes is supported using a syntax similar to the Java type cast syntax, i.e. by placing the converted mode name (in brackets) before the item to be casted, e.g. `Person* p = (reference) x`, where x might be a value variable. In this trivial example it would have been simpler to use the keyword **`new`** to achieve the same effect if it were clear that x is a value, but if it were possible that x were already a reference, that would cause an error. As we shall see in chapter 7 section 8 this situation can arise when Timor "handles" are used. To convert to a value the syntax is `(value)`; to convert to a reference the syntax is `(reference)`, to convert to a capability the syntax is `(capability)`[24].

7 Numerical Representation

Timor uses the same rules as Java for representing and casting numbers.[25]

8 Instance Records

Compilers implement values, object references and capabilities using an instance record which contains pointers into a heap. An instance record also contains a pointer to the list of methods of the implementation. This organisation is described in more detail in Appendix I.

9 Shared Objects and Collections

Section 3 described how objects are reached via references with the beneficial

[24] The ModelOS access rights in converting to a capability are all set to true.

[25] see for example http://faculty.salina.k-state.edu/tmertz/Java/041datatypesandoperators/07typecoercionandconversion.pdf

consequence that they become shareable (in contrast with values). This opens up the possibility that the same objects can appear in several object collections. This is taken into account in the Timor Collection Library (TCL), which allows a collection to be copied using the operator =* ('reference copy')[26]. In this case programmers must of course ensure that access is synchronised where appropriate.

The effect of the reference copy operator is that the operation copies only the references which refer to the elements in the collection. References within the elements themselves are not affected by the operation. One advantage of this is that the copy operation is faster than it would otherwise be. When such an object is deleted via a shared collection, only the reference is deleted. Similarly when the shared collection is deleted, this does not affect the original collection. However, if the original collection or an element (object) in it is deleted this action is taken as if the collection is not shared, with the consequence that errors can occur when accessing the objects via a shared version. Since the aim of this mechanism is to speed up temporary activities (such as taking an intersection of two object collections) this restriction is not considered to be problematic

[26] see chapter 13 section 4.5.

Chapter 5 Abstract Variables and Records

Timor types strictly follow the information hiding principle by not permitting "raw" data declarations (i.e. fields) to appear in interface definitions. Only methods are permitted. This is important to ensure that synchronisation and protection can be organised correctly.

However, this can be extremely inconvenient for programmers, especially when "small" types are being defined. Consequently Timor allows some methods to be defined in a type interface as if they were variables. This simplifies programming, but in reality the compiler treats each such variable declaration as a pair of methods, often known as "setters" and "getters". One method is a writer method which apparently sets (modifies) the value of a (hidden) state variable, while the second, an enquiry, returns the value of the (hidden) state variable to the caller. (In implementations of the type there is no compulsion for an implementation programmer actually to declare such hidden variables in the state section; he can implement the methods however he wishes, provided that the implementation fulfils the specification.) In this way the application programmer's task is simplified, the interface is easy to understand, but above all, justice is done to the information hiding principle, allowing synchronisation and protection to be organised in an orderly and straightforward manner.

For example, in the **`instance`** section of a type definition a programmer might make the following abstract variable declaration:

```
Date dateOfBirth;
```

This is equivalent to the following method pair[27]:

[27] As is common in OO languages, Timor allow the overloading of method names.

```
final op Date dateOfBirth(Date dateOfBirth);
final enq Date dateOfBirth();
```

The modifier `final`, when associated with a method, means that the method in question cannot be overridden in a subtype (to be discussed below). The user of an object can access it in the familiar OO style, e.g.

```
if (mySon.dateOfBirth.day == 7) ...;
```

The compiler converts this into method calls, e.g.

```
if (mySon.dateOfBirth().day() == 7) ...;
```

An *abstract variable* declaration can be modified by the keyword `final`, in which case there is only an `enq` method.

1 Standard Implementations of Abstract Variables

The standard implementation for abstract values has the following pattern:

```
state:
 Date dateOfBirth;
 ...
instance:
 final op Date dateOfBirth(Date dateOfBirth)
  {return this.dateOfBirth = dateOfBirth;}
 final enq Date dateOfBirth()
  {return dateOfBirth;}
 ...
```

In order to take advantage of this, the programmer simply declares the state variable; the compiler then automatically adds the implementation.

The same technique can be used for references and capabilities. Thus, if a type definition includes the declaration of a reference, say, to another `Person`, e.g.

```
Person* spouse;
```

this corresponds to two methods

```
final op Person* spouse(Person* spouse);
final enq Person* spouse();
```

with standard implementations

```
state:
 Person* spouse;
 ...
instance:
 final op Person* spouse(Person* spouse)
```

```
    {return this.spouse = spouse;}
   final enq Person* spouse()
    {return spouse;}
   ...
```

Similarly, if the `Person` definition includes a capability for a related file module, e.g.

```
  Document** birthCertificate;
```

this corresponds to two methods

```
  final op Document** birthCertificate
                                   (Document** birthCertificate);
  final enq Document** birthCertificate();
```

with standard implementations

```
  state:
   Document** birthCertificate;
   ...
  instance:
   final op Document** birthCertificate
                                   (Document** birthCertificate)
    {return this.birthCertificate = birthCertificate;}
   final enq Document** birthCertificate()
    {return birthCertificate;}
   ...
```

2 Records

Often for programming in the small, a type definition may consist entirely of abstract variables; this is often called a "record", e.g.

```
  type Person {
  instance:
   String name;
   String address;
   Date dateOfBirth;
   Person* spouse;
   Document** birthCertificate;
  }
```

In this case the compiler produces a standard implementation of the entire type with the implementation name `<typename>::Impl` (here `Person::Impl`). If abstract variables are not explicitly initialised the default methods are provided automatically.

Hence abstract variables and records comply with the information hiding principle without burdening the programmer with the work of declaring setter and getter methods.

3 Using the Methods

Like other OO languages, Timor uses the dot notation to call the methods associated with an instance of a type, i.e. `<instance>.<method_call>`. The instance can be a value, a reference or a capability which is currently reachable.

Thus to initialise the name of Peter's spouse, where Peter is a `Person` object declared as follows

```
Person* peter = new Person::Impl();
...
```

the following statement would be formally correct:

```
peter.spouse().name("Mary");
```

The nominated instance is <`peter.spouse()`> which is a method call to the enquiry `spouse` in the instance record for `peter`. This returns a reference which is then used as an instance to call the operation `name` associated with the instance record for Peter's spouse; this sets the state variable called `name` to the string `"Mary"`.

Calling methods associated with abstract variables can be simplified so that it appears if they could be accessed as variables, i.e.

```
peter.spouse.name = "Mary";
```

The compiler converts this back to method calls. This is necessary to ensure that synchronisation and protection work correctly.

4 Final and Constant Values

Timor allows value variables to be defined as **`final`** or **`const`**. A **`final`** variable is one which must be initialised as part of the initialisation of the instance in which it is embedded, and thereafter it cannot be reinitialised. However, all the methods of its component instances can be invoked, as with final Java variables. An interesting effect of **`final`** components is that the implementation cannot be changed at run-time, potentially allowing a compiler to make optimisations.

A **`const`** variable is defined such that once it has been initialised neither it nor its variables (recursively) can be modified after they have been initialised. This implies that it and its variables must be fully initialised by its constructor[28]. Thereafter its value can be read but not modified. The effect of declaring a vari-

[28] This statement is modified in chapter 11 section 2.

able as **`const`** is that only its **`enq`** methods and the **`enq`** methods of it variables may later be invoked.

Attempts to violate the limitations associated with a **`final`** variable (i.e. attempts to assign a new value to it) and with a **`const`** variable (i.e. attempts to assign a new value to it or to invoke **`op`** methods on it) can be detected at compile time.

5 Fixed and Constant References

A reference which has a fixed value (i.e. which cannot be changed) is declared using the modifier **`fixed`**, e.g.

```
fixed Person* spouse = new Person.init();
```

Such a reference must be initialised at the latest by the constructor for the implementation in which it is embedded[29]. Thereafter it cannot be changed. However, this is orthogonal to the issue whether content of the object to which it refers can be changed. The latter is determined by the modifier **`const`** as follows.

Access to an object via a reference declared as **`const`** has the same effect as if the object were accessed via a variable declared as **`const`**, i.e. only the **`enq`** methods of the object and of its variables may be invoked via the reference. Because there is no dereferencing operator which would allow the value of a referenced object to be modified directly, there is no need for a **`final`** modifier (which would by analogy allow the object's instance methods to be invoked without restriction but prevent assignments to the object as such) for references.

29 See chapter 11 section 2, which describes how an exception can be implemented.

Chapter 6 Inheritance in Conventional Object Orientation

Inheritance is widely regarded as one of the key advantages of OO programming. It is concerned with two different but related concepts: subtyping and subclassing. The first enables programmers to take advantage of polymorphism in their programs; the second allows code to be re-used. Both are based on the idea of extending classes of objects. These two concepts are first discussed in a fairly general way in this chapter and then in further chapters the Timor approach is described. The following discussion by no means exhausts the theme of inheritance in conventional OO languages, but instead concentrates on the points which are significant for understanding Timor.

1 Subtyping and Subtype Polymorphism

Given a base type, e.g. `Person` as defined in chapter 5 section 2, this can be extended in different ways to model more specialist kinds of persons, e.g. a student, a tutor, a professor. The extended types are called subtypes (here subtypes of `Person)`. In each case further attributes[30] can be added. The result is that different subtypes of the same type (called the supertype) have some features in common (here those defined in `Person`) but they also have some additional features which are not shared. This is how a subtype `Student` might be defined in Timor:

```
type Student {
extends:
 Person;
instance:
```

[30] i.e. abstract variables and/or (other) methods (see chapter 5).

```
  String university;
  Int studentId;
  String faculty;
  Date commencement;
  String degree;
}
```

The **extends** clause indicates that the methods of `Person` are automatically included as methods of students. This commonality allows *polymorphism* to be introduced by allowing an instance of a subtype to be assigned to a variable (or parameter) of its supertype, e.g.

```
Person* aPerson;
Student* aStudent = new Student::Impl();
...
aPerson = aStudent;
```

Note:

The effect of the last line is that the reference `aPerson` points to the same object as the reference `aStudent`. In Timor a value assignment (i.e. a copy operation) causes the entire value of a subtype to be copied (in contrast with C++). Thus the statement

```
Person aPersonValue = *aStudent;
```

results in the value `aPersonValue` holding an entire `Student` value[31].

This has the advantage that a `Student` instance can now be treated as a `Person` in all contexts which expect `Person` instances. For example a `Student` instance can be passed as a parameter to methods which expect to receive a `Person` instance as a parameter, or it can be added to a list of `Person` instances, etc. This works because only the methods of `Person` can be applied to instances which are assigned to a variable declared as being of type `Person`. It does not work the other way around, because although every student, tutor, etc. is a person, not every person is a student, tutor etc.

The subtyping paradigm is not limited to two levels. It is possible for example to extend, say, a `Professor` (i.e. a `Person`) to a `ScienceProfessor` or an `EngineeringProfessor`, etc.

A subtype hierarchy may, but need not, contain abstract types (i.e. types which help modelling and polymorphism but which are never instantiated as real types). For example an abstract type `collection` might be extended to (i.e.

[31] The dereferencing operator * preceding the name of an object pointer or ** preceding the name of a capability) returns the value of the object or file.

more precisely defined as) concrete types such as `list`, `bag` and `set`, which have at least methods which are defined in `collection`, such as *insert element*, *remove element*, etc.

2 Diamond Inheritance

A particular issue arises when a type needs multiple sets of attributes. For example if a senior student is also a tutor, a type is needed which contains the attributes of a person together with additional attributes for a student and for a tutor. This is sometimes called diamond inheritance, because it is no longer simply hierarchical but requires a network structure that forms a diamond to express the idea graphically (see Figure 6.1).

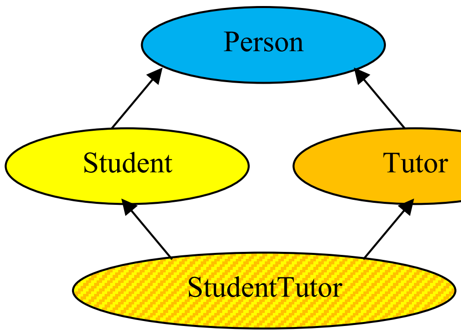

Figure 6.1: Diamond Inheritance

This is a special case of multiple inheritance (`StudentTutor` inherits some properties from `Student` and some from `Tutor`, but also – indirectly – properties from `Person`). In such cases subtyping is not always a particularly attractive modelling technique; the more subtypes are required, the more problematic the inheritance becomes. Consider that a person can be a male or female and can be a member of a library and several sports clubs, etc. at the same time as being a student-tutor. To model such complexity (which may well be required in a database) by means of conventional subtyping, is not always a suitable approach, because each possible combination must exist in a separate subtype definition[32]. Furthermore subtyping suffers from the problem that if a person changes his/her attributes after a while (e.g. ceases as a student and commences as an employee) this requires the deletion of one object and the creation of another; modifying the existing object is not possible.

While some OO languages (but not all) can support diamond inheritance, usually this is in a limited form and it creates complications both in terms of type definitions and implementations. Furthermore it doesn't fit well with the

32 In OO programming the possibility of modifying the behaviour of a supertype in subtypes further adds to the complexity.

idea of supporting components which can easily be added to an application. This issue will be considered later.

3 Multiple and Repeated Inheritance

Figure 6.2 provides an example of how an entertainment device type might inherit from two simpler entertainment device types, in one case repeatedly. In most OO languages defining and implementing such a type using conventional inheritance techniques would be difficult, if not impossible.

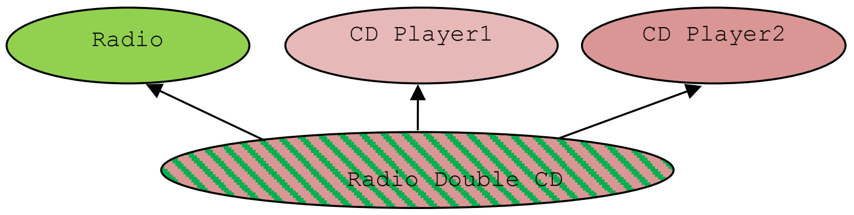

Figure 6.2: Multiple and Repeated Inheritance

4 Method Redefinition

In standard OO it is possible to override a method by changing its code in the subtype. However, changing code is an implementation technique. Usually, the behaviour of a method changes when its implementation changes, although that is not necessarily the case. Unfortunately it is not usually possible in conventional OO languages to distinguish between changes in behaviour and changes in the code.

5 Subclassing and Code-Reuse

Just as the attributes of a supertype are inherited in a subtype, so also the code which implements these attributes is re-used in implementations of the subtypes. This may seem to be a good idea, and in straightforward cases it often is. But in conventional OO programming languages subtyping and subclassing are very strongly related by the fact that they do not usually distinguish between type definitions (which are concerned with defining the logical attributes and properties of objects) and implementations of the type. These are bundled together into a single *class* definition which defines both the interface and the implementation of a class of objects[33].

The OO technique of combining subtyping and subclassing into a single mechanism causes problems in some cases, especially where multiple inheritance is involved. For example there are difficult issues involving both diamond

[33] Java allows *interfaces* to be defined separately, but this is optional for programmers, and there are restrictions regarding what can be defined in interfaces.

inheritance (see Figure 6.1) and multiple inheritance from more than one supertype (see Figure 6.2). Even in the case of single inheritance there can be problems. For example a type `DoubleEndedQueue` is not a true subtype of the type `Queue` (nor is a `Queue` a subtype of `DoubleEndedQueue`) and should therefore not be used polymorphically; but it is convenient to re-use the code of `Queue` to implement `DoubleEndedQueue` or vice versa [9].

6 Overriding Methods

It can sometimes be useful in a subtype to modify the implementation of methods which are inherited. This is achieved in conventional OO by repeating the method heading from the supertype class in the definition of the subtype. In some circumstances it can be useful as part of the new implementation of the method to invoke the original method in the supertype. This can usually be called by using a keyword such as *super*, but that only works for single inheritance.

7 Conclusion

There are several problems in the conventional OO paradigm, which Timor avoids, as is described in the following chapters.

Chapter 7 Type Inheritance in Timor

Timor's approach of separating types from their implementations has proved to be a valuable tool for overcoming the problems mentioned in chapter 6. In Timor, subtyping, polymorphism and method behaviour are issues relevant only to the definition of types. By separating these from issues related to implementations, Timor is able to define subtypes involving diamond inheritance [10], subtypes which have multiple and repeated supertypes [11] and the redefinition of methods.

The keyword **`includes`** has been introduced into Timor as an alternative to **`extends`**. Whereas the latter indicates a genuine subtyping relationship, the former indicates that a type definition includes the methods of another type without implying a subtype relationship. Consequently if a `DoubleEndedQueue` definition indicates that it **`includes`** `Queue` the compiler does not permit a `DoubleEndedQueue` variable to be assigned to a variable of type `Queue`. Together these two techniques (extension and inclusion) are referred to as *derivation*.

1 Single Inheritance

The inheritance examples shown in the more general discussion of inheritance (chapter 6 section 1) have already illustrated the Timor approach to very simple single inheritance.

2 Method Redefinition

Following the Java approach [20] to method collisions, Timor distinguishes between collisions merely in the names of methods and collisions of method signatures. Collisions of complete method signatures are treated as cases of redefinition, while collisions simply in the names of methods (i.e. where the signatures otherwise differ) are treated as overloading. When overloading occurs, each inherited method is considered to be a separate method. Thus discussions of collisions in the sequel refer to cases where the method signatures are indistinguishable.

It is possible in Timor to separate the redefinition of behaviour of a method from the overriding of its code, simply because behavioural redefinition is a matter which affects type definitions while overriding is an implementation issue. Thus in derived type definitions there is a section headed by the keyword **`redefines`**, which list the methods in question and allows the new behaviour to be informally redefined in comments[34].

3 Views

Timor provides a further construct related to inheritance, called a **`view`**. Some programming language experts would consider this to be an abstract type. However, treating a view as a separate construct allows it to be used without all the properties associated with abstract types, but with other features not usually associated with abstract types.

A **`view`** is a set of related interface methods which can usefully be included in many types (using **`extends`** or **`includes`**), *without embodying the central concept usually associated with a type*. For example, when modelling vehicles one might create a subtyping hierarchy which includes an abstract type `Car`. This clearly embodies a central concept behind a certain class of motor vehicles, and it may have a number of concrete subtypes corresponding to particular models of cars. Thus both the concrete and the abstract types generally correspond to nouns. But Timor views correspond rather to a certain class of adjectives, often ending in "able" and describing some aspect of many kinds of objects, often of quite different types. Here is an example of a Timor view which might usefully be included in the definitions of many types of devices and products.

```
view Switchable {
instance:
 op void switchOn();
 op void switchOff();
 enq Boolean isSwitchedOn();
}
```

This might for example be used to extend a type `Radio`, as follows:

```
type Radio {
```

[34] It is a long term aim to enhance the design of type definitions in Timor with a more formal technique for defining the behaviour of methods.

```
 extends:
  Switchable;
 instance:
  setStation(...);
  ...
 }
```

With this definition a `Radio` instance could be assigned polymorphically to a `Switchable` variable, e.g.

```
Radio aRadio = Radio::Impl();
Switchable s;
s = aRadio;
```

Views can, but need not, have implementations.

Views are normally defined explicitly, but they can also be defined retrospectively from methods in existing type definitions and used as if the type had been defined by extension. For example an instance of a type which had been declared (without the use of a view) with instance methods such as `switchOn` and `switchOff`, could be assigned to a view variable `Switchable`, provided that the corresponding method definitions are identical.

4 Multiple and Repeated Inheritance

Assuming the existence of two type definitions, one for a type `Radio` and one for a type `CdPlayer`, the example of multiple and repeated inheritance illustrated in Figure 6.2 can be defined in Timor as follows.

```
type RadioDoubleCdPlayer {
 extends:
  Radio r;
  CdPlayer cd1, cd2;
}
```

Notes:

1) To achieve inheritance from multiple types, the syntax of the **`extends`** section allows more than one type to be listed.
2) To allow for repeated inheritance, identifiers (called *part names*) have been associated with the type names. (Note: The identifier `r` could be omitted from the `Radio` type, but it has been included as it will prove useful in the implementation presented in the next chapter.)
3) In this form the **`extends`** section looks rather like an **`instance`** section containing abstract variables, but they are not! In fact the modelling of such types is sometimes presented in that form (known as aggregation), but then

it cannot achieve two results which are straightforward in Timor: the polymorphic aspect (e.g. assigning a `RadioDoubleCdPlayer` to a variable of type `Radio` or of type `CdPlayer`) and the redefinition of methods.

We now consider the last two points from the viewpoint of type definitions.

a) *Polymorphism*: If repeated inheritance is not involved, part names need not be used. In that case a subtype can be assigned to a supertype variable in the usual way, e.g.

```
RadioDoubleCdPlayer rdcd = RadioDoubleCdPlayer::Impl();
Radio aRadio = rcdc;
```

However to do this where repeated parts are involved would be ambiguous, so in this case it is essential to nominate the part required, e.g.

```
CdPlayer aCD = rcdc.cd2;
```

b) *Method Redefinition*: In the original definition of `Radio` the view `Switchable` was included by extension. Assuming that this was also included in the type `CdPlayer`, its methods would appear three times in `RadioDoubleCd Player,` providing separate switching mechanisms for the three main components of the type, which would hardly be satisfactory. If aggregation had been used to model this device, this could not be changed. Timor can achieve the same result as that of aggregation; for example to switch on the separate switch for the second `CdPlayer`, the programmer could write:

```
rcdc.cd2.switchOn();
```

However, if the intention were to provide a single `switchOn()` method for both `CdPlayer` devices, the Timor definition of the combined device could be formulated as follows:

```
type RadioDoubleCdPlayer {
 extends:
  Radio r;
  CdPlayer cd1, cd2;
 redefines:
  [cd1, cd2] op void switchOn();
}
```

where the square brackets indicate which methods should be merged. However, it would be tedious to repeat this for all the methods of `Switchable`, especially if all three switches were to be merged. To do this, the programmer simply has to write

```
 redefines:
  [r, cd1, cd2] Switchable;
```

Finally, the user of such a device must be able to determine which of the devices should actually be in use at any particular time. For this purpose a new instance method (and an accompanying enumeration type) can be added:

```
enum Mode {playRadio, playCD1, playCD2}
type RadioDoubleCdPlayer {
 extends:
  Radio r;
  CdPlayer cd1, cd2;
 redefines:
  [r, cd1, cd2] Switchable;
 instance:
  op void changeMode(Mode m);
   // determines which device is active
}
```

5 Diamond Inheritance

Timor has a rule that when multiple types extend a single common ancestor, the methods of the base type appear only once in the new type. Thus at the type level there is no problem in defining situations which lead to simple diamond inheritance. Thus given types `Student` and `Tutor`, which both extend `Person`, the following definition suffices:

```
type StudentTutor {
 extends:
  Student;
  Tutor;
}
```

In appropriate cases methods can also be redefined. However, using this OO style does not solve the problems mentioned earlier, i.e. that new attributes cannot be added to individual objects based on this type definition and that attributes cannot be removed or changed without deleting and creating instances of the type. For such reasons Timor has an alternative mechanism (called *attribute types*), which is especially useful in database applications, but which can of course be used in any program, since Timor is a persistent programming language. This technique, which is not based on conventional inheritance, is introduced in chapter 9.

Part identifiers, which were introduced to permit repeated inheritance in types with multiple ancestors, can also be declared in types which share a common ancestor. The following shows how a type can be defined in which a student is a tutor in two departments.

```
type StudentTutor {
 extends:
  Student;
  Tutor dept1, dept2;
}
```

6 Inheriting from a Common Abstract Ancestor

The Timor Collection Library (TCL) provides a good example of the type issues associated with multiple inheritance from a common *abstract* ancestor. The following is based largely on the doctoral thesis of my former assistant Dr. Gisela Menger [2].[35]

The first criterion is concerned with the decision whether *duplicates* are permitted in collections of items (as in a mathematical *bag*), and if they are not permitted, whether they are simply ignored (as in a mathematical *set*), or whether an error is signalled when an attempt is made to insert a duplicate (which is often important in database applications).

The second criterion is concerned with the *order* of the elements in a collection, i.e. whether they are unordered (as in a mathematical *bag* or *set*), whether they are user ordered (as in a *list*) or whether they are automatically *sorted* according to some criterion, e.g. alphabetically.

The TCL supports all nine combinations, as is shown in Table 7.1.[36]

Collection Type Name	Duplication Criterion	Ordering Criterion
Bag	Allow duplicates	No ordering
Set	Ignore duplicates	No ordering
Table	Signal duplicates	No ordering
List	Allow duplicates	User ordered
OrderedSet	Ignore duplicates	User ordered
OrderedTable	Signal duplicates	User ordered
SortedList	Allow duplicates	Sorted
SortedSet	Ignore duplicates	Sorted
SortedTable	Signal duplicates	Sorted

Table 7.1.: The concrete collection types

In addition there are five abstract types which are used to enhance polymor-

[35] In practice the TCL is defined generically. Genericity in Timor is introduced in chapter 12.

[36] The following description is based largely on parts of section 4 of [8].

phism. The full type hierarchy is shown in Figure 7.1.

In order to guarantee behavioural conformity, all the common methods of all collection types are initially defined in the abstract type `Collection`. Thus it has a method `insert`, for example, but this does *not* define

– how an insertion affects the ordering of the collection,
– whether the insertion will be successful if it involves inserting a duplicate,
– whether an exception will be thrown to indicate a duplicate (but it defines an exception `DuplEx` which *might* be thrown).

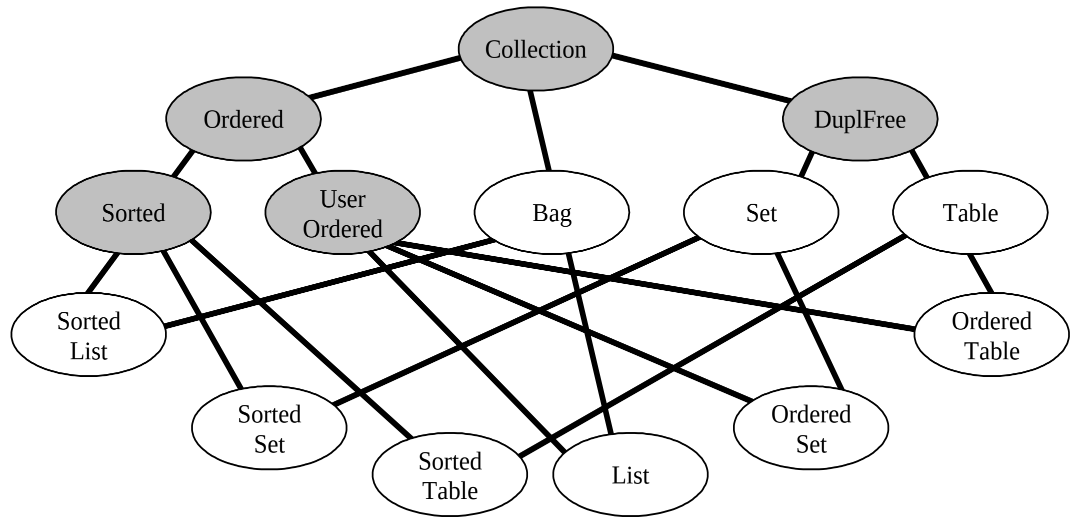

Figure 7.1: Structure of the TCL Collection Types

An abstract type with such non-deterministic methods is designed to allow a maximum of polymorphism. In derived types the actions of the `insert` method are specified more precisely, depending on the node in question. Thus the `insert` method of the abstract type `UserOrdered` defines that `insert` appends the element at the end of the collection (and adds new methods for inserting at other positions) but without defining its duplication properties further. On the other hand the `insert` method of the concrete type `Bag` is defined without specifying ordering, but indicating that duplicates are accepted (with the effect that the exception `DuplEx` can be removed from Bag's `insert` method).

Such redefinitions of methods must be reflected by listing them in a **`redefines`** clause of a derived type. As Timor does not support a formal specification technique, only the headers of such methods are listed, but these can include comments describing their intended behaviour. Sometimes a redefinition can lead to a change in the method header (e.g. where an exception defined in a parent type is not thrown in a derived type, cf. `Collection` with `Bag`), but in many cases the method header remains the same (though hopefully programmers will be encouraged to document the redefined behaviour in comments).

7 Type Rules

Type Inheritance Rule 1: If in a derived type multiple methods with the same signature[37] are derived from a common ancestor, they are treated as a single method (unless they have different return types, in which case a compile time error arises).

Type Inheritance Rule 2: If the definitions of such methods differ (i.e. if one or more of them has been redefined differently from the definition in their closest common ancestor), they must also be listed in a **redefines** clause in the type being defined.

Rule 2 in effect requires that conflicting definitions are clarified. If a definition in one of the ancestors can be used in the new type, this can be signalled by the use of the keyword `from` followed by the name of a type, e.g.

```
redefines {
  op void insert(ELEMENT e) from UserOrdered;
}
```

8 Handles

The assignment of subtype instances to supertype variables in Timor is normally restricted in the sense that value instances can only be assigned to value variables of the same type, object instances to object variables of the same type and capability instances to capability variables of the same type. However, the type name in a declaration can be followed by three asterisks, with the meaning that instances (of that type) in any of the three modes can be assigned to such a variable (e.g. **Person*****). Such variable declarations are called *handles*. If a variable has the three asterisk notation, all values, references and/or capabilities of that type (or a subtype) can be assigned to it. This is useful in cases where the mode of the object is irrelevant, e.g. in methods which compare the values of two instances of the same type.

Furthermore, variables can be declared to have a special type **Handle**. This is the supertype of all types, in the limited sense that any variable of any type can be assigned to such a variable, provided that the mode is correct. (If the declaration is of type **Handle*****, then variables of any type in any mode can be assigned to it.) However, this "type" has no methods, and no instances can be created from it. One important use is to allow variables to serve as repositories for temporarily storing items of different types. Thus the designation **Handle**** is useful in defining ModelOS directory types which hold capabilities, where

[37] As in Java, exception declarations are not considered to be part of the signature of a method.

different entries must hold capabilities but not necessarily for the same type of modules.

9 Cast Statements

Cast statements allow the instances assigned to variables to be treated according to their actual type. For example if a `Student` instance is assigned to a `Person` variable, a cast statement can be used to check whether it really is a `Student`, and then treat it accordingly. (Remember that not all `Person` variables have `Student` instances assigned to them.)

The Timor cast statement is one of the few statements which is radically different from its counterparts in conventional OO languages. One reason for this change is to allow for the use of handles for the different modes of instances, but another is to improve run-time safety against programmer errors.

A simplified form of the Timor `cast` statement is as follows:

```
cast (variable | parameter) as {
  (variable declaration 1) {code for alternative 1}
  (variable declaration 2) {code for alternative 2}
  ...
  [else {optional code if no match}]
}
```

For example, to check whether a `Student` object or a `Professor` object is assigned to a `Person` variable called `aPerson`, the programmer can write:

```
cast (aPerson) as {
  (Student s) {code which addresses aPerson as s,
               if the underlying object is a Student}
  (Professor prof) {code which addresses aPerson as prof,
               if the underlying object is a professor}
else {code to be executed if aPerson is neither a Student
      nor a Professor}
}
```

Modes are relevant in cast statements and handles can be used. It is possible to cast upwards, sideways and downwards.

10 Comparison Operators and Subtyping

Comparisons between separate instances are not carried out via methods declared in the type, but in a related *co-type* (see chapter 11). If a comparison operator is used to compare the values of two instances which have different static types but have a common supertype, the `equal` and/or `less` methods of the standard co-type of the nearest common supertype are applied. Thus if a

`Student` instance is compared with say a `Professor` instance (or a `Person` instance) the `Person&s`[38] `equal` and/or `less` methods will be selected.

38 `Person&s` is the standard co-type for `Person` (see chapter 11).

Chapter 8 Implementations and Code Re-Use in Timor

Whereas in conventional OO programming languages the re-use of code is achieved via subclassing, which is based on inheritance, Timor separates code re-use entirely from inheritance. An implementation of a type is designated as the type name followed by a double colon followed by an implementation name, e.g. `Queue::ArrayImpl` designates an implementation (called `ArrayImpl`) of the type `Queue`. Each type must have at least one implementation, called `<typename>::Impl`. However, this must not necessarily have been explicitly coded. For example, the basic types have an implicit implementation; similarly the compiler can automatically provide implementations for abstract variables and records.

1 Re-Use Variables

To support code re-use in Timor a new concept, called *re-use variables*, is introduced. Like other concrete variables, such variables are included in the **`state`** sections of implementations, but unlike most other variables which typically appear in a **`state`** section, they may be declared either as types or as implementations. They may *not* be declared as local variables in individual methods.

Re-use variables are like normal variables in that they form part of the state of the implementation in which they are declared. They are recognisable because their declarations begin with a hat symbol (^), e.g.

```
^Queue myQueue = Queue::ListImpl(); /* Here the re-use
     variable myQueue is declared as a type variable and
     a list implementation constructor initialises it */
```

or

```
^Queue::ArrayImpl myQueueImpl = Queue::ArrayImpl(100);
 /* Here the re-use variable is declared as an implementation
```

```
      variable and is initialised by an array implementation
      constructor */
```

or in the case of a typeless implementation simply

```
  ^::usefulCode
```

In all cases the programmer of the implementation in which the declarations are embedded has access to their interface methods, but in the case of an implementation variable being declared, the programmer can also access its internal **`state`** variables and its private instance methods. (If a re-use variable is declared via a type declaration, a maintenance programmer can recognise immediately that *any* implementation of the type can be used.)

A re-use variable may even be another implementation of the type being implemented. On the other hand the type of a re-use variable does not necessarily have any formal relationship with the type being implemented, except that some (or all) of the interface methods may have the same definition.

The important difference between a re-use variable and a normal variable is that the compiler compares the definitions of its interface methods with those of the type being implemented. If some of these have matching signatures, it uses the methods of the re-use variable to implement them, unless the programmer has also declared the same method explicitly in the **`instance`** section. In the latter case the explicit method in effect *overrides* that provided by the re-use variable.

Interface methods of a re-use variable which do not match the type definition are ignored. However they can be invoked in the implementation by programmers.

2 Clashing Methods in Re-Use Variables

Several re-use variables can appear in a **`state`** section. If more than one of these has a method which matches an interface method of the type being implemented, the first match (in the order of the declarations) is selected, though the programmer can override this by declaring the clashing method explicitly in the **`instance`** section and then from this method simply invoking the preferred method implementation.

This technique easily imitates (and simplifies) the standard OO techniques of delegation and normal subclassing. Since there is no relation between the type of the re-use variable and the type in which it is embedded, the former can be a subtype or a supertype of the latter or it may implement a formally unrelated type.

Here is an example of how an implementation of `Person` might be used to implement `Student`.

```
impl Student::Impl {
state:
 ^Person aPerson = Person::Impl();
  // re-uses any implementation of Person
 ...
instance:
 /* the Student public methods added in the subtype
    are implemented here and methods of Person can also be
    overridden */
}
```

3 Reversing the Re-Use Relationship of Subtypes

Since any implementation in Timor can be a complete implementation of a type (without using re-use variables), an implementation of `Student` might be coded independently of an implementation of `Person`. In this case the re-use relationship can be reversed, e.g.

```
impl Person::Impl {
state:
 ^Student aStudent = Student::Impl();
}
```

In this example an **`instance`** section is not needed (unless overriding is required), because all the methods of `Person` can be matched in the methods of `Student`.

4 Re-Use of Independent Types

Similarly, given the following definition of the type `DoubleEndedQueue`,

```
type DoubleEndedQueue {
includes:
 Queue;
instance:
// methods added to make the queue double ended
}
```

and an implementation `DoubleEndedQueue::Impl`, here is a complete implementation of the type `Queue`:

```
impl Queue::Impl {
state:
^DoubleEndedQueue deq = DoubleEndedQueue::Impl();
}
```

The various implementations of a type are in principle independent of each other and where appropriate the methods can be re-implemented from scratch without code re-use. Since the types `DoubleEndedQueue` and `Queue` are not related at the type level, no polymorphic problems arise.

5 Overriding Code

In Timor the OO concept of overriding, as an implementation concept, simply involves providing a new implementation for a method in an **`instance`** section. This overrides any clashing methods from re-use variables. If it is appropriate to re-use some of the code from another implementation the latter is included as a re-use variable, and its method(s) can be called in the normal way.

6 Implementing Views

In preparation for presenting an implementation of multiple and repeated inheritance, a trivial implementation of the view `Switchable` is now illustrated:

```
enum SwitchState {off, on}
impl Switchable::Impl {
state:
 SwitchState switch = off;
instance:
 op void switchOn() {switch = on}
 op void switchOff() {switch = off}
 enq Boolean isSwitchedOn() {return switch}
 }
}
```

7 Implementing Multiple and Repeated Inheritance

Re-use variables have the advantage that they re-use not only the methods of an implementation but also its state. This greatly simplifies the implementation of types which use repeated inheritance, since in such cases multiple versions of state variables might be required. First the type `Radio` is implemented:

```
impl Radio {
state:
 ^Switchable = Switchable::Impl();
instance:
 setStation(...);
 // an implementation of the the remaining Radio methods
 ...
}
```

This, together with a similar implementation of `CdPlayer`, is now re-used in the combined device, illustrating the re-use of state, as well as access to and the overriding of methods of re-use variables:

```
impl RadioDoubleCdPlayer::Impl {
state:
 ^Radio r = Radio::Impl();
 ^CdPlayer cd1 = CdPlayer::Impl();
 ^CdPlayer cd2 = CdPlayer::Impl();
 SwitchState theSwitch = off;
 Mode currentMode = playRadio;
instance:
 op void switchOn() // overrides this method
 {theSwitch = on; r.switchOn(); currentMode = playRadio}
 // the radio is on by default
 op void switchOff() {
  case (currentMode) of {
   (playRadio) {r.switchOff();}
   (playCD1)   {cd1.switchOff();}
   (playCD2)   {cd2.switchOff();}
  }
 }
 enq Boolean isSwitchedOn(){return theSwitch;}
 op void changeMode(Mode m) {
  case (m) of {
   (playRadio) {cd1.switchOff(); cd2.switchOff();
                r.switchOn();}
   (playCD1)   {r.switchOff(); cd2.switchOff();
                cd1.switchOn();}
   (playCD2)   {r.switchOff(); cd1.switchOff();
                cd2.switchOn();}
  }
  currentMode = m;
 }
}
```

8 Implementing Diamond Inheritance

We apply re-use variables in what at first sight might appear to be the obvious way to implement a `StudentTutor`.

```
impl StudentTutor::Impl {
 state:
  ^Student s;
```

```
    ^Tutor t;
  }
```

Although this syntactically matches the type definition, *semantically it does not*, because each of the two re-use variables has its own state, i.e. there are two sets of state variables for the `Person` part of each! Including a further re-use variable `^Person p` does not solve the problem as such, since any `Student` or `Tutor` methods which might access the `Person` part would access the wrong `Person` state. It would be possible to override these methods to produce a correct result, but this would create considerable work for the programmer, which would increase with each added attribute.[39] This problem arises as a result of the lack of genuine modularity behind the idea of subtyping, which binds extensions very tightly (not as attachable units) to a base type.

Other consequences of this phenomenon have already been mentioned, viz. the inability to add and remove attributes dynamically for individual objects. Together, these issues, which are important in database applications, led to the decision to introduce a new kind of type, which retains the key properties of polymorphism but in a more modular way. This is described in the next chapter.

9 Implementing Types with a Common Abstract Ancestor

Since Timor does not bind implementations to their types in the form of subclassing, the Timor Collection Library (TCL) can be implemented in an unusual way. The first type to be implemented is `List`, and the code of this implementation is declared as a re-use variable in standard implementations of all the other concrete types, with remarkably few modifications. This is illustrated in chapter 13, which provides an outline of both the type definitions and implementations of the TCL methods. Furthermore, application programmers are free to provide their own implementations of the concrete TCL types and can also extend the TCL type hierarchy and/or the TCL implementations.

However, this does not preclude the existence of other implementations of individual types in the TCL (e.g. using a bit list to implement a fixed size *set* of elements).

[39] It would of course be possible to implement `StudentTutor` from scratch.

Chapter 9 Attribute Types

Timor provides a modular alternative to a particular aspect of conventional inheritance, in the form of *attribute types*. These are useful primarily when a base type (a *concrete* type such as `Person`) can potentially have subtypes which require add-on *state*, especially if for individual objects of the base type these can change dynamically. Such types can often be defined largely in terms of Timor abstract variables (corresponding to records in database systems, see chapter 5), though additional methods can also be added.

Attribute types are *not* appropriate, for example, for defining subtypes which primarily exhibit different variations on the behaviour of a base type (often, but not necessarily, an *abstract* type such as `Collection`) and which at the individual object level have a state that is not logically extended by additional state relating to its individual subtypes.

Timor's attribute types allow the add-on methods and state to be treated as separate issues. Thus a type `Person` can be declared (as described in chapter 5 section 2), and can be instantiated as an object in the usual way. A separate type `Studying`, which contains the attributes required to make a `Person` into a student (but without the `Person` attributes) can also be defined and implemented. This can be separately instantiated as an object and can then be attached to a `Person` object (and later detached). Other attribute types can be defined in the same way (e.g. a type `Tutoring`) and can also be attached to the same `Person` object. In this way a `StudentTutor` can be created without the problems associated with diamond inheritance. This is possible because, although the individual attribute types have static definitions, there is no requirement (in contrast with the subtyping technique) for a combined type to exist statically.

1 Defining and Implementing Attribute Types

Generally speaking attribute types are given adjectival names (although this is only a convention), because they add additional information to a base type, just as adjectives add information to a noun. Hence instead of naming an attribute `Student`, for example, it is more appropriately called `Studying`, and when the attribute is associated with a `Person` instance, it is appropriate to call this a `StudyingPerson`. However, such a naming scheme is simply by convention.

Here is a possible definition of a type `Studying`, which looks very similar to the definition of the subtype `Student`.

```
type Studying for Person {
instance:
 String university;
 Int studentId;
 String faculty;
 Date commencement;
 String degree;
}
```

The keyword **`for`** indicates that a type is an attribute type and also indicates the base type (here `Person`) to which it can be attached. In this example an implementation, called `Studying::Impl`, would be automatically produced by the compiler. Implementations follow the normal pattern (see chapter 5 section 2).

Attribute types can be defined for other attributes. For example an attribute `PartTime` might be attached to the `Tutoring` attribute, to indicate a part-time position as tutor.

The *public* methods of a base type can be accessed in an implementation of the attribute type via the pseudo-variable **`base`**. Limiting the access in this way ensures that no behaviourally non-conform accesses can take place, thereby guaranteeing for other attributes attached to the same object that related problems cannot arise [10].

Attribute types can also be defined as **`for any`**; these can be attached to any other type. Here is an example:

```
type Loanable for any {
instance:
 op void putOnLoan (Person* toWhom; Date loanDate);
 op void returnFromLoan(Date returnDate);
 Boolean currentlyLoaned;  // an abstract variable
 Date dueDate;             // an abstract variable
 enq Int daysOverdue();
 enq Person* borrower();
 enq Person* previousBorrower();
 enq Date dateLastReturned();
```

```
  ...
}
```

The pseudo-variable **`base`** cannot be used in connection with **`for any`**.

2 Static Use of Attributes

Attributes can be composed statically into other types, as the following example illustrates.

```
type DoubleStudyingTutor {
extends:
 {Studying s1, s2; Tutoring;} Person;
}
```

This can be instantiated and accessed in the normal way, e.g.

```
Person p = DoubleStudyingTutor::Impl();
String theUni = p.s2.university;
```

Because attribute types and implementations are simply add-on units which do not include the base type, they can easily be used to solve the problems previously encountered in implementing diamond inheritance, as the following illustrates:

```
impl DoubleStudyingTutor::Impl {
state:
 ^Person;
 ^Studying s1;
 ^Studying s2;
 ^Tutoring t;
}
```

3 Instantiating Attribute Types

Attribute types can be instantiated as values or as objects, but not as modules known to the operating system, since neither ModelOS nor conventional operating systems support the idea of modules with add-on sections. However static types which include attributes can be created as modules.

Once they have been instantiated as objects, attributes can be attached dynamically to other objects.

4 Attaching Attributes Dynamically to Objects

An attribute can be attached to an *object* of its base type as follows:

```
Person* p = new Person::Impl();
Studying* s = new Studying::Impl();
p += s; // attach s to p
```

Given a further attribute type `Tutoring`, this could also be attached to the same `Person`, e.g.

```
Tutoring* t = new Tutoring::Impl();
p += t; // attach t to p
```

The result is a studying tutoring person, equivalent to `StudentTutor`, but without the complications of diamond inheritance. Multiple attributes of the same type can be dynamically added to the same base object. Thus if the student described above were enrolled in two universities, a second **`Studying`** attribute could be added:

```
Studying* s2 = new Studying::Impl();
p += s2; // attach s to p
```

5 Removing Attributes from Objects

An attribute can be removed from an object without deleting it, e.g.

```
p -= s;
```

In this case it might be reattached to the same or a different object later. However, an attribute can only be attached to one object at a time. Failure to follow this rule leads to a run-time error.

Alternatively an attribute can be removed from its base object simply by deleting it via its own object reference, e.g.

```
Person* p = new Person::Impl();
Studying* s = new Studying::Impl();
p += s;
...
delete(s);
```

If an attempt is made to access a previously attached but then deleted attribute, an exception is thrown.

6 Casting with Attributes

For the attribute relationships described in the previous section, explicit casts can succeed in both directions. For example to cast between `Person` and `Studying` the cast would have a pattern such as the following:

```
Studying* s = new Studying::Impl();
...
String aName;
cast (s) as {
 (Person* p) {aName = p.name;}
 // only executed if s is attached to a Person instance
}
```

and similarly

```
Person* p = new Person::Impl();
...
cast (p) as {
 [Studying* s] {if (s.uni == "Oxford") {...};}
}
```

In the second example the cast clause uses square brackets rather than round brackets. This indicates that the corresponding code is to be executed repeatedly, i.e. for each attached attribute of the matching type. In this way it is possible to select all matching attributes.

7 Final Remark

As noted above, attributes cannot be instantiated as modules. But their importance for ModelOS is that they provide a flexible way of building databases in the ModelOS persistent virtual memory.

Chapter 10
Qualifying Types

Qualifying types are types which can qualify or modify the behaviour of the instances of other types. We proposed the basic idea, under the name "attribute types"[40] in 1997 [21], when my research students and I were beginning to formulate ideas for a new language to support ModelOS. In fact they are related conceptually to the attribute types described in the previous chapter, in that both can be viewed as *adjectival* types. To understand what is meant by this, consider that in object oriented programming the emphasis is on defining *objects* which in natural language usage are represented as *nouns*. These can often be qualified by adjectives. For example a *person* (a noun) can be described as *studying* (an adjective, actually an adjectival form created from a verb, known as a present participle). In natural language, nouns (e.g. *student*) can exist which in a single word convey the meaning of a more general noun (*person*) combined with an adjective (*studying*). We used the corresponding pattern in the previous chapter by allowing attributes to be added to objects to form more specific forms of object, e.g. `{Studying} Person`.

Qualifying types are conceptually related to attribute types in that they also allow us to express adjectival qualities (in this case often better expressed by past participles, reflecting the passive voice, used as adjectives, e.g. *monitored*) which can be associated with nouns. But whereas attribute types simply add new behaviour which in computer terms is easily expressed as further attributes, qualifying types added new features which can radically affect the behaviour of the objects which they qualify.

Since the publication of the first paper in 1997 we have considerably developed the basic idea behind qualifying types [13, 22, 23, 24, 25, 14]. Qualifiers (i.e. instances of qualifying types) play a central role for ModelOS by

[40] This should not be confused with the Timor concept of attribute types described in chapter 9 above.

providing a technique which gives its users an important mechanism for confining and controlling the flow of information from other modules, as is described in chapter 24 of [6], see also [24]. But they also have other significant roles, such as providing synchronising access to shared data, deceiving hackers and recording activity of threads, etc. How they are implemented at the module level in ModelOS is described in volume 2 chapter 24 of [6].

1 Qualifiers: The Basic Idea

A qualifier is an instance of a qualifying type which has all the normal features of objects, including its own data and methods. But it also has some special methods, known as *bracket methods*, which are designed to bracket the code of other objects. There are two kinds of bracket methods, *call-in* and *call-out* brackets, which are activated differently from normal methods. For readers unfamiliar with the idea behind these concepts we repeat part of the description in chapter 13 of [6] in the following subsections.

1.1 Call-In Bracket Methods

When one object calls a method of another, this can be represented as shown in Figure 10.1:

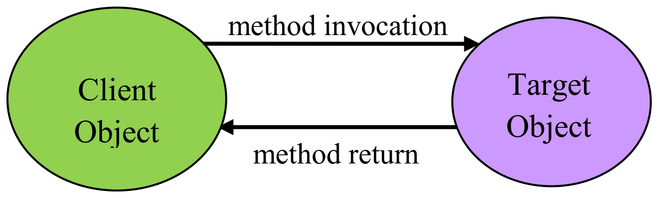

Figure 10.1: A Normal Method Invocation

A qualifier can be associated with a target object such that its call-in bracket methods can "catch" a normal method invocation before it reaches the target object (i.e. its qualified object), i.e. instead of the code of the method of the target being invoked, the code of the appropriate call-in bracket method is invoked (see Figure 10.2).

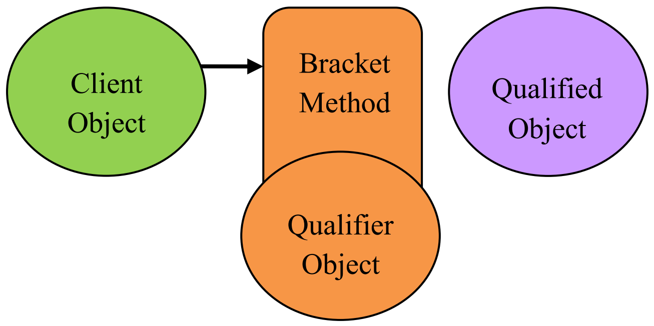

Figure 10.2: A Qualifying Type with a Call-In Bracket Method

Depending on how it has been defined, the bracket method may have access to the parameters which the client object intended to pass to the qualified object. But it has no access to the state data of either the client object or of the qualified object.

1.2 The Body Statement

A call-in bracket method contains normal code, but it has one extra feature, called a *body* statement. The effect of this is to call the method of the qualified object which the client originally intended to call. This organisation of bracket methods gives its programmer a number of interesting options.

1.3 Augmenting Bracket Routines

Additional code can be added before calling the qualified object (in the part of the bracket method called a *prelude*). This code might for example access synchronising variables in the data of the qualifier, thus causing an unsynchronised qualified object to be synchronised. Or from the security viewpoint it might for example maintain a log of calls to the qualified object which can later be printed out or analysed by another computer program to detect attempts to hack the qualified object.

When the method of the qualified object has completed its task, it returns to the *postlude* section of the call-in method (i.e. the statements following the *body* call). In the postlude section it can, for example, release the synchronisation variables. This option, which augments the qualified object, is shown in Figure 10.3.

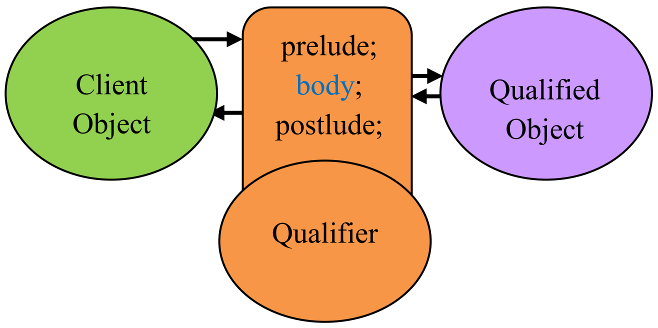

Figure 10.3: An Augmenting Bracket Method

1.4 Testing Bracket Methods

Code in the prelude can check some condition (e.g. a security condition) and depending on the result might decide not to invoke the interface method of the qualified object. The result might be that the target object is not called at all. This is illustrated in Figure 10.4.

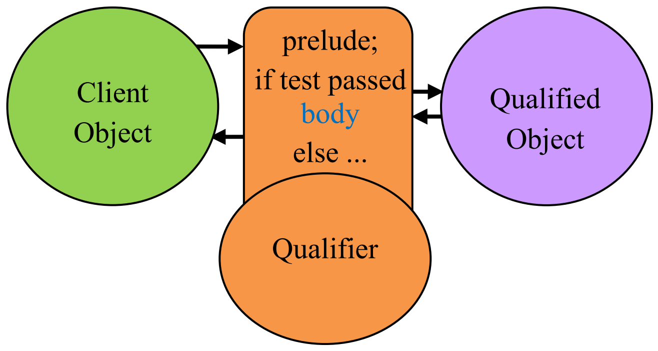

Figure 10.4: A Testing Bracket Method

1.5 Replacing Bracket Methods

Finally, the bracket method need not contain a *body* call at all (not even in a conditional statement). In this case the target object is in effect replaced by the qualifying object. One possible use of this is to set up a qualifier as a decoy for a hacker, which serves as a disinformation technique. Figure 10.5 illustrates this possibility.

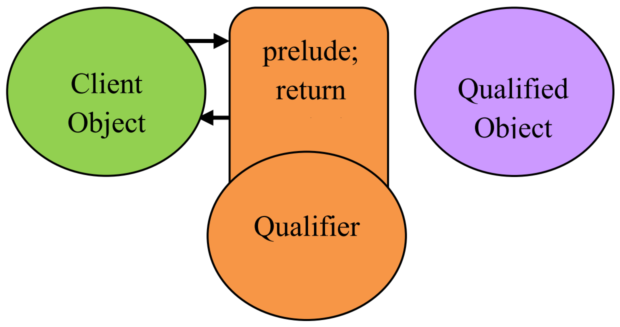

Figure 10.5: A Replacing Bracket Method

1.6 Multiple Qualifiers

More than one qualifier can be associated with a qualified object. In this case there is a defined order such that the first is invoked as a result of a routine call from a client object, the next is then invoked if this makes a body call, etc.; a body call from the final qualifying object (if it ever happens) results in the target object being called. The postludes are executed in reverse order.

1.7 Call-Out Bracket Methods

The principle of call-out bracket methods is similar to that of call-in methods, except that

a) they are triggered by a call *from* a qualified object to some other object (the call-out object);

b) a *call* statement (cf. the *body* statement for call-in methods) is used if the call-out bracket decides to pass the call on to the call-out object.

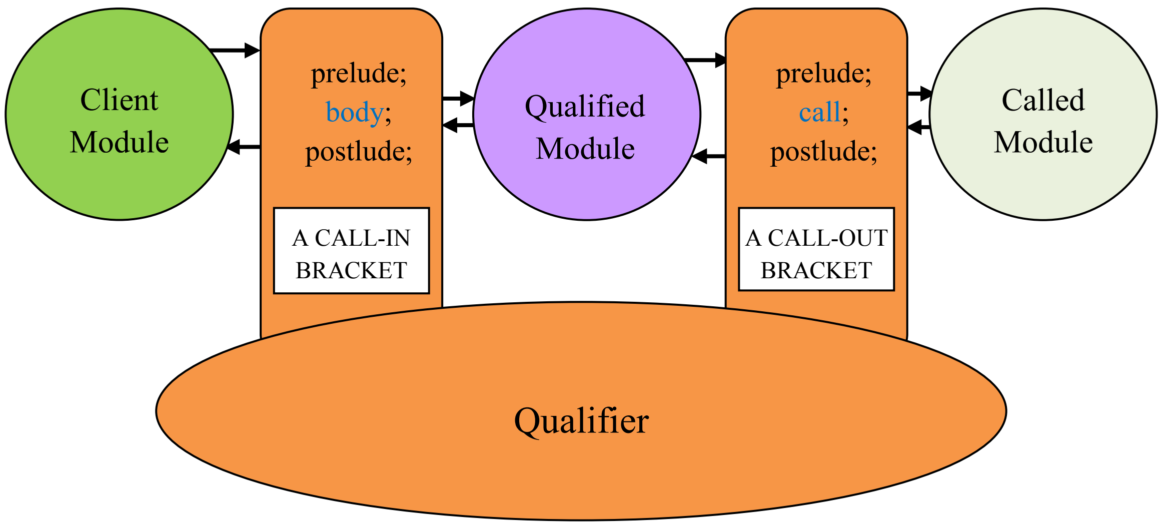

Figure 10.6: A Qualifier with Call-In and Call-Out Bracket Methods

The basic concept is illustrated in Figure 10.6, where a qualifying object has both call-in and call-out bracket methods. However, a qualifier can be programmed to have only call-in or only call-out routines if that is appropriate.

Call-out brackets can be freely programmed to include or omit a *call* statement, and can optionally place it in a conditional statement.

At first sight it might be thought that call-out routines are superfluous, with the argument that they could be implemented as call-in brackets of the call-out object. However, this is not the case, because a call-in bracket is activated whenever the qualified object is called, whereas a call-out bracket is activated each time the qualified object makes a call to another object, not each time the called object is invoked. However both a client object and its qualified object can be qualified (usually, but not necessarily, by different qualifier objects), as is shown in Figure 10.7.

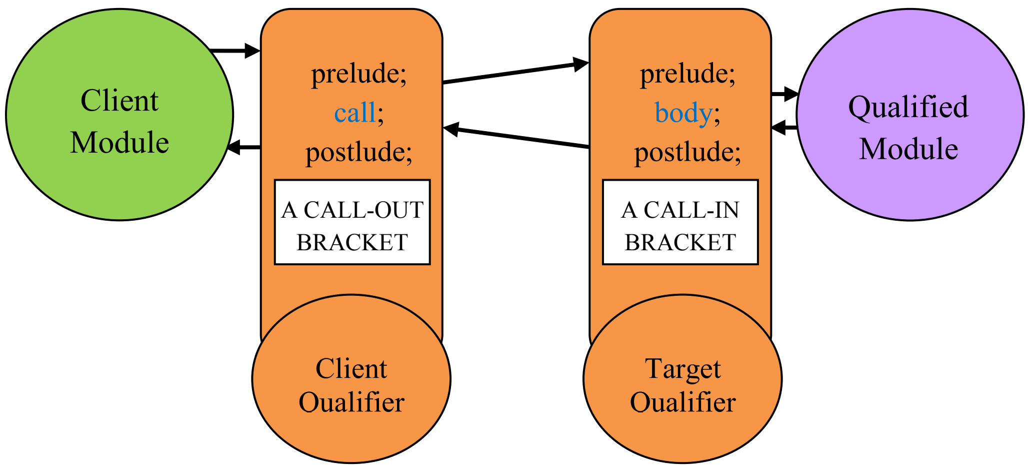

Figure 10.7: A Client with Call-Out and a Target with Call-In Brackets

The following sections describe how qualifying types are defined in Timor.

2 Qualifying All the Methods of Any Type in the Same Way

It is possible to define qualifying types which have bracket methods that can qualify any other type. Here is a simple but important example, which provides basic synchronisation in the form of mutual exclusion:

```
library type Mutex {
qualifies any:
 op bracket all(...); /* this provides mutual exclusion
    and brackets all the instance methods of a
    target instance */
}
```

Notes:

1) The type `Mutex` has no normal instance methods and only one bracket method. Most qualifiers have instance methods and can have more than one bracket method.
2) It can qualify *any* other type, as is indicated in the **`qualifies`** clause.
3) The bracket method is an operation (**`op`**) because it modifies its own state variable, i.e. a synchronising variable.
4) The special return type **`bracket`** indicates that the actual return type is that of the method which it is currently qualifying.
5) The keyword **`all`** replaces a method identifier and is used to indicate that the bracket method is used to qualify *all* the instance methods of its target, including bracket methods, if any (but excluding **`open`**/**`close`** methods).
6) The bracketed ellipsis **`(...)`** indicates that it ignores (and cannot access) the parameters of the methods which it qualifies.

The type `Mutex` can be implemented using semaphores, as follows:

```
impl Mutex::SemImpl {
state:
 Sem mutex = Sem::Impl(1);
qualifies any:
 op bracket all(...) {
  mutex.p();
  try {return body(...);}
  finally {mutex.v();}
 }
}
```

Notes:

1) The keyword **`body`** indicates where the target routine is called. The ellipsis

(...) indicates that the original parameters are passed on when **body** is executed.

2) The **return** statement indicates that the result parameters from the target are passed back to the caller.
3) The **body** statement is embedded in a **try ... finally** statement because the target (or a bracket method between this and the target) might throw an exception. In this case it is important that the mutual exclusion is released.
4) Not all **body** statements need to be embedded in a **try ... finally** statement, but for synchronisation it is usually important.

3 Distinguishing Between Reader and Writer Methods

The following standard example of reader-writer synchronisation illustrates how reader and writer methods can be bracketed differently.

```
library type RwSync {
    // provides reader-writer synchronisation
qualifies any:     // for any type
 op bracket op(...);  // brackets op methods (writers)
 op bracket enq(...); // brackets enq methods (readers)
}
```

Notes:

1) This type provides separate bracket methods for the reader and the writer methods of a target. The method definition **op(...)** stands for any method of the target which is an operation. The method definition **enq(...)** stands for any method of the target which is an enquiry.
2) As in the previous example, this qualifier can qualify *any* other type.
3) Although the second bracket method qualifies enquiries, it is itself an operation (**op**) because it modifies its own state variables, i.e. the synchronising variables.

An implementation now follows. The code selects the reader priority algorithm.

```
impl RwSync::ReaderPriority { /* This is an implementation
 of the type RwSync which is given the name ReaderPriority */
state:
 RwSem rwEx = RwSem::Impl(); /* the standard constructor
    which provides reader-writer synchronisation */
qualifies any:
 op bracket op(...) {  // the writer protocol
  RwEx.writep();
  try {return body(...);}
```

```
   finally {rwEx.writev();}
  }
  op bracket enq(...) { // the reader protocol
   rwEx.readp();
   try {return body(...);}
   finally {
    rwEx.readv();
   }
  }
 }
```

Note: The basic semaphore types `Sem` and `RwSync` are actually supported in ModelOS (see chapter 15, section 4). The above are merely illustrations of how bracket routines can function.

4 Qualifiers with Instance Methods

The previous examples are exceptional in that all their methods are bracket methods. Most qualifiers have instance methods which are defined in a normal **`instance`** section, i.e.

```
type typename {
instance:
  // the normal instance methods
qualifies qualified_type:
  // the bracket methods
}
```

To make an example more interesting it is shown how qualifying types can be created by extension. First an access control list module is defined, without bracket methods. This is useful as a stand-alone module.

```
seq AccessMode {noaccess, read, write}
type Acl {  // a normal access control list
instance:  // these are normal instance methods
 op void addUser(ContainerId user; AccessMode access)
  throws InvalidUser;
  /* adds a user to the ACL. This is used to check that
    thread owners have appropriate access
    to the controlled module */
 op void removeUser(ContainerId user)
  throws InvalidUser;
  // removes a user from the ACL
 enq AccessMode currentAccess(ContainerId user)
```

```
  /* In ModelOS the identifier of a user is a (world-wide)
     unique identifier, which is the same as the identifier
     of his very first container (file) */
   throws InvalidUser;
   // returns the current access rights of this user
 }
```

Then a qualifier is defined which inherits its methods.

```
 type AclQualifier {
 extends: Acl;
 qualifies any:
  enq bracket op(...) throws InvalidAccess;
  enq bracket enq(...) throws InvalidAccess;
 }
```

Assuming that an implementation `Acl::Impl` already exists, the implementation of the qualifier can re-use this.

```
 impl AclQualifier::Impl {
 state:
  ^Acl::Impl theACL = Acl::Impl();  // a re-use variable
 qualifies any:
  enq bracket op(...) throws InvalidAccess {
     /* The bracket routine checks that in writer routines
        the currently active thread has write access */
   ContainerId caller = kernel.currentThreadOwner();
     /* This is a call to a ModelOS kernel instruction, see
        chapter 15 */
   if (theACL.currentAccess(caller) != write)
    throw new InvalidAccess();
   else return body(...);
  }
  enq bracket enq(...) throws InvalidAccess {
   /* The bracket routine checks that in reader routines
      the currently active thread has at least read access */
   ContainerId caller = kernel.currentThreadOwner();
   if theACL.currentAccess(caller) == noaccess
    throw new InvalidAccess();
   else return body(...);
 }
```

Notes:

1) Timor programs can call some instructions of the ModelOS kernel without presenting a capability[41] by using the reserved name **`kernel`** as if it were a callable object.
2) `ContainerId` is a built in type in Timor, along with similar in-built ModelOS types.

5 Qualifying Specific Methods

Bracket methods can be defined to qualify specific methods of some type or view. This possibility is illustrated using a view `Openable`, which can usefully be added to many types.

```
enum OpenMode {closed, read, write}
view Openable {
instance:
 open void open(OpenMode mode) throws OpenError;
 close void close() throws CloseError;
 enq OpenMode openMode();
}
```

The following qualifier illustrates how objects/modules which incorporate this view can be synchronised as readers and writers.

```
type OpenSynchroniser {
qualifies Openable:
 open void open(OpenMode mode) throws OpenError;
  /* allows multiple readers or a single writer.
     OpenError is thrown if the mode parameter is 'closed' */
 close void close() throws CloseError;
   /* throws InvalidAccess if current OpenMode is
      'closed', otherwise releases the synchronisation */
 enq OpenMode openMode();
qualifies any:
 enq bracket op(...);
 /* throws InvalidAccess if not open for writing */
 enq bracket enq(...);
 /* throws InvalidAccess if not open for read or write */
}
```

Notes:

1) This type has no instance methods, although it brackets instance methods of the target module.

41 See chapter 15, section 3.

2) The target module must embody the view `Openable`. Defining this type to qualify a view has the advantage that it can be used to qualify many actual types which are openable.
3) The bracket routines listed in the **`qualifies`** `Openable` section qualify the routines of the target which match the bracket method definitions.
4) A specific bracket method can access the parameters of the routine which it qualifies if, as here for `Openable`, they are explicitly described. If the parameters of a specific method use the ellipsis notation **`(...)`** the bracket method has no access to them.
5) The two bracket routines listed in the **`qualifies any`** section qualify those routines of the target for which no specific bracket methods have been defined.

Here is an implementation, which again uses reader priority.

```
impl OpenSynchroniser::ReaderPriority {
state:
 RwSem rwsem = RWSem::Impl();
 OpenMode currentmode = closed;
qualifies Openable:
 op void open(OpenMode mode) throws OpenError {
  case (mode) of {
   (closed){throw new OpenError();}
   (read)  {rwsem.readp(); currentmode = read; body(...);}
   (write) {rwsem.writep(); currentmode = write; body(...);}
  }
 }
 op void close() {
  case (currentmode) of {
   (closed){throw new NotOpen();}
   (read)  {try {return body(...);} finally {rwsem.readv();}
   (write) {try {return body(...);} finally {rwsem.writev();}
   }
   currentmode = closed;
 }
 enq OpenMode openMode() {return currentmode;}
qualifies any:
 enq bracket op(...) {
  if ((currentMode == closed) || (currentMode == read))
   throw new InvalidAccess();
  body(...);
 }
 enq bracket enq(...) {
  if ((currentMode == closed)
   throw new InvalidAccess();
  body(...);
}
```

Notes:

1) The first part of the reader-writer protocols, for both readers and writers, appears in the bracket method for the *open* method.
2) This also contains **`body`** statements allowing correct calls to proceed to the main module. After **`body`** there is no postlude to be executed.
3) It would also be possible, for example, for the qualifier to have a simple access control list of users with permitted access rights. In ModelOS this could be used to allow the module's owner to revoke write access, while continuing to allow read access.
4) The bracket for the `openMode` method returns without using **`body`**, as it already has the information required.
5) In principle this qualifier could be considered unnecessary in its present form, since (almost) the same code could easily be implemented as a direct (re-usable) implementation of the view. However, it was useful to present this as an example, since it illustrates some aspects of qualifiers not previously discussed.

6 Call-Out Methods

The examples so far have only illustrated call-in methods. Call-out methods are very similar to call-in methods. They can be defined independently or in the same type as call-in methods. The following example shows how the original mutual exclusion example might be enhanced by call-out brackets which release mutual exclusion if the synchronised object calls another object and then reclaims it after the call has completed[42].

```
type ReleasableMutex {
qualifies any:        // the call-in bracket methods
 op bracket all(...);
 /* provides the synchronisation needed to enter a
    target method */
```

[42] Such a module may not necessarily be useful, since when the call-out reclaims exclusion, the state of the synchronised object might have changed. However the example is useful for illustrating the principle of call-in and call-out brackets.

```
 callout any:          // the call-out bracket methods
  op bracket all(...); /* releases the synchronisation
   when the target invokes methods of any other object */
 }
```

Notes:

1) Call-outs have a separate section headed **callout.**
2) This is the only formal difference at the type definition level.

Here is an implementation which re-uses that of the original example.

```
 impl ReleasableMutex::SemImpl  {
 state:
  ^Mutex::SemImpl inMutex = Mutex::SemImpl();
 callout any:
  op bracket all(...) {
   inMutex.mutex.v(); // releases mutual excl. on call-out
   try {return call(...);}
   finally {inMutex.mutex.p();} // reclaims mutual excl.
  }
 }
```

Notes:

1) The same rules apply to bracket methods as to normal instance methods with respect to the matching of methods in re-use variables. In this case the call-in brackets are matched.
2) The **call** statement is the call-out equivalent to the **body** statement in call-in brackets.

7 Combining Call-Out and Call-In Brackets

The following example illustrates how the same qualifier can be used in the sending of messages between two separate modules. In this scenario Module A invokes a method of Module B, sending it a plain text message as a parameter. In the returned value of the same call Module B sends a plain reply back to Module A.

The method in Module B which receives the message is defined in a view, allowing it to be easily incorporated into many modules, and at the same time facilitating the definition of the qualifier.

```
 view Transmission {
  op Text transmit(Text message);
 }
```

Bracket methods can be used to encrypt and decrypt the messages. Provided that the call-out bracket is placed in the first position of the callouts associated with Module A and the call-in bracket in the last position in the call-ins of Module B, the qualifier can be useful in at least two scenarios.

a) Within a single ModelOS node they can be used to ensure that other bracket methods cannot read the message or the reply.

b) When transmitting a message between nodes they can ensure that the message is not readable during transmission. (See further comment in the notes below.)

```
type Encrypting {
callout Transmission:
 enq Text transmit(Text message);
  // encrypts message and decrypts reply
qualifies Transmission:
 enq Text transmit(Text message);
  // decrypts message and encrypts reply
}
```

The framework for an implementation is now described:

```
impl Encrypting::Impl {
state:
 String theMessageKey, theReplyKey;
constr:
 Encrypting::Impl(String key1, key2){
  theMessageKey = key1; theReplyKey = key2;
 }
callout Transmission:
 enq Text transmit(Text message) { // message is plain text
  Text encryptedMessage, encryptedReply, plainReply;
// this is the prelude: encrypt the message and send it
  encryptedMessage = encrypt(message, theMessageKey);
/* the following modifies the input message so that it is
   encrypted and passes it to the receiver (or next bracket)
*/
  encryptedReply = call(encryptedMessage);
/* when the postlude is activated it receives a reply which
   has been encrypted by Module B's call-in bracket */
  plainReply = decrypt(encryptedReply, theReplyKey);
// It decrypts this and places is in the return parameter
  return plainReply;
 }
qualifies Transmission:
 enq Text transmit(Text message){ // message is encrypted
  Text decryptedMessage, encryptedReply, plainReply;
 // this is the prelude: decrypt the message and send it
  decryptedMessage = decrypt(message, theMessageKey);
  plainReply = body(decryptedMessage); // this illustrates
/* how in Timor an input parameter can be changed
   after return from body encrypt the reply and return it */
  encryptedReply = encrypt(plainReply, theReplyKey);
  return encryptedReply;
 }
instance:
 enq Text encrypt(Text plainText; String aKey) {
  ... // encryption algorithm
 }
 enq Text decrypt(Text encryptedText; String aKey) {
  ... // decryption algorithm
 }
}
```

Notes:

1) This configuration (see Figure 10.8) will only function on a single ModelOS node, since it relies on the use of shared state variables. If the modules are on different ModelOS nodes, the keys must first be exchanged as separate messages (or a copy of the same qualifier, with the same keys, could be used on each node).

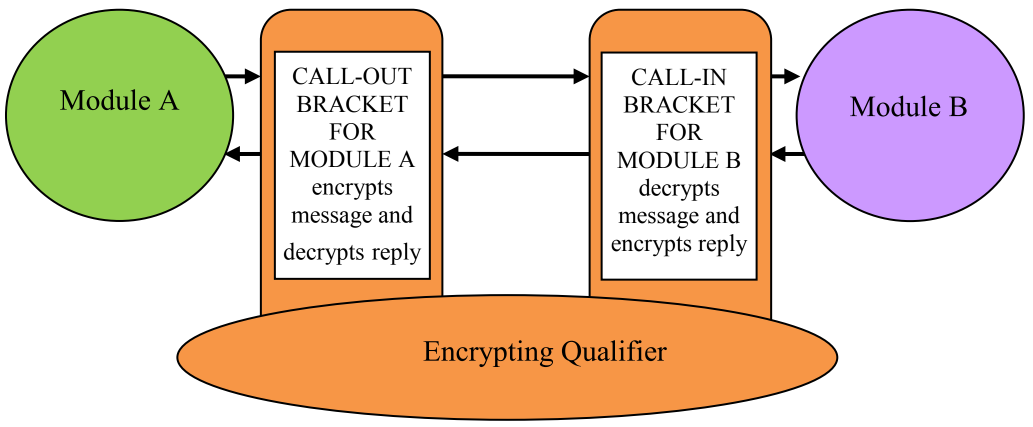
Module A
CALL-OUT BRACKET FOR MODULE A encrypts message and decrypts reply
CALL-IN BRACKET FOR MODULE B decrypts message and encrypts reply
Module B
Encrypting Qualifier

Figure 10.8: An Encrypting Qualifier Bracketing a Sender and a Receiver

2) The keys are not visible in the type definition, but only in the definition of a constructor for a particular implementation (in this case with separate keys for the message and for the reply). Different implementations might use dif-

ferent algorithms. For example a single key could be used for encrypting both the message and the reply, or different keys might be needed for the encryption and decryption algorithms, etc. None of this is visible from the type definition and only the sender and receiver need know not only which keys but which implementations are being used.

3) Although the instance section contains the encryption methods, these do not appear in the type definition. This is how internal methods are implemented in Timor.

4) The example illustrates how a bracket method can modify parameters and return values.

8 Instantiating and Using Qualifiers

Qualifiers can be dynamically associated with the objects which they qualify, as is described in section 8.1, or they can be statically associated with (i.e. "compiled into") their target objects, as is explained in section 8.2

8.1 Qualifying Target Objects Dynamically

Like attributes, qualifiers are often given adjectival names (which in this case are frequently past participles). Their instantiation follows the normal pattern, e.g.

```
AclQualifier* secured = new AclQualifier::Impl();
// AclQualifier is defined in section 4 above
```

This can be viewed as a normal object, and its instance methods can be invoked (e.g. to add entries into the ACL). To use it as a qualifier involves placing it in a list associated with the object(s) to be qualified. Objects of type `Book` can be instantiated and secured as follows:

```
Book* myBook1 = new {Qualifier* secured} Book::Impl();
Book* myBook2 = new {Qualifier* secured} Book::Impl();
```

In this case both books are secured by the same actual qualifier with the same ACL. The expression `{Qualifier* secured}` is an anonymous list literal[43] which in this example is permanently associated with its qualified object. The list can contain multiple qualifiers, e.g. `{Qualifier* secured, synchronised}`, where `synchronised` might be a qualifier of type `Mutex` or `RwSync`, etc. Since it contains copies of the qualifier references, the original, or other copies, can be used to insert and remove entries in the ACL after the book and the list

[43] The type of this list is `List<Qualifier*>*`. `Qualifier` is a supertype of all qualifiers. A `List` is a generic collection type which is part of the Timor Collection Library (TCL), described in chapter 13. For list literals see chapter 13, section 4.1.

have been created.

A qualifier list can be explicitly created and attached to a new object, e.g.

```
List<Qualifier*>* qualified1 = new List<Qualifier*>*::Impl();
Book* myBook1, myBook2 = new qualified1 Book::Impl();
```

This has the advantage that qualifiers can then be dynamically inserted into and deleted from the qualifier list, e.g.

```
qualified1.insert(secured); // inserted at the list end
qualified1.insertAtPos(synchronised, 0) // inserted at front
```

The `List` methods allow applications to define a precise order for the qualifiers in a list. This is important because the call-in brackets provided in the qualifiers of such a list are executed from left to right, while the call-out brackets are executed from right to left. Changes in the ordering can cause significant behavioural changes.

Since entries in a qualifier and qualifiers in a list can both be modified, it is important that changes take place in a synchronised manner. This is achieved by bracketing the list using the synchronisation mechanisms described in chapter 15, section 4.

Brackets can be used to qualify local objects in a module, but also to qualify ModelOS modules. In the latter case (file) lists of qualifier modules must be used and the lists themselves must be file modules, e.g.

```
List<Qualifier**>** qualified2 =
                             new List<Qualifier**>**::Impl();
PayrollFile** myPayroll = new qualified2 PayrollFile::Impl();
```

8.2 Qualifying Target Objects Statically

This subsection is based substantially on [23], and the reader is encouraged to read the entire paper, since some of its detailed content has not been covered here.

Like the attribute types described in the previous chapter, qualifying types can be tightly bound to, i.e. composed statically into, the objects which they qualify.

The syntax for this follows the same (adjectival) pattern as that used to bind attributes into an object. This syntax can be defined in EBNF as follows:

```
derivationClause = ( "extends:" | "includes:" ) inheritedItems.
inheritedItems   = { [qualifyingList] simpleItem ";"
                   | qualifyingList compositeItem ";" }.
qualifyingList   = "{" inheritedItems "}".
compositeItem    = "(" inheritedItems ")".
```

The basic bracketing rule is that the public methods of `inheritedItems` (whether simple or composite) are qualified by the bracket methods of their `qualifyingList` (if any), but public methods added by other items in the same `qualifyingList` are not. When a client invokes a method of a qualified item, the appropriate bracket method (if any) of each item in its `qualifyingList` is scheduled in turn (from left to right) at the point where the predecessor executes a `body` statement.

Thus a reader-writer synchronised person can be defined as follows:

```
type SynchronisedPerson {
extends:
{RWsync;} Person;
}
```

Multiple qualification is achieved by extending the adjectival list.

```
type SecuredSynchronisedPerson {
extends:
{Monitoring; RWsync;} Person;
}
```

Attributes and qualifiers can be included in the same adjectival list:

```
type StudyingSynchronisedPerson {
extends:
{RWsync; Studying;} Person;
}
```

Here the bracket methods are applied to methods of the target type (i.e. `RWsync` qualifies `Person`) but not to the methods of its attributes (i.e. in this example not to the instance methods of `Studying`).

Adverbial qualification (i.e. qualification of qualifiers) is possible, because items in a qualifying list can themselves have a qualifying list, e.g.

```
type SynchronisedlyStudyingPerson {
 extends: {{RWsync;} Studying;} Person;
}
```

Here the methods of `Studying` are synchronised, but not those of `Person`.

Assuming that `Employed` is another **`for`** `Person` attribute type, the methods of `Studying`, `Employed` and `Person`, are all synchronised by `RWsync` in this example:

```
type SynchronisedStudyingEmployedPerson {
 extends:
 {RWsync;} ({Studying; Employed;} Person;);
}
```

Here `Studying`, `Employed` and `Person` are grouped to define a `compositeItem` qualified by `RWsync`. To synchronise the methods of `Studying` and `Employed` (but not those of `Person`) with the same qualifier, i.e. using the same synchronising variables, the following definition can be used:

```
type SynchronisedlyStudyingEmployedPerson {
 extends:
 {{RWsync;}(Studying; Employed;);} Person; }
```

while in the next example each attribute (but not `Person`) is synchronised, but using separate synchronisation variables:

```
type SyncStudyingSyncEmployedPerson {
 extends:
 {{RWsync rws1;} Studying; {RWsync rws2;} Employed;} Person;
}
```

Chapter 11 Co-Types

The only kinds of methods which can appear in a type definition and its implementations are instance methods, callout methods and bracket methods, together with a single constructor per implementation. Other OO languages often support additional kinds of methods, e.g. binary methods, static methods, and also multiple constructors for a type.

To provide equivalent functionality in a clean and simple manner, Timor has introduced a new concept, called *co-types*[44]. A more detailed description of this concept and of related ideas, including a more substantial motivation for introducing co-types, appears in [17, 18]. The identifier of a co-type is the identifier of its base (partner) type followed by the ampersand symbol & and a co-type suffix. A base type can thus have different co-types, which each has a different co-type suffix.

1 The Basic Structure of a Co-Type

A co-type can *expand* its basic type by providing the equivalent of OO static methods in its instance section. It can also include several additional sections, including the following. A section introduced by the keyword **`binary`** is used to add binary methods to the type. The further section, introduced by the keyword **`maker`**, allows the programmer more flexibility by adding new makers, i.e. implementation independent constructors which build upon the implementation dependent constructors introduced in section 4 of chapter 3. Binary methods and makers are in most respects normal instance methods. Only co-types can have these sections. The order of sections is not important, and multiple sections of the same kind can appear in a type definition.

A type can have zero or more co-types. Here is an example of a co-type

[44] The PhD of my former student Prof. Dr. Axel Schmolitzky played a significant role in the formulation of the idea of co-types, see [3].

declaration, `Person&s`, which is a co-type for the type `Person` with a co-type suffix `s`. This suffix is typically used for makers and binary sections.

```
type Person&s expands Person {
instance: /* This section contains the equivalent
             of static methods for Person */
 enq Int instanceCount();
 /* returns the number of created instances
    of the type Person */
 op void changeName(Person*** p; String newName)
                                  throws NullPtr;
 // changes name of an existing Person (e.g. after marriage)
maker:
 op Person init(String name, address; Date dateOfBirth)
                               throws InvalidParams;
 /* carries out consistency checks and where appropriate
    creates a Person value, initialises name, address and
    date of birth and increments the count of instances */
binary:
 enq Boolean equal(Person*** p1, p2) throws NullPtr;
 // compares two Person instances
 enq Person*** older(Person*** p1, p2) throws NullPtr;
 /* returns the older of two Person instances; if they are
    the same age to the day then p1 is returned */
}
```

2 The Maker Section

Makers are methods which provide a substitute for more conventional constructors. In contrast with the constructors which appear in implementations (with parameters oriented to an individual implementation of a type[45], see chapter 3 section 4), makers have application-oriented parameters which can, for example, initialise various application related variables.

A maker might initialise an instance of the type "from scratch" (as in the example) or it might for example create an instance of the type by, say, merging information from other instances of the type (possibly with different implementations), or simply convert from one implementation to another.

To carry out its purpose, a maker typically selects an implementation of the

45 For example a constructor for a particular implementation of a list might provide a maximum length parameter and use this to initialise an array, while a constructor for a linked list implementation does not require such a parameter.

type (with or without parametric advice from the user), calls its constructor and then initialises some values, e.g. by invoking instance methods of the new instance or by using information from other objects passed as parameters.

When makers call constructors they can choose a suitable implementation and the user of the type (and co-type) does not have to be concerned about the definitions of constructors, which may vary from implementation to implementation. The result type of a maker is always a result of the expanded type. Typically they return a *value* of their base type, since it is easy for the user to convert this into an object reference or a module capability (using the appropriate keyword **`new`** or **`create`**). However, this is only a convention.

If makers are defined for a type, then these are the *only* methods which can invoke constructors of the implementations of the expanded type. If a type has no makers in any of its co-types the basic constructors for its implementations can be invoked without restriction. Thus the (optional) existence of makers always implies control of a type. We therefore refer to a co-type which contains makers as a *controlling co-type*.

If a maker exists for an expanded type which is defined to contain "constants"[46], then only the makers and other methods of the co-type (e.g. co-type instance methods designed to allow controlled changes) can invoke the **`op`** method for setting the constant. However, a controlling co-type is not restricted to doing this only once, which means that it can for example make controlled changes to "constants" after the initialisation phase, e.g. to correct errors, or to change a surname after marriage. (In this sense Timor does not support genuine constants, except where the type has no controlling co-type.)

3 The Instance Section

Instance methods follow the same rules in co-types as in other types. In the context of a co-type they usually access instance data structures which typically serve a function similar to class data in conventional object oriented languages. In this case they will typically not have parameters of the expanded type.

However instance methods can be used to provide other useful co-type functions, in which case they may have a parameter of the expanded type. For example a method which allows a final abstract value of the expanded type (e.g. a surname of a newly married person), which has to be modified for a particular instance (see the previous subsection on makers), needs an actual parameter indicating to which instance the new value applies.

[46] see chapter 5 sections 4 and 5.

4 The Binary Section

This section provides the application with the possibility of declaring binary methods, without creating a number of problems that can occur in normal OO languages [26]. From the Timor standpoint a binary method typically receives parameters of the expanded type and uses their instance methods to carry out the binary task, e.g. to compare the instances. Usually they return a boolean result. They may not return a result of the expanded type; that can be achieved in the **maker** section.

A binary method must have at least two parameters of the expanded type, or a compile time error occurs. It is recommended that these be declared as handles for the expanded type, thus ensuring that separate binary methods do not have to be written for accessing instances of the type which have different (or even mixed) modes. Although it will generally not be necessary, an implementation of a binary method (like any other method) can if necessary use a cast statement to determine the actual modes of the parameters passed to it.

5 A Simple Co-type Implementation

A co-type, like any other type, can have multiple implementations. Here is a simple example of an implementation.

```
impl Person&s::Impl {
state:
 Int count = 0;
instance:
 enq Int instanceCount() {return count;}
 op void changeName(Person*** p; String newName)
                                            throws NullPtr {
  if (p == null) throw new NullPtr ();
  p.name = newName;
 }
maker:
 op Person init(String name, address; Date dateOfBirth)
                                  throws InvalidParams {
  if (dateOfBirth.year < 1900) throw new InvalidParams();
  count++;
  Person tempP = Person::Impl();
  tempP.name = name;
  tempP.address = address;
  tempP.dateOfBirth = dateOfBirth;
  tempP.spouse = null;
```

```
   return tempP;
 }
 binary:
  enq Boolean equal(Person*** p1, p2) throws NullPtr {
   if (p1 == null || p2 == null) throw new NullPtr();
   if (p1.name == p2.name &&
                      p1.dateOfBirth == p2.dateOfBirth)
    return true;
   else return false;
  }
  enq Person*** older(Person*** p1, p2) throws NullPtr {
   if (p1 == null || p2 == null) throw new NullPtr();
   if (p1.dateOfBirth.year < p2.dateOfBirth.year)
    return p1;
   if ((p1.dateOfBirth.year == p2.dateOfBirth.year) &&
       (p1.dateOfBirth.month < p2.dateOfBirth.month))
    return p1;
   if ((p1.dateOfBirth.year == p2.dateOfBirth.year) &&
       (p1.dateOfBirth.month == p2.dateOfBirth.month) &&
       (p1.dateOfBirth.day <= p2.dateOfBirth.day))
    return p1;
   return p2;
  }
 }
```

6 Accessing Parameters at the Implementation Level

The co-type methods above are able to work independently of the implementations of their parameters, which might be different implementations of the same type. This is possible because the methods access their parameters using the public methods of their parameter types. However, this may not always be appropriate, for example when files (especially large files) have to be converted from one format to another.

For example a maker for video files of a type `MpgFilm` might be designed to convert films from one *mpg* format (i.e. implementation) to another, e.g. from *mpeg-2* format to *mpeg-4* format. To do this efficiently the maker needs access to the implementation of its parameter. One way of achieving this would be to define a maker as follows:

```
type MpgFilm&s expands MpgFilm {
maker:
 enq MpgFilm** convertToMpeg4(MpgFilm::Mpeg2** mpg2)
```

```
                                               throws InvalidParams;
 }
```

To attempt such a conversion on the basis of semantic routines such as play, fast forward, etc. would obviously be unreasonable.

The compiler cannot check at compile time whether the actual parameter has the specified implementation, but it can insert code to check this at run-time and if necessary raise a standard run-time error.

There is no problem in having further makers which also convert other formats to mpeg-4, e.g.

```
 enq MpgFilm** convertToMpeg4(MpgFilm::Mpeg1** mpg1)
                                               throws InvalidParams;
```

Where multiple methods are defined with the same name and the same parameter types but with varying implementations, the compiler must generate run-time code to distinguish between their parameters and cause the appropriate method to be invoked. If an appropriate method is not available, a standard run-time error is generated, except in the case where a method is also available which simply defines the parameters in question by their type (without naming an implementation). In this case this method is chosen as a default. This technique is not confined to makers; it can be particularly useful for example in binary methods to compare parameters with the same type but different implementations.

Sometimes conversion programs are needed to produce a file of a quite different type, which from the viewpoint of type definitions is barely related. In the video environment conversion from *avi* format to *mpg* format (and vice versa) would be a case in point. To cite another example, in conventional systems many kinds of files can be converted to PDF format. This is a further case where restricted access via semantic routines would be a futile exercise.

The definition of parameters in terms of their implementations can also be applied in such cases, but the use of the technique is restricted to co-types (makers for conversions and binary methods for comparisons), e.g.

```
 type Pdf&s {
 maker:
  enq Pdf** msWord2010convert(Word::Word2010** word2010)
                                               throws InvalidParams;
 }
```

The definition of parameters in terms of implementations is a deliberate violation of the information hiding principle, and is only permitted in co-type makers and binary methods[47]. (In ModelOS implementations can use free capability parameters, discussed in [6][48], to access the content of a file.)

7 Derivation and Adjustment of Co-Types

Just as normal types (e.g. `Person`) can have subtypes (e.g. `Student`, `Tutor`, etc.) so also can co-types, e.g. which add new methods to those defined in the original co-type. Since co-types are structurally just types like other types (except that they can have special sections for methods such as makers and binary methods), the rules for creating and using subtypes are the same as those for normal types.

If a type which has a co-type also has subtypes, an *adjustment hierarchy* can be (partially) automatically created which provides a parallel hierarchy for their co-types, as is illustrated in Figure 11.1.

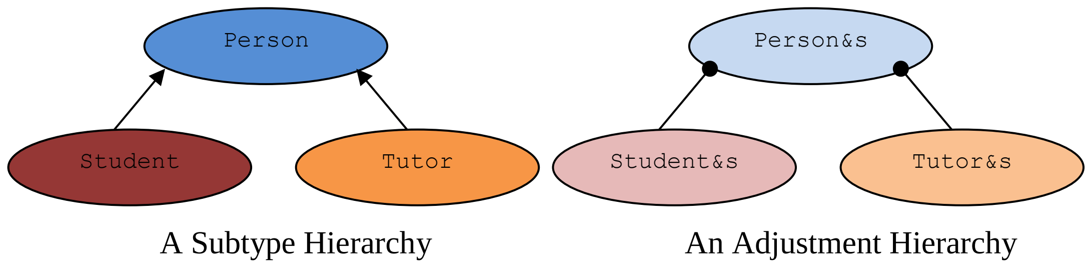

Figure 11.1: Parallel Hierarchies for Subtypes and their Co-Types

An adjustment hierarchy does *not* consist of subtypes. However, it has a number of advantages, including the following[49].

1) Input parameters and return types in the corresponding methods of an adjustment hierarchy can be safely changed covariantly.
2) An adjustment hierarchy can help the co-type designer to ensure that all cases are covered, because it provides a systematic approach which allows methods to be predefined, and these are available by definition even for subtypes that are added later in the subtype hierarchy. This is particularly helpful when co-types are designed as components which can be added to different systems.
3) For the application programmer the existence of an adjustment hierarchy can guarantee that where certain methods exist in a co-type for an expanded

[47] In his PhD thesis Schmolitzky restricted the use of this technique to parameters which, in Timor terms, have different implementations of the type being expanded in the co-type.

[48] see Chapter 18 section 8, Chapter 19 section 13.5 and chapter 24, section 5 of [6].

[49] The following lists of advantages and differences and differences originally appeared in [18], which has also served as a basis for the text of the sections 7 to 14 of this chapter.

type, similar methods will exist in co-types for all the subtypes of the expanded type.

4) Implementations of co-types can be re-used in implementations of other co-types in the adjustment hierarchy.

Here are some of the differences between a subtype hierarchy and an adjustment hierarchy, as illustrated from Figure 11.1:

- Whereas `Student` and `Tutor` are subtypes of `Person`, the adjusted types `Student&s` and `Tutor&s` are not subtypes of `Person&s`. Thus an instance of type `Student` can be assigned to a variable of type `Person`, but an instance of type `Student&s` cannot be assigned to a variable of type `Person&s`.
- A co-type functions only for its corresponding expanded type. Thus `Student&s` cannot act as a co-type for `Person` or `Tutor`.
- Whereas a subtype hierarchy is open-ended and is extended explicitly, an adjustment hierarchy has a parallel set of nodes corresponding to those in the subtype hierarchy, starting at the node where a new co-type is explicitly defined.
- Because co-types in an adjustment hierarchy are not subtypes of their adjusting ancestors, methods in a higher level co-type need not appear in corresponding lower level co-types.
- It is possible for example to define an additional co-type `Student&s2` for `Student`, and co-types derived by adjustment from `Student&s2` will not apply to `Person` or `Tutor` or their subtypes, but only to subtypes of `Student`.

8 Syntactic Features Limited to Co-Types in an Adjustment Hierarchy

Two additions are available to enhance the functionality of adjustment hierarchies.

i) Where the system designers decide that methods of a co-type section (e.g. **`instance`**, **`binary`**, **`maker`**) *must* appear in each successor co-type in the hierarchy, the name of the section is preceded by the keyword **`predefines`**. However, it is a matter of the system designer's judgement to determine which methods are predefined in this way, and it is possible to have section names which are not preceded by this keyword.

ii) The keyword **`TheType`** is used to indicate specific type names where covariance occurs. The modes of the parameters are indicated using the normal asterisk notations. Not all parameters need be defined as covariant.

It is an orthogonal issue to determine which methods (if any) should be declared to be **`protected`** (see chapter 3 section 8).

9 Covariant Makers

Although abstract types do not have makers (because an abstract type cannot be instantiated) it is possible to have **`predefined maker`** sections (but not normal **`maker`** sections) in a co-type for an abstract type. The purpose of these is to signify that certain makers *must* appear in the co-types for concrete subtypes of the abstract type.

10 Covariant Instance Methods

The instance methods of a co-type can have parameters of the expanded type, in which case these can be declared using **`TheType`**. However if the instance methods have an equivalent role to that of class methods in conventional OO they will not normally have parameters of the expanded type. But this does not prevent them from being defined in a **`predefined instance`** section (even for an abstract type).

11 Modifying Co-Type Definitions

The above descriptions allow a compiler automatically to define members of a co-type adjustment hierarchy based on the co-type for the base type. However, it may sometimes be appropriate to modify such automatically defined co-types. To indicate that such modifications should be made the keyword **`adjusts`** (analogous to the keywords **`extends`** or **`includes`** for normal types) can be used.

In this case methods which need to be redefined appear in a **`redefines`** section. If the signature does not change the redefined methods replace the corresponding methods from the adjusting type (cf. overriding). Changes which only involve the meaning of **`TheType`** require no redefinition.

New methods can be added in the usual way. If a new method has the same identifier as that of a predefined method but with a changed signature (other than changes involving only the meaning of **`TheType`**) then this method is considered to be a new method (analogous to overloading). In this case both the predefined method and the new method are present in the adjusted type.

Where signatures of predefined methods need not be changed (except for covariant adjustments), neither the relevant section nor its methods need (but they can) be included in the redefinition of the co-type.

12 Merging Co-Type Methods which Result from Diamond Inheritance

This issue primarily concerns diamond inheritance resulting from an abstract supertype, such as arises in the Timor Collection Library and is handled in a similar way to the merging of methods in a subtyping hierarchy.

Type Adjustment Rule 1: If in an adjusted type multiple methods which result from a common adjusting predecessor and which have the same signature (in this context including parameters defined using the keyword **`TheType`**), they are treated as a single method (unless they have return types which differ from each other, in which case a compile time error arises). According to this definition, makers, binary methods and instance methods can all be merged.

Type Adjustment Rule 2: If the definitions of such methods differ (i.e. if one or more of them has been redefined differently from the definition in their closest common predecessor), they must also be listed in a **`redefines`** clause in the type being defined.

Rule 2 in effect requires that conflicting definitions are clarified. Where a definition in one of the ancestors can be used in the new type, this can be signalled by the use of the keyword **`from`** followed by the name of the appropriate co-type.

13 Merging Multiply Adjusted Co-Types for Parts

In this case the principles basically follow *mutatis mutandis* those used in defining the types themselves (see [11]). In the co-type the keyword **`adjusts`** is used instead of **`extends`** or **`includes`**, and in the case of repeated inheritance multiple co-type methods for a repeated expanded type are not required.

14 Implementing Adjustment Hierarchies

The rules for implementing adjustment hierarchies follow a similar pattern to those for implementing types. A fundamental difference arises as a result of covariant adjustment of parameters for re-use variables defined using the keyword **`TheType`** (see chapter 8). The solution is to indicate that a re-use variable which requires covariant adjustment is indicated by a double hat symbol. For example if a re-use variable based on the TCL implementation of the co-type **`Collection&s`** is used in the implementation of a **`Bag`** co-type **`Bag&s`**, in order that the covariant adjustment occurs it is implemented as follows:

```
impl Bag&s::Impl {
state:
^^Collection&s collections = Collection&s::Impl();
   /* an adjusted re-use variable is indicated by a double
      hat symbol */
}
```

In this example each occurrence of **`TheType`** in `Collection&s::Impl` is considered to mean `Bag`.

The keyword **predefines** is not normally used in implementations, because the compiler does not automatically produce adjusted implementations of successor co-types. However, it can appear in implementations of co-types in the context of makers, when a co-type is being defined for an abstract base type (which of course has no makers), but it is anticipated that the co-type implementation will be used as a re-use variable in the implementations of concrete subtypes. An example of this appears in chapter 13 section 3.2.1 in the implementation of `Collection&s`.

15 A Further Example of Co-Types

The basic idea behind co-types is that they can provide a standard way of adding ancillary methods to a basic type (i.e. the expanded type) without directly affecting the definition of the expanded type itself. This approach provides a modular framework which allows new components to be defined (and have multiple implementations) and easily added to application systems. Makers and binary methods are obvious examples where this approach is useful, but these are not the only examples.

15.1 Co-types for Standard Input-Output Operations

Here is a further example which illustrates the versatility of this idea and at the same time illustrates how input-output methods can be added to types in a modular way. We begin by defining a *view* `StandardIO`, as follows:

```
view StandardIO expands TheType {
 predefines maker:
  op TheType fromString(String s);
 predefines inout:
  enq String toString(TheType t);
  ...
}
```

Notes:

1. The use of the keyword **expands** indicates that this view can only be extended by or included in a co-type.
2. In co-types the **expands** keyword is normally followed by the name of the specific type being expanded, but since in the context of a view which is designed for use in any co-type a specific type cannot be nominated, the keyword `TheType` is used instead.
3. An implementation can be defined for a co-type view, but of course this can only be fully compiled in a specific co-type context.
4. The keyword **inout**, like **maker**, etc. is the name of a co-type section.

The view `StandardIO` has an implementation which can, but need not, be re-used in implementations of co-types which inherit its methods:

```
impl StandardIO::Impl {
 predefines maker:
  op TheType fromString(String s) {
   if (s == null) throw new NullEx();
   return null;
  }
 predefines inout:
  enq String toString(TheType t) {
   if (t == null) throw new NullEx();
   return "undefined";
  }
}
```

Here is an example of a co-type `Person&io`, which is an input-output co-type for the type `Person`:

```
type Person&io expands Person {
extends: StandardIO;
}
```

An implementation might be coded along the following lines:

```
impl Person&io::Impl expands Person {
 predefines maker:
  op TheType fromString(String s) {
   if (s == null) throw new NullEx();
   Person p;
   p.name = ... // select name from the input string s
   p.address = ... // select address from s
   String sdate = ... // select substring with date from s
   p.dob = date&io.fromString(sdate);
   return p;
  }
 predefines inout:
  enq String toString(TheType t) {
    if (t == null) throw new NullEx();
    return "Name is "+ t.name + "Address is " + t.address +
           "Date of Birth is " + date&io.toString(t.dob);
  }
}
```

This approach differs substantially from the Java approach, which adds such methods as `toString` by making them methods of the general Java supertype `Object`. The Timor general supertype `Handle`, by contrast, has no methods. Instead Timor opts for the more modular, more flexible and more extensible approach illustrated.

As a further illustration of this flexibility we define an abstract type `Shape` and then expand this in a co-type `Shape&io`.

```
abstract type Shape {
 instance:
  Int colour; // all shapes have a colour
  Int coordinate1, coordinate2;
  // all shapes have at least two coordinates
}
```

Now we declare a co-type for this. The keyword `TheType` refers to the expanded type (i.e. `Shape`), and can be covariantly adjusted in co-types for subtypes of `Shape`.

```
type Shape&io expands Shape {
extends: StandardIO;
predefines inout:
 enq void draw(TheType s; Canvas c);
  // draws a single shape on a canvas
 enq void drawAll(Collection<TheType>*** ct; Canvas c);
  // draws a collection of shapes on a canvas
}
```

The `draw` and `drawAll` are methods for drawing instances of `TheType` which are inherited in concrete subtypes; they cannot actually draw a `Shape` object, because `Shape` is an abstract type and has no objects in its own right. The aim is that methods in the appropriate subtype can draw concrete examples of `Shape`, such as rectangles and circles. It is also useful to be able to draw collections containing different shapes which together make up a drawing.

Why should we want to define I/O methods in co-types? The answer lies in the modularity which can be gained and the flexibility which it gives. For example with the simple `draw` method defined above, we can design this method (for subtypes of `Shape`) in an endless number of ways. For different applications we might want to provide a simple shape, or we could write the name of the shape (or a user-defined name indicating what the shape represents in a particular picture) inside the shape, or over it or underneath it, etc. By detaching the drawing methods from the definition of the base type itself, we gain considerable flexibility, because we can, if we choose, have multiple co-types which provide different ways of representing the same shape.

It should be recalled in this respect that different co-types for the same type can appear and be concurrently invoked in a single system or application, and that these need not all contain similar methods. A co-type containing only I/O methods, for example, is quite normal. It need not also contain binary methods, etc. The latter are best held in a separate co-type.

15.2 Adjusting Input-Output Methods in Co-types

We now define two subtypes for the type `Shape`:

```
type Circle{
 extends: Shape;
 instance:
  Int radius;
}
type Rectangle{
 extends: Shape;
 instance:
  Int length, width;
}
```

According to the co-type adjustment rules, a co-type is automatically generated for each subtype, and this can then be manually modified to add new methods and to provide appropriate implementations of the co-type methods. Assuming that the co-type `Shape&io` extends `StandardIO`, the co-type `Circle&io` is automatically generated as follows:

```
type Circle&io expands Circle {
 adjusts: Shape&io;
 // The following maker is automatically adjusted
 predefines maker:
  op TheType fromString(String s);
predefines inout:
 // The following inout methods are automatically adjusted
 enq String toString(TheType t);
 enq void draw(TheType s; Canvas c);
  // draws a single circle on a canvas
 enq void drawAll(Collection<TheType>*** ct; Canvas c);
  // draws a collection of circles on a canvas
}
```

An analogous co-type for rectangles, called `Rectangle&io`, also exists automatically and methods can be correspondingly added. Implementations of the types and their co-types are not shown, as these would illustrate nothing new.

15.3 Using Inout Methods

The I/O methods in the above example are based on similar methods described in Bracha's tutorial on generics in Java [27], though there the `draw` method is defined as an instance method in each shape, and a class `Canvas` contains a further `draw` method and a `drawAll` method which both invoke the draw methods in the individual shapes. Bracha uses the example to argue for the use of upper-bounded wildcards, indicating that it makes sense to have a list of shapes which together form a drawing and can be passed to `drawAll`.

In the following example a list of shapes `shapeList` is assumed to exist which contains circles and rectangles, and this is passed to the `drawAll` method of a `Shape&io` co-type:

```
shape&io.drawAll(shapeList, c);
```

The issue which we now address is how `drawAll` can be implemented. The question arises because the `draw` routine is not a normal instance method defined in the expanded types but appears instead in their co-types. Here is a possible implementation:

```
enq void drawAll(Collection<TheType>*** ct; Canvas c){
  /* draws a collection of shapes on a canvas.
     Since this is code in an implementation of
     Shape&io, TheType here is Shape */
  for (TheType t in ct) {
    cast (t) as {
     (Rectangle rect) {rectangle&io.draw(rect, c);}
     (Circle circl) {circle&io.draw(circl, c);}
    }
  }
}
```

In this example `rectangle&io` and `circle&io` are variables to which instances of the co-types `Rectangle&io` and `Circle&io` respectively have been assigned. This follows the Timor practice that for each co-type included in a program a co-type variable normally exists with an identifier which is the same as the co-type identifier, except that it begins with a small letter.

While the above solution works, it is not particularly attractive, because the code of the cast statement has to be modified whenever a new shape is introduced or an existing shape is deleted, which is not very modular.

Here is another approach. The method `draw` (but not `drawAll`) is placed not in the co-type but as an instance method in each shape type/subtype. For example the method could be defined in `Shape` (as distinct from `Shape&io`) as:

```
enq void draw(Canvas c);
  // draws a single shape on a canvas
```

This would be inherited in all subtypes, and an implementer would have to provide code for the corresponding shape. It would then be possible to implement `drawAll` in `Shape&io` as follows:

```
enq void drawAll(Collection<TheType>*** ct, Canvas c){
    for (TheType t in ct) t.draw(c);
    }
```

This code also works (thanks to dynamic method binding), but like the previous solution, it also has a drawback. In this case the method which actually draws a shape must be included in the description of the shape subtype itself, which defeats one of the aims of including `inout` routines in co-types, namely that multiple co-types can co-exist and can be used flexibly to define different ways of representing each shape.

We now pursue an approach which combines the advantages and eliminates the disadvantages of both solutions above. It is based on the simple idea that what one would like to achieve is something like the second solution, but instead of invoking a method of the expanded type a method of its co-type is invoked.

To achieve this we extend the syntax of Timor as follows: if `t` is a variable (or pseudo-variable) of some type `T`, then the pseudo-variable `t&io` (or of course a pseudo variable `t&s` where we are concerned with a co-type `T&s`, etc.) is a pseudonym for the co-type variable corresponding to the *actual* type of t. Hence if the `draw` method is defined as a predefining method in the `inout` section of `Shape&io`, then we can modify the second solution above to read:

```
enq void drawAll(Collection<TheType>*** ct, Canvas c){
    for (TheType t in ct) t&io.draw(c);
}
```

Notice that in this example we might be executing `drawAll` as a method in `Shape&io`, where the expanded type (i.e. `TheType`) is `Shape`, such that `ct` is a collection of shapes of various sorts. Since `Shape` itself is an abstract type, the actual shapes in `ct` must be concrete subtypes of `Shape`, e.g. rectangles and circles. Hence although `t` is formally declared to be of type `TheType`, i.e. in this case Shape, its *actual* type might for example be `Circle` or `Rectangle`. Hence an invocation of the draw method on `t&io` refers to the co-type variable of the actual type, e.g. `circle&io`.

This technique involves dynamic binding of co-type methods. It relies on the following premises:

- The methods which are invoked must be predefined methods in the adjusted co-types.
- For each subtype of the type in question (here `Shape`) there is a corresponding co-type, and these form an adjustment hierarchy of co-types which are automatically named with the help of the `&` character. (The suffix following the `&` symbol indicates which co-type adjustment hierarchy is involved. This must be the family of the co-type currently being defined.)
- The name of the method invoked using the pseudo-variable is the name of a predefined method in the co-type which is currently being compiled.

All of this can be checked by the compiler simply from the information which it has about the co-type (implementation) currently being compiled.

16 Access to Co-Types and other Components

When a module (in the Timor and ModelOS sense) and all its components have been completely compiled and integrated into a single unit, and the module is therefore ready to be used, the supporting software responsible for this (somewhat akin to a linker but with different duties) must ensure that all the co-types (and adjusted co-types) which can be invoked are also available (automatically) in the module. We do not define here how this happens but it is important that the compiler leaves sufficient information available in "linking" and software library files to make this possible. This involves locating those co-types which are actually used in the module; the software can recognise this from the names of co-types, since these are recognisable by their suffixes which follow the names of the base types. If a co-type or adjusted co-type or required implementation thereof does not exist for the system in question (e.g. in a library of co-types), then the software must draw this to the attention of the programmer responsible for the module.

Since one of the basic aims of Timor (and ModelOS) is to have standard libraries of commonly used components at all levels, it should frequently be the case that standard components (even for "small" objects such as `Person`, `Date`, etc. and for their commonly used co-types) already exist for programmers. Each automatically produced actualised co-type automatically has a variable by which its methods can be invoked. The name of this variable is the name of the actualised type followed by the name of the type which it expands followed the co-type suffix; for example for type variables actualised to `SortedSet<Person*>` there could be automatic co-type variables such as `personSortedSet&s`, `personSortedSet&io`, etc, depending whether these are actually used in the module.

Chapter 12
Generic Types and Implementations

Timor provides two generic features for use in programs. The first, which to some extent resembles genericity in other OO languages, allows a generic type name to represent actual type names in type definitions and implementations. This is especially useful for defining the TCL, since it allows the elements in a collection to be defined generically but then instantiated as actual collections types, e.g. as a *set* of the type `Person`, written `Set<Person>`[50]. The identifiers of generic types are distinguished from other identifiers in that the first two characters must be capital letters and all further letters must also be capitals, though other symbols are permitted. The definition of a set of generically defined elements might be `Set<ELEM>`.

The second generic form allows parameters for generic type constructors to be defined as generic functions. This second form of genericity is introduced in chapter 14.

In the following it is assumed that the type `Person` is defined as a "record" based on abstract variables (see chapter 5), as follows:

```
type Person {
instance:
 String name, address, dob;
 Int passportNum;
 Int intDialCode, areaDialCode, telephoneNum;
}
```

1 Generic Templates

Timor supports genericity in the sense that a generic definition provides a pattern for a number of types or their implementations. Such patterns are called

[50] Detailed generic type definitions and their implementations have been included as examples in chapter 13.

templates. In order to produce actual types and implementations, a template must be *actualised*[51]. A template itself plays no role at run-time, in contrast with the entities actualised from it.

A template consists of a *template header* and a *template body*. It has a *template identifier* and it may use both normal and generic identifiers in its definition. The template identifier is not a generic identifier; it must be unique in the same sense that type and implementation identifiers are unique.

A template header consists of the keyword **`template`** followed by a set of template parameters bracketed by the bracketing pair < and >. A template body is a normal type or implementation definition which includes the template identifier and may contain generic identifiers.

2 Type Templates

Here is an example of a template for producing types, known as a *type template*:

```
template <ELEM>
// ELEM is a generic identifier for a type parameter
abstract type Collection {
/* Collection is the template type identifier. This example
   is a shortened version of the abstract supertype of
   collections in the Timor Collection Library (TCL) */
instance:
enq Int size();  // returns current number of elements
op void clear(); // removes all elements in this collection
enq Boolean contains(ELEM e) throws NullEx;
// returns true if e is an element in this collection
op void insert(ELEM e) throws DuplEx, NullEx;
/* a general method to insert elements; a DuplEx exception or
  a NullEx exception may be thrown by appropriate subtypes */
op void remove(ELEM e) throws NullEx, NotFoundEx;
/* removes e (at most once) if this is contained in the
   collection */
}
```

In this example the *template identifier* is `Collection` and the single template parameter is `<ELEM>`, where `ELEM` is a generic identifier and **`type`** indicates what kind of template it is. The keyword **`abstract`** indicates that in this case the type defined in the following template is an abstract type.

[51] The term "instantiate" is used for creating instances of types, whereas the term "actualise" indicates the creation of an actual type from a type template.

The *template body* consists of the remaining lines of the example.

3 Implementation Templates

Like normal Timor types, each type template can have more than one implementation. These are defined in *implementation templates*, which also consist formally of a template header followed by a template body. The identifier of an implementation template is the name of the implementation. Here is an example:

```
template <ELEM>
impl List::Impl{       /* List::Impl is the template id; this
   is followed by the code of the generic implementation */
}
```

However, as the template header of an implementation cannot differ from that of its generic type, this can be abbreviated simply to the keyword **`template`**, as follows:

```
template impl List::Impl{
...
// code of the implementation
}
```

4 Actualising Templates

Actualising a generic template consists of substituting an actual type name for each generic parameter which appears in the template header. A variable of an actual type can be declared as follows.

```
List<Person*> personList;
// a variable for a List of Person references
```

Instances of generic types are initialised by invoking a constructor of one of the implementations of the type. Here is an example showing how a constructor for a `List<Person*>` might be called:

```
List<Person*>::Impl();
```

5 Deriving Templates by Single Inheritance

5.1 Deriving Templates by Extension

A type template can be derived by *extension* from some other type template, provided that it does not add any further generic type parameters to those already present in the ancestor. Instances which are actualised from a subtype template can only be assigned to variables of a supertype if

a) no additional type parameters are introduced at the subtype level,

b) the same type parameter arguments are used to actualise both the variable and the instance.

These rules do not preclude defining by *inclusion* a derived type which adds further template parameters, since such derived types cannot be used polymorphically.

Here is an example of a subtype template:

```
template type Bag {
/* without a complete template header as this is inherited
   unchanged from the template of the supertype Collection */
extends: Collection;
redefines:
op void insert(ELEM e) throws NullEx;
 // accepts duplicates, i.e. always inserts e
op void replace(ELEM oldE, ELEM newE)
                           throws NullEx, NotFoundEx;
 /* removes at most one element corresponding to oldE and
    if an oldE is removed, newE is inserted. */
}
```

The behavioural subtyping rules above are provided in order to guarantee type security at compile time. The following example, which attempts to add an additional generic type parameter, illustrates the need for the above rules:

```
template <ELEM, ELEM2>
type DoubleBag {
 // intended to allow a bag to contain two kinds of elements
extends:
 Bag;
instance: // a new instance method
 op void insert(ELEM2 differentE) throws NullEx;
  // accepts duplicates, i.e. always inserts differentE
}
```

If the above definition were valid then according to the normal method overloading rules the new `insert` method would overload the `insert` method in `Bag`. Consequently it would be possible to insert items both of type `ELEM` and of type `ELEM2` into a `DoubleBag`, provided that this is not accessed via a `Bag` variable.

Let us suppose that a programmer correctly does this, e.g.

```
DoubleBag<Person, Animal> myDoubleBag =
                       DoubleBag<Person, Animal>.init();
... // add some Person and some Animal elements to DoubleBag
```

but then assigns the object to a `Bag` variable:

```
Bag<Person> personBag = myDoubleBag;
```

Then he assigns arbitrary elements via the `Bag` variable to a variable of the type `ELEM` of `Bag`:

```
for (x in personBag) {Person p = x; ...}
 // Error: an element of type Animal might be assigned
 // to a Person variable.
```

This is an error which would not be detectable at compile time without the above rules.

5.2 Deriving Templates by Inclusion

These rules do not preclude defining by *inclusion* a derived type which adds further template parameters, i.e. the following definition is valid:

```
template <ELEM, ELEM2>
type DoubleBag {
 // intended to allow a bag to contain two kinds of elements
includes: // This is the fundamental difference!
 Bag;
instance: // a new instance method
 op void insert(ELEM2 differentE) throws NullEx;
  // accepts duplicates, i.e. always inserts differentE
}
```

This definition by inclusion prevents a `DoubleBag` instance from being assigned to a `Bag` variable and thus avoids the error described in section 5.1. The result would be a bag in which some elements are of type `ELEM` and others of type `ELEM2`.

Given a `DoubleBag` called `mixedBag` as follows

```
DoubleBag<Person, Animal> mixedBag =
                        DoubleBag<Person, Animal>.init();
... // add some Person and some Animal elements to DoubleBag
```

the **`for`** statement could be used as follows

```
for (x in mixedBag) {
 cast (x) as {
 (Person p) { /* code which addresses x as p,
                 if the underlying object is a Person */}
 (Animal a) { /* code which addresses x as a,
                 if the underlying object is an Animal */}
 else {throw new ErrorEx}
}
```

A type template derived by inclusion does not exclude the possibility that this can itself have subtype templates derived for example by extension, provided that rules a) and b) are followed.

6 Deriving Type Templates by Multiple Inheritance

6.1 Deriving from Templates with a Common Ancestor Template

If a template is derived from multiple methods which directly or indirectly have a common ancestor template the common methods are merged into a new template according to the normal Timor rules involving a common ancestor (see chapter 7), including the rules provided for overloading and overriding methods.

6.2 Deriving from Templates without a Common Ancestor Template

The following example has been construed to illustrate the principle involved when a type inherits from separate ancestors. It is not part of the Timor Library.

Let us suppose that two forms of queue exist, defined as follows:

```
template<ELEM>
type FrontInserter{
 instance:
  op void insertAtFront(ELEM e);
  op ELEM removeFromBack();
  enq Boolean contains(ELEM e);
  enq Int size();
  op void clear();
}
template<ELEM>
type BackInserter{
 instance:
  op void insertAtBack(ELEM, e);
  op ELEM removeFromFront();
  enq Boolean contains(ELEM e);
  enq Int size();
  op void clear();
}
```

These could be used to create a type `DEQ` (double ended queue) as follows, using the multiple inheritance technique described in chapter 7, section 4:

```
template<ELEM>
type DEQ{
  extends:
    FrontInserter, BackInserter;
}
```

This definition[52] is complete and according to the rules of parts inheritance the new type has two insertion and two removal methods, but a single method `size`, a single method `clear`, and a single method `contains`.

The general rule for inheriting from multiple type templates by extension is that the two (or more) supertype templates must have the same type parameter(s). If these differ, or if one has more generic parameters than the other, then derivation can only be defined by inclusion.

Types which have generic type parameters may not be combined with types which either do not have, or have different, generic type parameters.

7 Generically Defined Views

Views can be defined where appropriate with generic type parameters, but if these are explicitly used to extend or to be included in another type, the same generic type parameter must already be present in that type. In contrast with other types, views which themselves do not have generic type parameters can be extended by or be included with type templates to create a new type.

8 Implementing Generic Types and Code Re-Use

The name of a generic implementation is its type name (which does not include a generic identifier) and the appropriate suffix, e.g. `List::Impl`. This distinguishes it from the implementation identifiers for actualised implementations, e.g. `List<Person*>::Impl`.

An implementation for an actualised type can be separately programmed explicitly. For example, a bit-list implementation of a set of integers only makes sense for `Set<Int>` and `SortedSet<Int>` (or equivalent types based on enumeration types).

Implementations of generic types otherwise follow the normal implementation and code re-use rules.

9 Implementing Generic Co-Types

The code re-use technique cannot be used in its normal form to achieve the re-

[52] It is arguable whether the new subtype is actually behaviourally conform with its supertypes, but that is not something which the language designer should attempt to determine in cases where no obvious type safety rules are compromised. Designers who consider that this example is not behaviourally conform can of course define the subtype template in terms of inclusion rather than extension.

use of code in adjusted co-types, because of the covariant adjustment of `TheType`. But a small modification of the technique, using a double (rather than the normal single) hat symbol, indicates to the compiler that the code of a re-use variable should be covariantly adjusted to `TheType` of the co-type currently being compiled (cf. chapter 11 section 14).

10 Generic Function Parameters

The second generic feature of Timor is called *generic function parameters*. A description of these appears in chapter 14, after the main features of the Timor Collection Library (TCL) have been described. The reason for this decision is that generic function parameters were introduced primarily as an enhancement for the TCL, and are best understood in that context.

Chapter 13
The Basic Timor Collection Types

As was already indicated in Chapter 7 section 6, the Timor Collection Library (TCL) provides a good example of the typing issues associated with multiple inheritance from a common *abstract* ancestor, following the scheme proposed in the doctoral thesis of Dr. Gisela Menger [2], which derives four further abstract types and nine concrete types from the abstract type `Collection`, using two basic criteria. The first concerns the decision whether *duplicate items* are permitted, whether they are simply ignored, or whether an error is signalled when an attempt is made to insert a duplicate. The second criterion is concerned with the *order* of the elements in a collection, i.e. whether they are unordered (as in a mathematical *bag* or *set*), whether they are user ordered (as in a *list*) or whether they are automatically *sorted* according to some criterion, e.g. alphabetically. Table 13.1, which repeats Table 7.1 for convenience, shows all nine concrete types in terms of these criteria.

Collection Type Name	Duplication Criterion	Ordering Criterion
Bag	Allow duplicates	No ordering
Set	Ignore duplicates	No ordering
Table	Signal duplicates	No ordering
List	Allow duplicates	User ordered
OrderedSet	Ignore duplicates	User ordered
OrderedTable	Signal duplicates	User ordered
SortedList	Allow duplicates	Sorted
SortedSet	Ignore duplicates	Sorted
SortedTable	Signal duplicates	Sorted

Table 13.1. The concrete collection types

In addition five abstract types are used to enhance polymorphism. The type hierarchy is shown in Figure 7.1, repeated below as Figure 13.1.[53]

In order to guarantee behavioural conformity all the common methods of all collection types are initially defined in the abstract type `Collection`. Thus it has an (abstract) method `insert`, for example, but this does *not* define

– how an insertion affects the ordering of the collection,

– whether the insertion will be successful if it involves inserting a duplicate,

– whether an exception will be thrown to indicate a duplicate (but it defines an exception `DuplEx` which *might* be thrown).

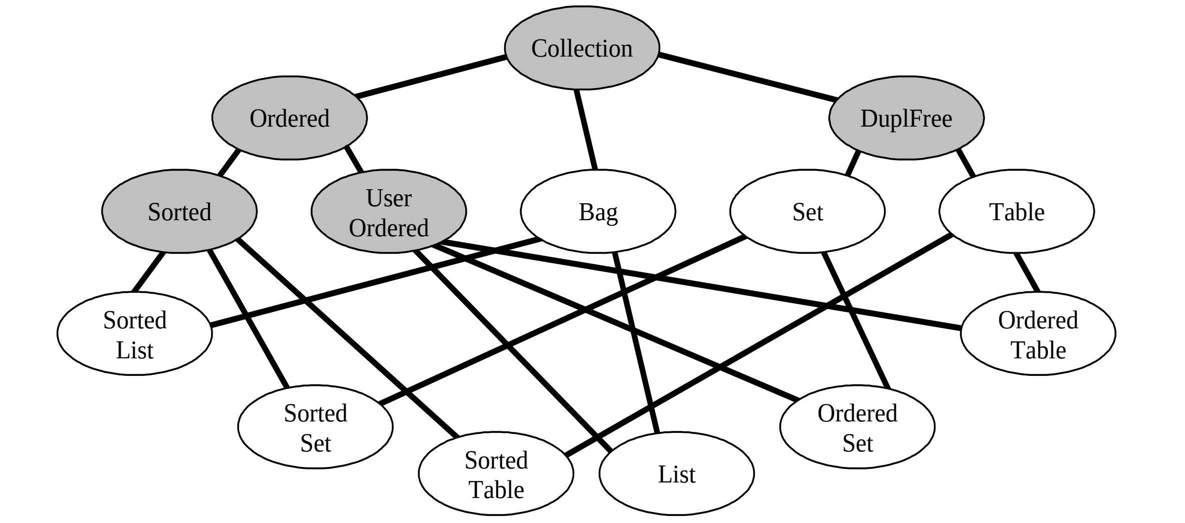

Figure 13.1: Structure of the TCL Collection Types

An abstract type with such non-deterministic methods is designed to allow a maximum of polymorphism. In derived types the actions of the `insert` method are specified more precisely, depending on the node in question. Thus the `insert` method of the abstract type `UserOrdered` defines that `insert` appends the element at the end of the collection (and adds new methods for inserting at other positions) but without defining its duplication properties further. On the other hand the `insert` method of the concrete type `Bag` is defined without specifying ordering, but indicating that duplicates are accepted (with the effect that the exception `DuplEx` can be removed from the `insert` method of `Bag`).

In order to reflect in an abstract way how the items in a collection are handled the TCL makes extensive use of Timor's generic features, using the generic identifier `ELEM` to represent a particular element (item) in a collection.

This chapter first provides a provisional definition of the abstract type `Collection` and the types derived from it. Then an implementation is provided,

[53] The following description is based largely on parts of section 4 of [8].

which illustrates the power and flexibility that can be achieved by separating type inheritance from code re-use. It then illustrates how collection co-types can be defined.

Note:

Although types such as `Queue`, `DoubleEndedQueue`, `Stack`, `BoundedBuffer` and similar (and implementations thereof) belong in the library, definitions and implementations are trivial and are not discussed here.

WARNING 1: The aim of the chapter is not to provide an exact definition of the TCL, but to indicate its main features. A final definition may provide more methods (e.g. by providing methods to remove *all* the elements indicated in the parameters using a single method, or to replace one element by another).

WARNING 2: Parts of the chapter contain detailed code. Since a compiler does not exist at the time of writing, the reader is warned that there may be both logical and syntactical errors in the code which could not immediately be detected. I apologise for such errors.

WARNING 3: This chapter provides a preliminary explanation of collections. It builds up an important basis for understanding the main principles underlying Timor collections, and in that sense is comparable with collections in other object-oriented languages. The chapter also provides a basis for a more advanced collection concept, e.g. in commercial environments.

1 The Collection Types

This section provides type definitions for the standard collection types, beginning with a definition of the abstract type `Collection`.

1.1 The Abstract Type Collection

The generic type `Collection` forms the basis for the remaining collection types:

```
template <ELEM>
library abstract type Collection {
/* Collections may accept or reject duplicates.
   They may or may not be ordered. */
instance:
 enq Int size();
  // returns the number of elements in the collection
 op void clear();
  // removes all elements
 enq Boolean contains(ELEM e) throws NullEx;
  /* returns true if e is in the collection. The equality
```

```
      test uses the operator ==. Chapter 14 describes a more
      flexible mechanism for defining equality. */
  enq Int occurrences (ELEM e) throws NullEx;
  // returns the number of elements == e.
  op void insert (ELEM e) throws NullEx, DuplEx, BoundsEx;
   /* inserts element e. DuplEx may be thrown by subtypes. */
  enq ELEM get();
  /* gets and returns an element as determined in the
     subtypes.
     If the collection is empty the return value is null. */
  op void remove(ELEM e) throws NullEx, NotFound;
  // removes one element == e.
  op void loopInit ();
  /* This operation must be called before an interation
     begins */
  /* the implementation of loopInit and of getNext is
     described in section 4.8, where it is explained how the
     implementation "knows" what is the next element to be
     returned. */
 enq ELEM getNext() throws EmptyCollection, IterationComplete;
  /* returns the next element in the List. */
 } // end of type Collection
```

1.2 The Concrete Type Bag

```
 template <ELEM>
 library type Bag {
 extends:
  Collection;
  // Bag accepts duplicates and is unordered.
 redefines:
  enq ELEM get();
  /* returns an arbitrary element. For an empty bag it
     returns null. */
  op void insert(ELEM e) throws NullEx, BoundsEx;
  /* Since duplicates are permitted the exception DuplEx is
     removed from the insert method.
 } // end of type Bag
```

1.3 The Ordered Types

There are three abstract types which are concerned with the ordering of elements.

1.3.1 The Abstract Type Ordered Collection

The methods in this type are added for all ordered collections, regardless of the ordering technique.

```
template <ELEM>
library abstract type OrderedCollection {
/* OrderedCollection assumes an order based on the operator
   ≤. The positions are numbered from 0 to the end. A more
   flexible way of determining the order of elements is
   described in the next chapter. */
extends:
 Collection;
redefines:
 enq ELEM get();
 /* returns the first element in the collection, unless the
    element returned is null (in the case of an empty
    collection) */
instance:
 enq ELEM getAtPos(Int pos) throws BoundsEx;
 /*  returns the element at the indicated position pos */
 op Boolean removeAtPos(Int pos) throws BoundsEx;
 /* removes the element at the indicated position. */
}
 // end of type OrderedCollection
```

1.3.2 The Abstract Type UserOrderedCollection

The methods in this type allow the user to organise the ordering of elements by position.

```
template <ELEM>
library abstract type UserOrderedCollection {
// For UserOrderedCollection the user determines the position
extends:
 OrderedCollection;
redefines: // The following instance methods are redefined.
 op void insert(ELEM e)
          throws NullEx, DuplEx, BoundsEx;
 // appends e at end of collection
instance:
 op void insertAtPos(ELEM e, Int pos)
         throws NullEx, DuplEx, BoundsEx;
 /* inserts e at the specified position. The elements which
    were previously at the specified position and higher
    numbered  positions automatically have the next higher
    positions; position is defined as an integer,
    starting at 0 for the first position */
}
 // end of type UserOrderedCollection
```

1.3.3 The Abstract Type SortedCollection

Collections of this type are automatically sorted. The operator ≤ is used to determine the order.

```
template <ELEM>
library abstract type SortedCollection {
extends:
 OrderedCollection;
redefines:
 op void insert(ELEM e)
        throws NullEx, DuplEx, BoundsEx;
 /* uses the operator ≤ to determine the position at which
    the element e is inserted */
}
 // end of type SortedCollection
```

1.3.4 The Concrete Type List

A list may contain duplicates and is manually ordered.

```
template <ELEM>
library type List { // manual ordering, allows duplicates
extends:
 UserOrderedCollection, Bag;
redefines:
// DuplEx is removed from all instance methods:
 op void insert(ELEM e) throws NullEx, BoundsEx;
 op void insertAtPos(ELEM e, Int pos)
                              throws NullEx, BoundsEx;
}
 // end of type List
```

1.3.5 The Concrete Type SortedList

A sorted list may contain duplicates and is automatically ordered. The operator ≤ is used to determine the order.

```
template <ELEM>
library type SortedList{
// Automatic ordering, allows duplicates
extends:
 SortedCollection, Bag;
redefines:
// DuplEx is removed in following instance methods:
 op void insert(ELEM e)
                  throws NullEx, BoundsEx;
 }
 // end of type SortedList
```

1.4 The Unordered Duplicate Free Types

The operator == determines what is regarded as a duplicate.

1.4.1 The Abstract Type DuplFreeCollection

```
template <ELEM>
library abstract type DuplFreeCollection {
/* DuplFreeCollections reject duplicates. The test for
   equality is defined by the operator == */
extends: Collection;
}
 // end of type DuplFreeCollection
```

1.4.2 The Concrete Type Set

Like the type `Bag`, the type `Set` is unordered. The difference is that a `Set` may not contain duplicates as determined operator ==. If it detects that a duplicate is being inserted it ignores the insertion without signally this.

```
template <ELEM>
library type Set {
// Set rejects duplicates without throwing an exception
extends: DuplFreeCollection;
// A DuplEx is not thrown in the following method:
redefines:
 op void insert(ELEM e)
                 throws NullEx, BoundsEx;
 }
 // end of type Set
```

1.4.3 The Concrete Type Table

The only difference between a `Table` and a `Set` is that in the case of a `Table` the `insert` routine causes an exception. (This can be important in data processing applications.)

```
template <ELEM>
library type Table {
// Table throws DuplEx when it receives a duplicate
extends: DuplFreeCollection;
// A DuplEx is thrown as approp. in the following method:
redefines:
 op void insert(ELEM e)
          throws NullEx, DuplEx, BoundsEx;
}
 // end of type Table
```

1.5 The Ordered Duplicate Free Types

In the following types use a the operator == to determine what is a duplicate and where appropriate the operator ≤ to determine the ordering automatically.

1.5.1 The Type Sorted Set

```
template <ELEM>
library type SortedSet {
 /* Automatic ordering (based on the operator ≤);
    rejects duplicates by ignoring them. */
extends:
 SortedCollection,
 Set;
redefines:
 enq ELEM get();
 /* returns the first element, or null in the case of an
    empty Sorted Table. */
 op void insert(ELEM e)
                  throws NullEx, BoundsEx;
 // DuplEx is removed from the insert method:
 } // end of type SortedSet
```

1.5.2 The Concrete Type SortedTable

```
template <ELEM>
```

```
library type SortedTable {
 /* Automatic ordering (based on the operator ≤);
    rejects duplicates by raising an exception */
extends:
 SortedCollection, Table;
redefines:
 enq ELEM get();
 /* returns the first element, or null in the case of an
    empty SortedTable */
 /* the following method is redefined to throw an exception
    if a duplicate element would result */
 op void insert(ELEM e)
          throws NullEx, DuplEx, BoundsEx;
}   // end of type SortedTable
```

1.5.3 The Concrete Type OrderedSet

```
template <ELEM>
library type OrderedSet {
// manual ordering, ignores duplicates
extends:
 UserOrderedCollection, Set;
redefines:
 enq ELEM get();
 /* returns the first element, or null in the case of an
    empty OrderedSet */
 // DuplEx is removed from the following instance methods:
 op void insert(ELEM e)
                  throws NullEx, BoundsEx;
 op void insertAtPos(ELEM e, Int pos)
                  throws NullEx, BoundsEx;
} // end of type OrderedSet
```

1.5.4 The Concrete Type Ordered Table

```
template <ELEM>
library type OrderedTable {
// manual ordering, rejects duplicates by raising exception
extends:
 UserOrderedCollection, Table;
redefines:
 enq ELEM get();
```

```
  /* returns the first element, or null in the case of an
     empty OrderedTable */
  // DuplEx is thrown in following instance methods:
  op void insert(ELEM e)
           throws NullEx, DuplEx, BoundsEx;
  op void insertAtPos(ELEM e, Int pos)
           throws NullEx, BoundsEx, DuplEx;
 } // end of type OrderedTable
```

2 The Collection Implementations

There are many ways of implementing the Timor collection types, e.g. with linked lists, with hash tables, etc. The following example implementation is based on arrays[54].

The approach adopted takes advantage of the Timor concept of re-use variables (see chapter 8). In this way it is possible to provide a complete implementation of the type `List`, and re-use this with minor changes and without a loss of efficiency to implement the remaining concrete types in the collection hierarchy. For the reader's convenience subsection 2.1 provides a consolidated version of the type `List`. Subsection 2.2 describes the array implementation of `List`, and its adaptation to implementations of the other types follows in subsection 2.3.

2.1 A Consolidated Definition of the Type List

It may be helpful to some readers at this stage to have before them a consolidated definition of `List` to compare this with the implementation in section 2.2.

```
template <ELEM>
library type List {
// Lists accept duplicates. They are user ordered.
instance:
 enq Int size();
  // returns the number of elements in the collection
 op void clear();
  // removes all elements
 enq Boolean contains(ELEM e) throws NullEx;
  /* returns true if e is in the collection. The equality
     test uses the operator == */
 enq Int occurrences (ELEM e) throws NullEx;
 // returns the number of elements == e.
 op void insert (ELEM e) throws NullEx, DuplEx, BoundsEx;
```

[54] Arrays in Timor are similar to Java arrays.

```
  // appends e at end of collection
  op void insertAtPos(ELEM e, Int pos)
           throws NullEx, DuplEx, BoundsEx;
  /* inserts e at the specified position. The elements which
     were previously at the specified position and higher
     numbered  positions automatically have the next higher
     positions */
  enq ELEM get();
  /* returns the first element in the collection, unless the
     element returned is null (in the case of an empty
     collection) */
  enq ELEM getAtPos(Int pos) throws BoundsEx;
  /*  returns the element at the indicated position. If
      the position is invalid, null is returned */
  op Boolean remove(ELEM e)() throws NullEx;
  /* removes one element == e; returns true if an element
     is found and removed, otherwise false. */
  op Boolean removeAtPos(Int pos) throws BoundsEx;
  /* removes the element at the indicated position */
  enq ELEM getNext throws EmptyCollection, IterationComplete;
  // returns the next element in the List.
  /* the implemntation of getNext is described in section 5.8,
     where it is explained how the implementation "knows" what
     is the next element to be returned. */
  } // end of type List
```

2.2 An Array Implementation of the Generic Type List

```
template impl List::ArrayImpl {
// manual ordering, allows duplicates
state:
 ELEM[] arr = null;
 Int maxlength, length = 0;
constr: // The constructor for this implementation.
 List::ArrayImpl (Int maxlength = 1000) throws BoundsEx
  /* an example of a parameter with default value, see
     Chapter 3 section 6 */
  {
  if (maxlength < 1 || maxlength > 1000000)
                             throw new BoundsEx();
  this.maxlength = maxlength;
```

```
   arr = ELEM[]::Impl(maxlength);
  }
 instance:
  enq  Int size() {
    // returns the number of elements in the collection
   return length;
  }
  op void clear() {
   // removes all elements
   length = 0;
   arr = null;
  }
  enq Boolean contains(ELEM e) throws NullEx {
   /* returns true if e is in the collection. The equality
      test uses the operator == */
   if (e == null) throw new NullEx();
   for (Int index in {0..length-1}){
    if (e == arr[index]) {return true;}
     else (return false;);
   }
  }
  enq Int occurrences (ELEM e) throws NullEx {
  // returns the number of elements == e.
   if (e == null) throw new NullEx();
   if (length == 0)  return 0;
   Int count = 0;
   for (Int index in {0..length-1}){
    if (e == arr[index]) {count++;}
   }
   return count;
  }
  op void insert(ELEM e) throws NullEx, BoundsEx {
   // appends e at end of collection
   if (e == null) throw new NullEx();
   if (length == maxlength) throw new BoundsEx();
   arr[length] = e; // insert at end
   length++;
  }
  op void insertAtPos(ELEM e; Int pos)
                                 throws NullEx, BoundsEx {
```

```
  /* inserts e at the specified position. The elements which
     were previously at the specified position and higher
     numbered  positions automatically have the next higher
     positions */
   if (e == null) throw new NullEx();
   if (length == maxlength || pos < 0 || pos > length)
                                            throw new BoundsEx();
   insertPos(pos, e);
   // call internal method; this adjusts length
  }
  enq  ELEM get(){
  /* returns the first element in the collection, unless the
     element returned is null (in the case of an empty
     collection) */
   if (length == 0) return null;
   return arr[0];
  }
  enq ELEM getAtPos(Int pos) throws BoundsEx {
  /*  returns the element at the indicated position. If
      the position is invalid, null is returned */
   if (pos < 0 || pos > length) return null;
   return arr[pos];
  }
  op Boolean remove(ELEM e) throws NullEx {
  /* removes one element == e; returns true if an element
     is found and removed, otherwise false. */
   if (e == null) throw new NullEx();
   if (length == 0) return false;
   for (Int index in {startPos..length-1}) {
    if (arr[index] == e) {deletePos(index); return true;}
   }
   return false;
  }
 op Boolean removeAtPos(Int pos) throws BoundsEx {
  /* removes the element at the indicated position */
   if (length == 0) return false;
   if (pos < 0 || pos >= length) throw new BoundsEx();
   deletePos(pos);
   return true;
  }
```

```
op void loopInit (){ /* This operation must be called
                        before each interation begins */
/*  the implementation of loopInit and of getNext is
    described in section 4.8, where it is explained how the
    implementation "knows" what is the next element to be
    returned. */
}
enq ELEM getNext() throws EmptyCollection, IterationComplete{
 /* returns the next element in the List. */
 /* the implementation of getNext is described in section
    4.8, where it is explained how the implementation "knows"
    what is the next element to be returned. */
 }
internal:
 /* the method deletePos(Int pos) deletes the element at pos
    by moving all elements from (pos+1) down one position in
    the array, so that the former pos+1 is now at pos. The
    length is adjusted in the routine */
 op void deletePos(Int pos) {
   length--;
   Int index = pos;
   while (index < length) {arr[index]=arr[index+1]; index++;}
 }
/* the method insertPos(Int pos, ELEM e) moves all elements
   (starting at the end of the array) from pos down one
   position in the array, so that the former pos is now at
   pos+1. The new element is inserted at pos and the length
   is adjusted in this routine */
 op void insertPos(Int pos; ELEM e) {
  Int index = length;
  while (index ≤ pos) {arr[index]=arr[index-1]; index--;}
  length++; arr[pos] = e;
 }
} // end of List::ArrayImpl
```

2.3 Implementing the Remaining Types

The following implementations apply the Timor re-use technique to implement the remaining types. In all cases this is based on the array implementation of `List` (see section 2.2).

2.3.1 Implementing the Type Bag

All the methods in `Bag` are already implemented in `List`. (Some methods of `List` (e.g. `getAtPos`) do not appear in `Bag`, because the latter has no defined ordering of its elements. Since this is the only fundamental difference between `List` and `Bag` (both accept duplicates) the `List` methods can be re-used without change. The fact that `List` has an ordering and `Bag` does not is not significant, because the order of elements in `Bag` is arbitrary (and can therefore be that of `List`).

Note: the following implementation is based on `List::ArrayImpl`, but any other implementation of `List` can be used to implement `Bag` following the same pattern (subject to different constructor parameters). Similar considerations apply to the implementation of other collection types.

```
template impl Bag::ArrayImpl {// unordered, allows duplicates
state:
 ^List::ArrayImpl myBag;
 /* a re-use implementation variable of List, which is
    constructed using the default constructor parameter */
with (myBag){
constr:
 Bag::ArrayImpl(Int maxlength) throws BoundsEx {
 // the constructor for this Bag implementation
   if (maxlength < 1 || maxlength > 1000000)
                              throw new BoundsEx();
   myBag = List::ArrayImpl(maxlength);
 }
 /* no additional instance methods are needed; all the Bag
    methods are matched by List methods */
}
} // end of Bag::ArrayImpl
```

2.3.2 Implementing the Type Set

The difference between `Set` and `Bag` is that elements of the latter may not contain duplicates as defined via the operator ==. Hence to re-use `Bag` all that is required is to override the latter's `insert` method. However, as the implementation of `Bag` simply re-uses that of `List` (without change), it is simpler to re-use the `List` implementation directly.

```
template impl Set::ArrayImpl {
// unordered, ignores duplicates without raising exceptions
state:
 ^List::ArrayImpl mySet; // a re-use variable of List
with (mySet){
constr:
 Set::ArrayImpl(Int maxlength) throws BoundsEx {
   if (maxlength < 1 || maxlength > 1000000)
                             throw new BoundsEx();
   mySet = List::ArrayImpl(maxlength);
 }
 instance:
  op void insert(ELEM e)
   /* overrides the insert method of List by first checking
      that an insertion is appropriate for a set, then uses
      the original insert method of List */
               throws NullEx, BoundsEx, FunctionParamEx {
   if (e == null) throw new NullEx();
   if (length >= maxlength || length < 0)
                                     throw new BoundsEx();
   if (length == 0) {mySet.insert(e); return}
    // check for duplicates and insert
   for (Int index in {0..length-1}){
    if arr[index] == e)) return;
    mySet.insert(e);
  }
 }
} // end of Set::ArrayImpl
```

2.3.3 Implementing the Type Table

The difference between `Table` and `Bag` is that elements of the latter may not contain duplicates in accordance the operator ==. Hence to re-use `Bag` all that is required is to override the latter's `insert` method. However, as the implementation of `Bag` simply re-uses that of `List` (without change), it is simpler to re-use the `List` implementation directly.

```
 template impl Table::ArrayImpl {
 // unordered, raises exception for duplicates
 state:
  ^List::ArrayImpl myTable;
 with (myTable) {
 constr:
  Table::ArrayImpl(Int maxlength) throws BoundsEx {
    if (maxlength < 1 || maxlength > 1000000)
```

```
                              throw new BoundsEx();
    myTable = List::ArrayImpl(maxlength);
  }
  instance:
   op void insert(ELEM e)
     throws NullEx, DuplEx, BoundsEx {
     if (e == null) throw new NullEx();
     if (length >= maxlength || length < 0)
                                      throw new BoundsEx();
     if (length == 0) {myTable.insert(e); return;}
     // check for duplicates and insert
     for (Int index in {0..length-1}){
       if arr[index]== e)) throw new DuplEx();}
     myTable.insert(e);
    }
  }
} // end of Table::ArrayImpl
```

2.3.4 Implementing the Type OrderedSet

The difference between `OrderedSet` and `List` is that elements of the latter may not contain duplicates. Hence to re-use `List` all that is required is to override the latter's `insert` methods.

```
template impl OrderedSet::ArrayImpl {
// ordered, ignores duplicates without raising an exception
state:
 ^List::ArrayImpl myOrderedSet;
with (myOrderedSet) {
constr:
 OrderedSet::ArrayImpl(Int maxlength) throws BoundsEx {
   if (maxlength < 1 || maxlength > 1000000)
                              throw new BoundsEx();
   myOrderedSet = List::ArrayImpl(maxlength);
  }
instance: // The following methods of List are overridden
 op void insert(ELEM e)
            throws NullEx, BoundsEx {
  if (e == null) throw new NullEx();
  if (length >= maxlength || length < 0)
                                     throw new BoundsEx();
  if (length == 0) {myOrderedSet.insert(e); return}
```

```
    // check for duplicates then insert
    for (Int index in {0..length-1}){
     if arr[index] == e return;
     myOrderedSet.insert(e);
   }
   op void insertAtPos(ELEM e, Int pos)
               throws NullEx, BoundsEx, FunctionParamEx {
    if (e == null) throw new NullEx();
    if (length >= maxlength || length < 0)
                                throw new BoundsEx();
    if (length == 0) {myOrderedSet.insert(e); return}
    // check for duplicates then insert
    for (Int index in {0..length-1}){
     if arr[index] == e)) return;
     myOrderedSet.insertAtPos(e, pos);
   }
  } // end of with statement
  } // end of OrderedSet::ArrayImpl
```

2.3.5 Implementing the Type OrderedTable

The difference between `OrderedTable` and `List` is that elements of the latter may not contain duplicates in accordance with the operator ==. Hence to re-use `List` all that is required is to override the latter's `insert` methods.

```
  template impl OrderedTable::ArrayImpl {
  // ordered, raises exception for duplicates
  state:
   ^List::ArrayImpl myOrderedTable;
  with (myOrderedTable) {
  constr:
   OrderedTable::ArrayImpl(Int maxlength) throws BoundsEx {
     if (maxlength < 1 || maxlength > 1000000)
                               throw new BoundsEx();
     myOrderedTable = List::ArrayImpl(maxlength);
    }
   instance:
    op void insert(ELEM e)
       throws NullEx, DuplEx, BoundsEx {
     if (e == null) throw new NullEx();
     if (length == maxlength) throw new BoundsEx();
     if (length == 0) {myOrderedTable.insert(e); return}
```

```
      // check for duplicates and insert
     for (Int index in {0..length-1}){
       if (arr[index] == e) throw new DuplEx();}
       myOrderedTable.insert(e);
    }
    op void insertAtPos(ELEM e, Int pos)
                            throws NullEx, DuplEx, BoundsEx {
      if (e == null) throw new NullEx();
      if (length == maxlength) throw new BoundsEx();
      if (length == 0) {myOrderedTable.insert(e); return}
      // check for duplicates then insert
      for (Int index in {0..length-1}){
       if arr[index] == e)) throw new DuplEx();}
      myOrderedTable.insertAtPos(e, pos);
    }
  }
  } // end of OrderedTable::ArrayImpl
```

2.3.6 Implementing the Type SortedList

The difference between `List` and `SortedList` is that elements of the latter are ordered in accordance with the operator >. Hence to re-use `List` all that is required is to override the latter's `insert` method.

```
  template impl SortedList::ArrayImpl {
  // sorted according to >, allows duplicates
  state:
   ^List::ArrayImpl mySortedList;
  with (mySortedList) {
  constr:
   SortedList::ArrayImpl(Int maxlength) throws BoundsEx {
    if (maxlength < 1 || maxlength > 1000000)
                              throw new BoundsEx();
    mySortedList = List::ArrayImpl(maxlength);
   }
  instance:
   op void insert(ELEM e)
              throws NullEx, BoundsEx {
    if (e == null) throw new NullEx();
    if (length == maxlength) throw new BoundsEx();
    if (length == 0) {mySortedList.insert(e); return}
      // now insert at appropriate position
    Int i = 0;
     while (arr[i] > e) i++;
     mySortedList.insertPos(i, e); /* Internal method of List;
       Internal methods of re-use variables can be invoked */
   }
  } // end of SortedList::ArrayImpl
```

2.3.7 Implementing the Type SortedSet

The difference between `SortedList` and `SortedSet` is that elements of the latter may not contain duplicates in accordance with the operator ==. Hence to re-use `SortedList` all that is required is to override the latter's `insert` method.

```
 template impl SortedSet::ArrayImpl {
 // ordered, ignores duplicates without raising an exception
 state:
  ^SortedList::ArrayImpl mySortedSet;
 with (mySortedSet) {
 constr:
  SortedSet::ArrayImpl(Int maxlength) throws BoundsEx {
    if (maxlength < 1 || maxlength > 1000000)
                              throw new BoundsEx();
   mySortedSet = SortedList::ArrayImpl(maxlength);
   }
  instance:
   op void insert(ELEM e)
             throws NullEx, BoundsEx{
     if (e == null) throw new NullEx();
     if (length == maxlength) throw new BoundsEx();
     if (length == 0) {mySortedSet.insert(e); return}
     // check for duplicates then insert
     for (Int index in {0..length-1}){
      if (arr[index] == e)) return;}
      mySortedSet.insert(e);
   }
  }
 } // end of SortedSet::ArrayImpl
```

2.3.8 Implementing the Type SortedTable

The difference between `SortedList` and `SortedTable` is that elements of the latter may not contain duplicates in accordance with the operator ==, and that an exception is thrown if an attempt is made to insert a duplicate. Hence to re-use

`SortedList` all that is required is to override the latter's `insert` method.

```
template impl SortedTable::ArrayImpl {
// ordered, ignores duplicates but raises an exception
state:
 ^SortedList::ArrayImpl mySortedTable;
with (mySortedTable.mySortedList) {
constr:
 SortedTable::ArrayImpl(Int maxlength) throws BoundsEx {
   if (maxlength < 1 || maxlength > 1000000)
                              throw new BoundsEx();
   mySortedTable = SortedList::ArrayImpl(maxlength);
  }
instance:
  op void insert(ELEM e)
     throws NullEx, DuplEx, BoundsEx, FunctionParamEx {
    if (e == null) throw new NullEx();
    if (length == maxlength) throw new BoundsEx();
    if (length == 0) {mySortedTable.insert(e); return}
    // check for duplicates then insert
    for (Int index in {0..length-1}){
     if (arr[index] == e)) throw new DuplEx();}
     mySortedTable.insert(e);
  }
}
} // end of SortedTable::ArrayImpl
```

3 Co-Types for the TCL Collection Hierarchy

Like other types the types in the TCL collection hierarchy can have corresponding co-types. Just as a type can have several implementations it can also have several co-types (and implementations thereof). In the first subsection a possible standard co-type `Collection&s` for `Collection` is presented. In contrast with the abstract type `Collection`, `Collection&s` is a concrete type. In the next subsection an implementation thereof is described. In view of the automatic existence of an adjustment hierarchy (see chapter 11 sections 7ff.) based on the co-type `Collection&s` it would be tedious and not particularly helpful to describe the co-types (and their implementations) for all the subtypes of `Collection`.

3.1 A Co-Type for the Base Type Collection

This co-type identifies by the keyword `TheType` those types in the definition which can be covariantly adjusted.

```
template <ELEM>
library type Collection&s expands Collection {
predefines instance:
 enq Int count(); // equivalent to a static method
 /* returns the number of created instances
    of concrete Collection subtypes, as recorded by
    the corresponding makers. See note below. */
predefines maker:
 /* The following makers provide a pattern for the
    concrete subtypes of Collection, which itself is an
    abstract type and therefore has no makers. */
 op TheType<ELEM> init();
 /* Creates an instance using the standard implementation
    and increments the count of instances.
    Successors for concrete expanded types create a new
    empty collection value of type TheType
    and increment the count of instances. */
 op TheType<ELEM> init(ImplName thisImpl) throws InvalidImpl;
 /* In this overloaded definion (see chapter 7 section 2)
    the parameter allows a user to select an implementation.
    Successors for concrete expanded types create a new
    empty collection value of type TheType
    and increment the count of instances. */
 op TheType<ELEM> convert
       (ImplName thisImpl; Collection<ELEM>*** c1)
                                  throws InvalidImpl, NullEx;
 /* Successors for concrete expanded types create a new
    collection value of type TheType and increment the count
    of instances. To do this they convert any collection c1
    to a collection value, using the specified
    implementation. */
 op TheType<ELEM> merge
        (ImplName thisImpl; TheType<ELEM>*** c1, c2)
                                   throws InvalidImpl, NullEx;
 /* Successors for concrete expanded types create a new
    collection value of type TheType<ELEM> by merging the
    content of two collections of the same type with
    elements of the same type and mode (see chapter 3
    section 6). The count of instances is incremented. */
 op TheType<ELEM> intersect
```

```
        (ImplName thisImpl; TheType<ELEM>*** c1, c2)
                                 throws InvalidImpl, NullEx;
  /* returns a new collection value by intersecting
     the content of two collections. The count is
     incremented */
  op TheType<ELEM> diff
        (ImplName thisImpl; TheType<ELEM>*** c1, c2)
                                 throws InvalidImpl, NullEx;
  /* returns a new collection value by taking the difference
     (c1-c2) between two collections. The count is
     incremented. */
 predefines binary:
  enq Boolean equal(TheType<ELEM>*** c1, c2) throws NullEx;
  // compares two Collection instances for equality
  /* returns false if
     - either or both collections have no elements
     - the collections have different elements
     - the number of occurrences of any element is not equal
  */
  enq Boolean includesAll(TheType<ELEM>*** c1, c2)
                                             throws NullEx;
  // checks if c1 includes but is not equal to c2
  /* returns false if
     - either or both collections have no elements
     - the number of elements in c2 exceeds or equals the
       number of element in c1
     - any element in c2 is not present in c1
  */
 } // end of Collection&s
```

Notes:

1. The co-type `Collection&s` is the base co-type for the `Collection` type hierarchy. As such it (partially) automatically defines an *adjustment hierarchy*, i.e. a parallel hierarchy of co-types for the subtypes in the `Collection` types (see Chapter 11 sections 7ff. and Figure 10.1). The special type identifier `TheType` designates where covariant adjustment takes place in the corresponding co-types. For example in the co-type `List&s`, the type identifier `TheType` means `List`.
2. The predefined makers provide a pattern for the co-types of the *concrete*

`Collection` types, since abstract types cannot be instantiated.

3. Most makers have an integer parameter which allows the user to select one of several implementations. How this can be implemented is shown in section 3.2.1.
4. The predefined **`binary`** sections apply to the co-types of both abstract and concrete types.
5. For each concrete type the corresponding co-type maintains a separate count of the instances which it creates. It would be possible to arrange for concrete types to inform the abstract types from which they are derived when new instances are created, and for these in turn to inform the abstract types from which they are derived, etc. In this way it would be possible for the application programmer to determine, for example, the number of instances of User Ordered or Duplicate Free instances, etc. and ultimately how many Collection instances have been created. However, this is more complicated to organise than one might think. But there is a much simpler way to achieve this, i.e. by providing an application module which simply sums the relevant concrete instances by using the `count` enquiry of the relevant co-types.
6. In each case the makers return a *value* variable of type `TheType`. It is then easy for the user to convert the result into a collection object or collection capability by using the **`new`** or **`create`** keyword. For example to create a module and its capability (with all access rights set to true) for a module of type `List<Person*>` he can simply write

```
List**<Person*> lpfile = create List&s<Person*>.init();
```

3.2 Implementing the Co-Type Adjustment Hierarchy

It is important to note that in the co-type implementations only the public and protected methods of the base types and of their co-types should normally be used, to ensure that they apply to all the implementations of the corresponding types. This implies that although it is theoretically possible to have implementations as re-use variables, a cleaner result can be achieved by relying only on re-use variables which are defined as types.

3.2.1 Implementing Collection&s

```
enum ImplName {default, array, linked, doubleLinked}
/* This names all the available implementations of the
   collection types. In our example the default is array.
   The enumeration type is used in makers to allow users to
   select an implementation suitable for their particular
```

```
     application. It is not intended that only those
     implementations listed should be made available in the
     final TCL. */
 template <ELEM>
 library impl Collection&s::Impl {
 state:
  Int count = 0;
  /* An explicit constructor is not required, because the
     state variables are initialised in the state section,
     see chapter 3 section 4. */
 /* Unlike the abstract type Collection, the type Collection&s
    is a concrete type, which needs an implementation. */
 predefines instance:
  enq Int count() {return count;}
  /* returns the number of created instances of TheType.
     For abstract types the count is always zero. */
 predefines maker:
  /* The following makers provide a pattern for the subtypes
     and can be implemented via re-use variables. Collection
     itself and the other abtract types have no makers.
  */
  op TheType<ELEM> init() {
  /* Creates an instance (concrete expanded types only) using
     the standard implementation. Concrete expanded types
     return a new empty collection value of type TheType. */
   count++; // increments the count of instances of TheType.
   return TheType<ELEM>::Impl();
      // a constructor call for the standard implementation
  }
  op TheType init(ImplName thisImpl) throws InvalidImpl {
   /* In this overloaded definion (see chapter 7 section 2)
     the parameter allows a user to select an implementation.
     Successors for concrete expanded types return a new
     empty collection value of type TheType. */
   TheType<ELEM> newCollection = selectImpl(thisImpl);
   /* selectImpl is an internal method which constructs the
     required implementation and returns this to the caller */
   count++;
   return newCollection;
  }
  op TheType<ELEM> convert
            (ImplName thisImpl; Collection<ELEM>*** c1)
                               throws InvalidImpl, NullEx {
  /* converts any collection c1 to a collection value
     of type TheType, using the specified implementation */
   if  (c1 == null) throw new NullEx();
   TheType<ELEM> newCollection = selectImpl(implName);
  for (ELEM x in c1)
   try { newCollection.insert(x); }
    catch (DuplEx de) { /* ignore it! */ };
   count++;
   return newCollection;
  }
  op TheType<ELEM> merge
            (ImplName thisImpl; TheType<ELEM>*** c1, c2)
                                throws InvalidImpl, NullEx {
  /* returns a new collection value by merging the
     content of two collections. */
   if  (c1 == null || c2 == null) throw new NullEx();
  // create new collection
   TheType<ELEM> newCollection = selectImpl(implName);
  // the merge algorithm
   for (ELEM x in c1)
    try {newCollection.insert(y);}
     catch (DuplEx de) { /* ignore it! */}
   for (ELEM x in c2)
    try {newCollection.insert(x);}
     catch (DuplEx de) { /* ignore it! */
   count++;
   return newCollection;
  }
  /* ... see note 3 below for the implementations of the
         intersection and difference methods */
 predefines binary:
  enq Boolean equal(TheType<ELEM>*** c1, c2) throws NullEx {
  // compares two Collection instances for equality
   if (c1 == null || c2 == null) throw new NullEx();
   if (c1.size = 0 || c2.size = 0) return false;
   if (c1.size() != c2.size()) return false;
   for (ELEM x in c1) {
    if (c1.occurrences(x) != c2.occurrences(x)) return false;
    }
   return true;
  }
  enq Boolean includesAll(TheType<ELEM>*** c1, c2)
                                            throws NullEx{
  // checks if c1 includes c2 but is not equal to c2
   if (c1 == null || c2 == null) throw new NullEx();
   if (c1.size = 0 || c2.size = 0) return false;
   if (c2.size >= c1.size) return false;
   for (ELEM x in c2)
    if (!(x in c1)) return false;
   return true;
  }
 internal:
  op TheType<ELEM> selectImpl(ImplName thisImpl)
                                       throws InvalidImpl {
  case (implName) of {
    (default) {return TheType<ELEM>::Impl();}
    (array) {return TheType<ELEM>::ArrayImpl(*);}
     /* The asterisk signifies that the default parameter
        value should be used if the programmer does not supply
        one, see Chapter 3 section 7. */
    (linked) {return TheType<ELEM>::LinkedImpl();}
    (doubleLinked) {return new TheType<ELEM>::DoubleLinked();}
    else {throw InvalidImpl();}
   }
  }
 } // end of Collection&s
```

Notes:

1. Makers always return a collection in value mode. Application programmers can easily use the **new** or **create** operator on the result to produce the required mode.
2. In some of the method input parameters for collections are passed using handles, using the *** notation. These parameters are used in **for** loops to cycle through the elements in the collections. This as such is not problematic, since a **for** loop can cycle through the elements of any collection.
3. The more interesting question concerns the modes of the elements in a col-

lection. These are not immediately obvious from a type definition using the generic identifier `ELEM`. However, when actualisation occurs at runtime the actual modes are known and if a single parameter is used (as in the case of the maker `convert`) the situation is straightforward. Where multiple collection parameters are input (as in `merge`) the collections to be merged must be of the same type and their elements must also have the same mode. This is guaranteed in that the parameter description `TheType<ELEM>*** c1, c2` defines them both as having the same `ELEM` type (which implies that their elements have the same mode). In a more complicated merge method which for example also changes the mode of elements in the second parameter to those of the first parameter, the two parameters would have to be declared separately and an explicit mode coercion cast (e.g. `(reference)`, see chapter 4 section 4) would be necessary.

4. In this implementation of `Collection&s` the binary methods are real methods which can accept collections of any of the `Collection` subtypes. In the binary methods no problems arise as a result of their differences. Both methods rely simply on the "statistical" methods and then run through the two collections (regardless of their types). This is also true for the other abstract types.
5. The makers `init()` and `init(ImplName thisImpl)` are also straightforward. The makers `convert` and `merge` contain code which simply ignores `DuplEx` exceptions in the **`for`** loops which insert elements into a new collection, as a result of attempting to create a `Table`, an `OrderedTable` or a `SortedTable`. With respect to the ordering of elements in collections which these methods create, this is organised by the `insert` methods.
6. The makers `intersect` and `diff` have somewhat more complicated algorithms than `merge`, but otherwise follow the same pattern. Since the aim is to describe the features of Timor and not to present and discuss particular algorithms, the descriptions of `intersect` and `diff` have been omitted.
7. In the internal method `selectImpl` some individual constructor calls take advantage of the default parameter facility (see chapter 3 section 6). If `TheType` were actualised to `List` and implementation name `array` were used, the actual array would be initialised according to the parameter default, in this case to hold up to 1000 elements (see chapter 13 section 2.2.).

3.2.2 Implementing the Remaining Co-Types

The considerations discussed in Notes 4 and 5 of section 3.2.1 apply not only to `Collection&s` but also to implementations for all the remaining co-types in the adjustment hierarchy. Hence they are very easy to implement, as follows

```
template <ELEM>
library impl Ordered&s::Impl /* or the co-type for any other
abstract or concrete collection type */
{state:
 ^^Collection&s::Impl;
}
```

Of course real makers appear only in the implementations for the concrete co-types.

4 Collection Syntax

Although the methods of the collection types can always be called explicitly, Timor provides additional syntactic possibilities for accessing collections[55]. In this subsection the following definitions are assumed:

```
List<Person>  pList;
List<Student> studentList;
```

Here are some of the features.

4.1 Collection Literals:

The expression `{1..10, 12, 15..20}` is collection expression which is an example of a literal of type `List<Int>`. Collection literals are *always* of type `List`, because the type `List` is compatible with all other collection types in the sense that lists are ordered and can contain duplicates, and hence can be converted to other collection types using the appropriate maker.

`List` literals are bounded by curly brackets and either

a) contain integers and/or ranges of integers, separated by commas (as in the above example), or

b) list elements of any appropriate type, separated by commas, optionally preceded by a type name and a colon, e.g.

   `{Person: p1, p2}`

   or

   `{"MyString", s, "HerString"}`, etc.

   The type name is optional if the type is that of the first element in the list.

4.2 Subcollection Selection

A subcollection expression returns a selection of the elements in a collection,

[55] This syntax is based largely on the syntax as defined for her language Collja by Dr. Gisela Menger in her Ph.D. thesis [2].

based either on a predicate or by position.

A. *Selection by Predicate*:

A variable (here `Person p`) can be declared which is used in the predicate (a Boolean expression) as the criterion for selecting the elements which appear in the subcollection, e.g.

```
myPersonSet{Person p:
  p.dateOfBirth.toString() < "1/1/1950" && p.spouse != null}
```

The subcollection which is returned is of the same type as the collection on which it is based. It can then be assigned, for example, to a collection variable, e.g.

```
Set<Person> marriedElderlyPersons = myPersonSet{Person p:
  p.dateOfBirth.toString() < "1/1/1950" && p.spouse != null}
```

B. *Selection by Position*:

For *ordered* collections a subcollection can be created as a subrange of the elements of the collection on which it is based. In this case a variable is not required. Here is an example:

```
Set<Person> reducedPersonSet = myPersonSet{i .. j};
```

where `i` and `j` are integer variables or literals. The first element in a collection is considered to have position 0.

4.3 Element Selection

An element expression selects a single element from a collection. This element is selected in a similar manner to that of subcollections, by predicate or by position.

A. *Selection by Predicate*:

Whereas the predicate for subcollections is enclosed in curly brackets, square brackets are used for selecting a single element, e.g.

```
Person marriedElderlyPerson = myPersonSet[Person p:
      p.dateOfBirth > "1/1/50" && p.spouse != null];
```

The *first* element matching the criterion is selected for ordered collections, or an arbitrary matching element for unordered collections.

B. *Selection by Position*:

Selection by position (for ordered collections) also uses square brackets and in this case a single position is nominated, e.g.

```
Person aPerson = myPersonSet[3];
```

In effect this allows ordered collections to appear as arrays in other languages.

4.4 Type Conversion

A special conversion operator is not necessary since the programmer can always achieve this by using a cast. For example a list of students `studentList` can be accessed as a `Set<Person>` by writing `(Set<Person>) studentList.`

4.5 Collection Operators

The union, difference and intersection expressions for collections are as follows.

Union/Merge:	`list1 + list2,`	`list1 + {p}`
Difference:	`list1 - list2,`	`list1 - {p}`
Intersection:	`list1 * list2,`	`list1 * {p}`

Each of the concrete collection types has a standard co-type which includes three makers: `merge`, which forms the union of (i.e. merges) two collections into a new collection), `difference` (which creates a new collection containing the difference between two collections) and `intersect` (which creates a new collection containing the intersection of two collections). The input parameters are in all cases of type `Collection`, i.e. any collections can be input, regardless of the place in the collection hierarchy. The returned instance is a collection of the type expanded in the individual co-type. The results of these operations take into account the duplication and ordering properties of the resultant type.

The reference copy operator `=*` is used for copying collections of objects (see Chapter 4 section 7). It creates a collection which contains only references to objects from another object collection.

The abstract collection types cannot be instantiated, and therefore cannot have makers. Consequently the rule used for the comparison operators, that the nearest common ancestor determines the methods to use, cannot straightforwardly be applied to the use of operators which create new collections (conventionally + (merge), - (difference), * (intersect)). The following rule is therefore applied. These three operations are only applicable to cases in which the two operands have the *same static type*, i.e. mixed collection types cannot be operated on using these operators[56]. However, this does not limit the programmer from carrying out operations on mixed types by directly using the appropriate co-type methods.

4.6 Boolean Expressions

The standard comparison operators can be used to compare two collections which have elements of the same type or where the elements are in a subtype

---

[56] This does not exclude operations on collections which include subtype instances of the static type.

relationship to each other.

Equality:	`list1 == list2,`	`list1 != list2`
Identity:	`list1 ~~ list2,`	`list1 !~ list2`
IncludesAll/IncludesEqual:		
	`list1 > list2,`	`list1 >= list2`
(alternative):	`list1 ⊃ list2,`	`list1 ⊇ list2`
Contains:	`elem in list1`	
Included/IncludedEqual:	`list1 < list2,`	`list1 <= list2`
(alternative):	`list1 ⊂ list2,`	`list1 ⊆ list2`

The compiler uses the 'equal' and 'includesAll' collection co-type methods in the obvious way, returning the normal subset/subbag relationships.

The operator **`in`** checks whether an element (first operand) is present in a collection (the second operand) and returns a boolean result, e.g.

```
if p in StudentCollection
```

The first operand must be a single instance and the second a collection with elements of the same type or of a subtype thereof.

4.7 Iteration

As indicated in chapter 2 section 1 Timor supports **`while`**, **`repeat`** and **`for`** statements. The **`for`** statement needs some further explanation here, since it describes how the programmer can iterate through the elements in a collection. Here is an example:

```
Int separatedPersons = 0;
for (Person p in myPersonSet) {
 if (p.spouse != null && p.spouse.address != p.address)
  {separatedPersonSet.insert(p); separatedPersons++;}
}
[else {...}] // do something if myPersonSet has no elements
```

Notes:

1) `myPersonSet` is a collection expression, which might for example be a subcollection.
2) `Person p` is a variable of the element type of the collection expression, which is used to address each element in the collection in turn.
3) The `else` clause is optional.

Here is an example of a simple **`for`** statement:

```
for (Int i in {1..10, 12, 15..20}) {...};
```

4.8 Implementing the `for` Statement

Some aspects of implementing the **`for`** statement need further explanation.

4.8.1 Iterating through Ordered and Unordered Collections

The order of selecting elements while iterating through an *ordered* collection is straightforward, the elements are selected in the order defined for the collection in question. The order of selecting the elements in an *unordered* collection, i.e. in a `Set`, a `Bag` or a `Table`, is nondeterministic and in practice may depend on the implementation of the collection in question. Hence for unordered collections the user cannot rely on the same order being used each time.

4.8.2 How Does the Method `getNext` know which Element to return next?

This is by no means a straightforward issue, especially in an environment such as ModelOS, where multiple threads may be concurrently accessing (and iterating through) the same collection in parallel. Furthermore it is possible that these threads might be using different implementations for the same type (but of course not for the same instance, since an actual instance has only a single implementation).

Timor follows the same strategy as ModelOS, which provides a mechanism called *retained* data[57]. The *retained* data section of an implementation enables separate data segments to be defined for each thread which opens an instance of a module. Data in this section can be initialised, accessed, modified and deleted as appropriate by the thread which opened the module whenever the latter is active in the module, until it closes the module (at which point the retained data for the calling thread is deleted). Different instances of the retained data can be active in parallel in different threads, provided that they have opened the module. No thread can address or access the retained data of a different thread.

In terms of the specific issue of iterating through a loop, a thread which has opened the module can make a note in its retained data of how far through the loop it has reached as it executes the `getNext` method. Since each thread has its own retained data which cannot be accessed by other threads, different threads cannot interfere with each other's progress through the loop. Note that the synchronisation protocol guarantees this, since when a module is opened by a thread it must indicate whether it wishes to access the module as a reader or a writer. An active writer thread can open and modify the module, but the synchronisation mechanism automatically excludes other threads (writers and readers) from accessing the module's data while the writer is active. If multiple readers are active they can change neither the elements nor their ordering, and since

[57] see chapter 3 section 4 and for the ModelOS aspects see chapter 18 section 4 of (1).

each has a separate retained data segment it cannot interfere with the others' progress markers.

The implementation of progress markers is not system defined but depends instead on the nature of an actual implementation. For example if the implementation uses an underlying array structure, as in the example implementation in section 2.2 of this chapter, then the marker might be an index value, whereas if for example a linked list is used, the marker might be a pointer into the collection structure, etc.

4.8.3 Adding a Progress Marker to the Example Implementation

The example implementation in section 2.2 can now be completed. This involves first adding a retained section:

```
retained:
 Int nextElem = 0; /* In the array implementation the marker
    for the next element is an index into the array */
```

Then appropriate looping methods must be provided:

```
op void loopInit (){ /* This operation must be called
                        before each interation begins */
 nextElem = 0;
}
enq ELEM getNext() throws EmptyCollection, LoopComplete {
 if arr == null throw new EmptyCollection();
 if nextElem == length throw new LoopComplete();
 nextElem++;
 return arr[nextElem - 1];
}
```

A thread can invoke the `loopInit` at any point in its execution. If an iteration is partially complete, the effect is that the next invocation of `getNext` will return the first element.

4.8.4 Openable Collections

The solution as presented so far is incomplete, since it relies on the collection in question being openable. It will be recalled that two special methods (`open` and `close`) were introduced in chapter 3 section 6, which are categorised neither as **enq** nor as **op**. Then in chapter 11 section 5 it was shown how these methods can easily be added to any other Timor type and thus produce an `Openable` version of the type. Does this imply that any collection module through which a program can iterate should be openable? We now seek an answer to this question.

4.8.4.1 Using Collections as File Modules in ModelOS

In the ModelOS environment collections will frequently be used as persistent free-standing file modules (data bases) which hold elements that can be accessed by other ModelOS modules via the routines described in section 1 above.

Since in multi-user or multi-threading environments such files will frequently be accessed in parallel by different threads there is a need to synchronise such access. At the Timor level this can be organised by defining them as openable ModelOS files as described in chapter 18 of [6]. At the Timor level this means that the chosen collection type should be composed together with the *view* component `Openable` (see chapter 11 section 5), for example to define a new type `OpenableList`, e.g.

```
type OpenableList<ELEM> {
  extends Openable, List<ELEM>;
}
```

A definition of `Openable` (cf. chapter 10 section 5) might be as follows:

```
enum OpenMode {closed, read, write}
view Openable {
  op void open(OpenMode mode) throws OpenError;
  op void close();
  enq OpenMode openMode();
}
```

The qualifier `OpenSynchroniser` (see chapter 10 section 5, which also provides an implementation) can then be used to synchronise the `OpenableList` methods.

4.8.4.2 Using Collections as Small Objects in a Timor Program

Most programming language designers usually think of collections as temporary objects in non-persistent systems, and might therefore find the idea of opening and closing collections somewhat strange. But they should also recall that conventional programming languages need to fall back on unusual mechanisms to allow programmers to iterate though collections, e.g. iterators in languages such as Java and C++. The main advantages of the ModelOS/Timor mechanism are:

a) They use a mechanism which is not special to the issue of iterating through collections. Retained data and the opening/closing of objects can be useful in many situations.

b) They provide synchronised access, which is important in systems that allow multiple threads to access a collection in parallel.

c) Although the application programmer must use the open/close routines, he does not need to be concerned with the complexities of retained data and the `getNext` routine, since this is hidden from him by the `for` statement.

d) In a single-threading (or an appropriately synchronised) environment which only uses the **`for`** statement to iterate through simple sets of literal expressions such as

```
for (Int i in {1..10, 12, 15..20}) {...};
```

the compiler can handle this without using open/close routines.

4.8.5 Implementing Retained Data in Conventional (non-ModelOS) Environments

In ModelOS environments retained data segments are provided at the operating system level. But such support is not available in conventional environments. To implement the equivalent in such systems is relatively easy, provided that the run-time system can identify threads/processes. In this case a data structure is needed which consists of a list of records consisting of a thread/process number and the retained data information for that thread/process. Such a record can be created by the open routine and deleted by the close routine called by the respective threads/processes, and of course updated by routines such as `getNext`, etc.

Chapter 14 Generic Function Parameters

Timor's second generic feature allows generic expressions to be provided as parameters to constructors[58]. In the form supported by Timor this is a variant of an idea first proposed by my former research student and assistant Dr. Mark Evered, which he called 'expression parameters' and which he incorporated into a Java-like language Genja [28, 29]; this idea was also included in the Java-like Collja language [2] developed by my former research student and assistant Dr. Gisela Menger.

However, two changes have been made to the original proposal (apart from adapting it to the different structure of Timor).

- In Evered's original work expression parameters (which are only associated with constructors) are a factor in determining the type of objects, e.g. a `Person` set with one criterion for rejecting duplicates differed in type from a `Person` set which used different criteria for rejecting duplicates; this is not the case in Timor.
- Second, his syntax for expression parameters includes a default option, which is not present in Timor.

Generic function parameters should not be confused with function parameters in other languages. They can be statically associated with *constructors* of implementations, in which case the actualisation of the generic function does not change over the lifetime of an instance); these are known in Timor as *static function parameters*. They appear in constructor definitions and can be actualised in different ways when a constructor is invoked.

[58] The primary contexts for which function parameters were designed are the Timor Collection Library (see chapter 14) and similar Timor container types, such as Stack and Queue types. Hence the examples in this chapter are based on the TCL types.

An extension of Evered's idea, which we called "dynamic function parameters", was also considered. The basic thought behind this was to allow function parameters to be defined in individual instance methods of a collection type in a similar manner to the static function parameters described below. But despite some attractive aspects of this (e.g. it in effect gave applications a mechanism which allowed them to formulate requests in terms of the "columns" of a collection and thus enabled an approach resembling relational databases). However, the idea was abandoned because it would have involved the use of call back methods; this would have resulted in less efficient and more restricted than programs than can be written simply using the normal instance methods of collections. It would also have unnecessarily complicated the ModelOS design. Nevertheless a similar effect with respect to the formulation of requests in terms of the "columns" of a collection can be realised in Timor (in a simpler manner) via some features of the collection syntax described in chapter 13, section 4 (e.g. subcollection selection and element selection by predicate).

1 Function Clauses

Static function parameters are defined in *function clauses*, which contain a list of *function headers* in a bracket pair < >, using syntax resembling that of Timor method headers. A function clause appears immediately before the normal parameter list of the constructor or method. If a constructor or method has more than one function parameter, these are separated by semicolons, e.g.

```
<Boolean EQUAL(ELEM e1,e2); Boolean PRECEDES(ELEM e1,e2)>
```

Each function header consists of a type name (defining the type which it returns), followed by its generic identifier and then by a list of "normal" method parameter declarations, which may use generic type identifiers defined in the type template in which the parameter is embedded. Typically the generic method parameters (of type `ELEM` in our examples) refer to elements in the collection.

In contrast with normal Timor method headers, function headers are not preceded by the keyword **op** or **enq**, because they are pure functions which may not modify (nor access in any way) the state data of an actual implementation.

A function parameter is actualised when the constructor is invoked; the actualisation takes the form of source code that uses expression syntax (in a round bracket pair **()**). The value returned must be of the type specified in the function parameter.

2 Motivation for Static Function Parameters

Static function parameters (like Evered's expression parameters) are needed when the same parameter actualisation can be used over the lifetime of the instance of the type, in one or more instance methods. Here are some examples.

Collection types derived from the abstract type `DuplFree` (see Figure 13.1) do not permit duplicate elements. An issue which then arises is what criterion determines when a potential element should be considered a duplicate. For example it might be reasonable in one case to define a `Set<Person*>` where the name and date of birth of individual elements must differ, but in another case it might be more appropriate to ignore `Person*` elements with the same passport number.

Similarly, concrete collection types derived from the abstract type `Sorted` (see Figure 13.1) automatically order the elements in a collection and therefore need a mechanism for determining which element should precede another.

Criteria of this kind only make sense if applied consistently over all the invocations of the relevant methods of a specific instance of a type and are therefore associated statically with an instance when it is constructed.

The function clause illustrated in the previous section, i.e.

```
<Boolean EQUAL(ELEM e1,e2); Boolean PRECEDES(ELEM e1,e2)>
```

is in fact a static function clause used for those types in the TCL which are derived by multiple inheritance both from `DuplFree` and from `Sorted` (i.e. `SortedSet` and `SortedTable`).

The first function is the EQUAL function which has two input parameters of the generic type ELEM and returns a boolean result 'true' if the comparison undertaken determines that these two elements `e1` and `e2` are 'equal' (according to a definition to be supplied later in an actualisation, when the constructor is called).

The second function is the PRECEDES function, which also has two input parameters of the generic type ELEM and returns a boolean result 'true' if the comparison undertaken determines that the element `e1` 'precedes' the element `e2`.

3 Defining Static Function Parameters in Type Templates

Although they do not play a role in determining the type of instances, static function clauses are included in type templates immediately following the template header, preceded by the keyword `func`. This allows the designer of the template to indicate that particular static functions are needed and should be used in implementation templates. From the implementer's viewpoint it defines what static function parameters should be made available in constructors for every implementation of the template. Users of the type can identify what parameters they must provide in every constructor call to create an instance of the type, regardless of the implementation chosen. Here is an example:

```
template <ELEM>
func <Boolean EQUAL(ELEM e1,e2);
      Boolean PRECEDES(ELEM e1,e2)>
   /* static function parameters are used in implementations
      of the collection type's instance methods */
type SortedSet {
extends: DuplFree<ELEM>; Sorted<ELEM>;
...
}
```

4 Using Static Function Parameters in Implementations

A static function parameter defined for a type can appear in an implementation of any method in the type, and is not repeated in the definitions of individual methods.

Here is a partial implementation of `SortedSet`, showing how the function parameters might be used in the `insert` method. Note that the static functions are implemented in the implementations of the collection type, but when these are later invoked in invocations of the methods in which they are embedded, they must be independent of a particular collection implementation.

```
template impl SortedSet<ELEM>::ArrayImpl {
state:
 ELEM[] arr = null;
 Int maxlength, length = 0;
constr: // The constructor for this implementation
 SortedSet::ArrayImpl /* the constructor id is followed
     by the static parameters from the type template,
     as follows */
  <Boolean EQUAL(ELEM e1,e2); Boolean PRECEDES(ELEM e1, e2)>
 // and is then followed by the normal constructor parameters
  (Int maxlength)
 { // and now the constructor code
  this.maxlength = maxlength;
  arr = ELEM[]::Impl(maxlength);// a Timor array constructor
 }
instance: // the instance methods of the type
 ...
op void insert(ELEM e) throws NullEx {
 if (e == null) throw new NullEx();
```

```
   Int i = 0;
   while (i < length && PRECEDES(arr[i], e)) i++;
    /* This loop iterates to find the appropriate position
       for the insertion, as defined in the actualisation of
       the static parameter PRECEDES */
   if (i == (length-1) && !(EQUAL(arr[i], e)))
                                   throw new OutOfBounds();
    // an insertion requested beyond end of the array
   if (i < length) {
    if (EQUAL(arr[i], e)) return;
     /* ignore this insertion request because the type is a
        subtype of Set */
    else {for (Int k in {k..(i+1)}) arr[k] = arr[k-1];}
    arr[i] = e; length++;
   }
  }
  }
```

The comments in the above code are intended to be helpful, but may actually be confusing for some. We therefore repeat the code without the comments:

```
  template impl SortedSet::ArrayImpl {
  state:
   ELEM[] arr = null;
   Int maxlength, length = 0;
  constr:
   SortedSet::ArrayImpl <Boolean EQUAL(ELEM e1,e2),
          Boolean PRECEDES(ELEM e1, e2> (Int maxlength)
   { this.maxlength = maxlength;
     arr = ELEM[]::Impl(maxlength); }
  instance:
   ...
  op void insert(ELEM e) throws NullEx {
   if (e == null) throw new NullEx();
   Int i = 0;
   while (i < length && PRECEDES(arr[i], e)) i++;
   if (i == (length-1) && !(EQUAL(arr[i], e))
                                   throw new OutOfBounds();
   if (i < length) {
     if (EQUAL(arr[i], e)) return;
     else {for (Int k in {k..(i+1)}) arr[k] = arr[k-1];}
```

```
    arr[i] = e; length++;
   }
  }
  }
```

5 Actualising Static Function Parameters

Static function parameters are actualised when an instance of a type is constructed. For example a constructor for `SortedSet` could be invoked as follows. This returns an instance of the type `SortedSet`.

```
SortedSet<Person>::ArrayImpl // a constructor
 <(e1.name == e2.name && e1.address == e2.address),
  // an expression actualisation of EQUAL
  (e1.passportNum < e2.passportNum)>
  // an expression actualisation of PRECEDES
(256); // the normal constructor parameter
```

Associating the static function parameters with constructor invocations allows each instance of the same type to have different criteria, e.g. for determining duplicates. At the same time actualising the function parameters with each instantiation of a type ensures that for a particular instance of the type the same static parameter actualisation is consistently applied whenever methods which use the actualisation (e.g. `insert`) are invoked on that instance.

When a constructor which has static function parameters is invoked, the compiler configures the code for that instance in accordance with the actualisation(s) provided. The language does not define how this happens, but several implementations are possible.

5.1 A Suggested Implementation

The advantage of static function parameters is that they do not change throughout the lifetime of the instance being created by the constructor.

Especially in the case of a collection being used as a persistent file module the lifetime of the module could be very long (e.g. a payroll file in a business); this suggests that run-time efficiency should be a primary consideration. In such cases it would be sensible to re-compile the module to match the requirements laid down in the specific actualisation of the static function parameters. This would primarily involve replacing each invocation of the function parameter(s) with the actualisations. This requires the identification of the parameters in the actualisation and substituting these for the appropriate values in the constructor's implementation code. In this case `e1` = `arr[i]` and `e2` = `e` in the EQUAL actualisation which are correspondingly modified by `.name` and `.address`, and *mutatis mutandis* for the PRECEDES actualisation. For example the `insert` routine in the last example (section 3.3) would be changed as follows using the actualisations described in section 3.4:

```
op void insert(ELEM e) throws NullEx {
 if (e == null) throw new NullEx();
 Int i = 0;
 while (i < length && (arr[i].passportNum < e.passportNum))
        i++;
 if (i == (length-1) && !(arr[i].name == e.name &&
                          arr[i].address == e.address))
                                throw new OutOfBounds();
 if (i < length) {
   if (arr[i].name == e.name &&
       arr[i].address == e.address) return;
   else {for (Int k in {k..(i+1)}) arr[k] = arr[k-1];}
   arr[i] = e; length++;
 }
}
```

This proposal has the further advantage that the compiler automatically checks the actualisation text for errors.

Chapter 15
Support for ModelOS

Timor was designed as a general purpose component oriented programming language, but with support for ModelOS as one of its main aims. How some special aspects relating to ModelOS are handled in Timor is explained in this chapter.

1 Returning Values of User-Defined Types

ModelOS has a rule that references cannot be passed or returned as parameters for *file modules*[59], to avoid a potential world-wide garbage collection problem, see Chapter 20 section 6 of [6]. The only parameters and return values permitted are values and capabilities. On the other hand Timor, like many OO languages, permits only a single return value for methods. This raises the question of how a value of a user-defined type which consists of multiple fields can be passed or returned.

The answer is that the Timor compiler decomposes, passes and returns the constituent fields of such a type as individual ModelOS parameters on inter-module calls and returns. However, if any of these fields are defined as references (rather than capabilities and values) or as values of nested user defined types (whether or not these contain references) a compile time error occurs. Note that this mechanism applies only to inter-module calls (i.e. calls to and returns from file modules activated via capabilities. Normal calls between objects held within the same module are managed as normal.

There is one exception to this rule. ModelOS modules which are held in the same container (including co-modules such as the segment manager, library modules and some closely related application modules, which are discussed in chapter 18 section 7 of [6], can pass reference (i.e. ModelOS pointer) parameters. Timor indicates which modules may receive such parameters in that the type name is preceded by the designation `comod` or `library`, as appropriate.

[59] The parameters of calls to *internal objects* within a module can include references.

2 Handling ModelOS Access Rights

Access rights (which are based on ModelOS semantic access rights) can be defined in a *restrictor* which directly follows the type name in a variable or parameter declaration. This consists of a list of *allowable* methods in the bracket pair `[: method list :]`. Methods can be individually named in such a list[60], e.g.

```
TextFile[:insert, remove:]* tf;
```

However, it is often more convenient to use *view* names. If a programmer wants to pass a parameter to a method which is only allowed to use the `Openable` methods of a text file object, the parameter could be declared as:

```
TextFile[: Openable :]* tf;
```

To ensure that programmers cannot avoid this restriction by assigning the associated instance to another `TextFile*` variable, the compiler (or its run-time system) ensures that assignments are only possible to variables with the same or more stringent restrictions. (In the case of modules, ModelOS controls the use of access rights in capabilities.)

Predefined Views

There are some predefined views which are especially useful in defining restrictions. These include (but are not limited to) the following:

`all`	=	all the methods of the type being restricted.
`allonly`	=	all the methods of the type are available, but also that the **`cast`** statement cannot be used to gain downcast access to the methods of a subtype object.
`op`	=	all the *operations* associated with the type.
`enq`	=	all the *enquiries* associated with the type.
`body`	=	for a qualifying type, the `body` operation can be called (in a call-in bracket method only).
`call`	=	for a qualifying type, the `call` operation can be called (in a call-out bracket method only).
`overwrite`	=	the value of the object currently associated with the variable may be overwritten with a new value (i.e. for a variable defined as `T[:all-overwrite:]* t`, the dereferencing operator `*t` may not be used on the left side of an assignment statement).
`copy`	=	the value of the object currently associated with the variable may be copied (i.e. for a variable defined as `T[:all-`

[60] If an overloaded method is named then all the methods with this name are allowed.

copy:]* t, the dereferencing operator *t may not be used in an expression).

Note: If the overwrite and/or copy views do not explicitly appear in a restriction, but op and/or enq brackets are explicitly restricted, then the overwrite and/or copy operations are considered to be restricted. However with the restriction T[:overwrite-op:]* the overwrite operation can be carried out, although op methods may not be called. Similar considerations apple with T[:copy-enq:]*.

Multiple views and method names can be combined, using the set union (+), set difference (-) and set intersection (*) operators. For example

```
TextFile[:all-Openable:]* tf2;
```

indicates that all the TextFile methods except the Openable methods can be used.

3 Calls to the ModelOS Kernel

The ModelOS kernel is not considered to be a module, but at the ModelOS level it provides a (partly protected) user interface which is considered as an enhancement to the instruction set. This interface is accessible to Timor modules as a built-in quasi-module (called **kernel**) such that each kernel instruction can be regarded as an interface method (with parameters where appropriate). If the instruction is a privileged instruction then a kernel capability (with the corresponding access right set) must be passed as a parameter.

This allows the Timor programmer to "tune" his module to take advantage of special facilities (mainly security facilities), which are not normally available in other programming languages. For example if the security of a Timor program can be enhanced by information about the security environment in which his module is intended to execute, he can access the kernel instructions which provide environmental information (see [6] volume 2 chapter 26 section 1).

3.1 Executing Simple Kernel Instructions

To discover the unique identifier of the user which is currently executing his module he can write

```
LongInt myCaller = kernel.current_thread_owner();
```

or to establish which user owns the file in which his code is currently executing he can include the statement

```
LongInt fileowner = kernel.current_file_owner();
```

3.2 Executing Kernel Instructions involving Access and Control Rights

The programmer might want to restrict the thread in which his module is executing so that it cannot be transferred as a remote inter-module call to some other node. He can do this by unsetting the `permit_remote_node` right in the ModelOS Thread Security Register (see [6] volume 2 chapter 26 section 4). This is achieved by executing the kernel instruction

```
refine_tc_rights (Bitlist rights; Boolean primary)
```

To assist the management of the bit list and thus help eliminate errors, the Timor environment should provide a number of predefined constants corresponding to the bit patterns which define the security settings. An appropriate such bit list (in this case one which has the `permit_remote_node` unset and the remaining bits set) might for example be called `remoteNode` and could be defined in an enumeration:

```
enum TCRsettings {remoteNode, foreignCalls, foreignFileCaps,
foreignCodeCaps, download, upload, subthreads, callBacks,
websites, mail, ftp, otherInternet}
```

From this a set[61] of enumeration values might be declared in the following statement:

```
instance:
Set<TCsettings> TCset = {TCsettings: remoteNode,
foreignCalls, foreignFileCaps, foreignCodeCaps, download,
upload, subthreads, callBacks, websites, mail, ftp,
otherInternet};
```

Now we can declare an actual set

```
const TCset noForeignNode = {foreignCalls, foreignFileCaps,
foreignCodeCaps, download, upload, subthreads, callBacks,
websites, mail, ftp, otherInternet};
 // note that remoteNode is absent
```

All of this (and constants with other values removed) already exists for the programmer, so that all he has to do to prevent the current thread from being transferred to another ModelOS node is to execute the following kernel instruction:

```
kernel.refine_tc_rights (noForeignNode, true); /* the second
 parameter indicates whether the setting is intended
 for the primary or secondary control rights in the TSR */
```

[61] actually a collection literal, i.e. a List, see chapter 13 section 4.1.

3.3 Normal Execution of Inter-Module and Similar Calls

If a programmer executes an inter-module or similar (e.g. co-module) call this is normally directly handled by the compiler. Suppose for example that a module `myMod` contains a capability for another module `yourMod`:

```
type myModType {
instance:
... // instance method definitions
}
```

and is implemented as:

```
impl myModType::Impl{
state:
 YourMod** yourMod;
constr:
    ... // initialises myFile
instance:
 op void aCall() {
  ...
  yourMod.aCall(); // an inter-module call
 }
}
```

then the Timor compiler will generate code to make the inter-module call from an instance of `myModType::Impl` to the method `aC` of the module `yourMod`. If `aCall` has parameters it will take care of these, so that inter-module calls (and co-module and library calls) are no hassle for the programmer.

3.4 Callback Calls

ModelOS supports callback modules, i.e. modules which can call their caller back (see [6] chapter 18 section 9, chapter 20 section 8.5 and chapter 28 section 7). To activate a routine in a call back module from a normal module which was invoked by the same call back module the programmer uses the pseudo variable (i.e. module) name **`callback`**, e.g.

```
callback.myCallBackRoutine(parameters).
```

where `myCallBackRoutine` is the name of a callback routine in the module of the caller of the current module.

3.5 Direct Execution of Inter-Module and Similar Calls

Situations may arise in which a programmer needs more control over inter-module and similar calls. In such circumstances it would be a mistake to think that this can simply be carried out by making a straightforward kernel call to the inter-module call instruction! The reason for this is that inter-module and similar calls have to deal with two distinct sets of parameters, i.e. those to be passed by the user to the called module and those needed by the kernel to carry out the call (e.g. the number of the called semantic routine for the destination module).

The ModelOS rule for passing kernel parameters is that they are passed in a segment addressed by the calling module's segment register 15[62]. Three kernel parameters are needed to call the IMC directly are:

a) a module capability for the module to be called;

b) an integer indicating the entry point number of the routine to be called:

c) a boolean parameter indicating whether the caller is requesting read-only or read-write access to the module's file data.

On a normal inter-module call (see previous subsection) the Timor compiler organises this, and it can also do this when the user makes a direct inter-module call as follows:

```
kernel.IMC(myCap, 11, true);
```

where `myCap` is a capability for the module to be called, `11` is the number of the semantic routine to be activated and `true` indicates that read-write access is required. If the programmer does not know the number of the semantic routine he could theoretically use the corresponding Template Manager for the module to be called (see [6] chapter 32 section 2.2), but in practice it would be simpler for the Timor compiler to relax the normal rule by allowing the user to supply the name of the semantic routine in place of the number, since it must contain the relevant code anyway.

We now consider how the *user* parameters are passed. These must be prepared before the direct IMC using the kernel instruction `create_imc_params`[63] (or in the case of co-module calls and library calls `create_pc_params`[64]). These instructions prepare space for the user parameters on the kernel stack of the current thread. At the ModelOS level the user can then address these using segment registers[65] and at the Timor level the compiler organises this by placing the user's parameters in the appropriate segments.

In the special case under discussion the compiler can save the programmer from having to address the parameter segments via segment registers and so can make the following kernel call:

62 For more details see [6] volume 2 chapter 17.

63 see [6] chapter 20 section 8.

64 The difference is that CMCs and LCs can pass pointers as parameters.

65 see [6] chapter 20 section 6.

```
kernel.create_imc_params(...); /* the user parameters as they
        would appear in a normal inter-module call at the
        Timor level */
```

The same applies *mutatis mutandis* to the `create_pc_params` kernel instruction.

4 Synchronisation and Semaphores

Dijkstra's original semaphore proposal [30] provides the basis for synchronisation in ModelOS, as is discussed in detail in Chapters 8 and 21 of [6]. The operations in question include the P (claim) and V (release) operations of Dijkstra, but further semaphore types have been added, as will become evident below.

At the ModelOS level the semaphore `suspend` operations have a parameter (`Thread** theThread`) which was not envisaged in Dijkstra's original proposal; this is added for security reasons (see Chapter 21 of [6]). It is a capability for the currently active thread, which is used to suspend the thread. This does not appear in the following descriptions, but in the ModelOS environment the compiler must add the appropriate code. For implementations in conventional systems this extra parameter is not required.

Timor hides the complexity of the operations from programmers by providing the following built-in types.

4.1 General Semaphores

For this purpose there is a type `Sem`, which is defined as follows:

```
library type Sem {
instance:
 op void p(); /* This corresponds to Dijkstra's P operation;
    it claims access to one of the resources controlled
    by the semaphore instance */
 op void v(); /* This corresponds to Dijkstra's V operation;
    it releases access to one of the resources controlled by
    the semaphore */
}
```

From the viewpoint of the normal programmer this type has a standard implementation `Impl`. Its constructor has a single integer parameter which sets the initial value of the semaphore. When used for mutual exclusion the initial value should be set to 1.

4.2 Resource Set Semaphores

This semaphore type [31] enhances the general semaphore by advising the caller (as a return value of the V method) of the resource which has been allocated.

```
library type RsetSem {
instance:
 op Int p(); /* This claims access to one of the resources
    which it controls, indicating in the return value
    which resource has been allocated */
 op void v(Int theResource);  /* This releases access to
    the resource indicated in its parameter */
}
```

The standard constructor has a parameter indicating the number of resources controlled. The maximum value of this parameter is 64. The actual resources are numbered 0..63

4.3 Reader-Writer Exclusion

Reader-writer exclusion [32, 33] is achieved by using a semaphore of type `RwSem`, defined as follows:

```
library type RwSem {
instance:
 op void readp(); /* This claims shared reader access
    to the resource which it controls */
 op void readv(); /* This releases shared reader access to
    the resource which it controls */
 op void writep(); /* This claims mutually exclusive
    writer access to the controlled resource */
 op void writev(); /* This releases mutually exclusive
    writer access to the controlled resource */
```

From the viewpoint of the normal programmer this has a standard implementation `Impl`, which provides reader priority and a second implementation called `WriterImpl`, which provides writer priority. Neither constructor has parameters.

4.4 Access to Basic Semaphore Variables

In order to allow basic access to synchronising variables, for example in order to implement priority semaphores or to synchronise variables of other types, the following can be used.

```
library type BasicSem {
instance:
 op Int dect(); /* This indivisibly decrements an internal
    counter then returns its value */
 op Int tinc(); /* This indivisibly copies the value of the
```

```
      an internal counter (to be returned) and then
      increments it. */
  }
```

From the viewpoint of the normal programmer this has a standard implementation `Impl` with a single integer parameter, which is used to initialise the internal counter. The operations and use of this type are described in Chapters 8 and 21 of [6]. In ModelOS they are intended for use in conjunction with the Thread Scheduler's `suspend` and `activate` methods.

4.5 Higher Level Synchronisation

These and other synchronisation mechanisms (see Chapters 8 and 21 of [6]) can be accessed by Timor compilers via ModelOS library calls (LC instructions) (see section 3.2. above) and can be used to build further synchronisation libraries (e.g. to support OpenMP applications[66]).

[66] see https://en.wikipedia.org/wiki/OpenMP

Chapter 16
Why Timor Does Not Need Wildcards

When Java introduced genericity it also introduced the idea of wildcards to solve a number of problems which were not present in earlier versions of Java, but which arose as a consequence of introducing genericity [27, 34].

Torgersen et al. [34] gave as a fundamental reason for this that the inclusion of genericity alone "lacked some of the flexibility associated with object-oriented subtype polymorphism" (p.98). In particular without wildcards (or a similar mechanism) there would be "no general way to abstract over ... different kinds of lists to exploit their common properties, although polymorphic methods may play this role in specific situations".

At first sight the idea of wildcards appears to be very convincing. Yet when we examined whether this concept should also have a place in Timor we found that it was not necessary, since it appears that existing Timor concepts (together with a relatively trivial extension) provide an adequate basis for supporting the kinds of situations in which Java programmers can apply wildcards.

1 Upper Bounded Wildcards

Under normal OO subtyping rules a `List<Circle>` is not considered to be a subtype of a `List<Shape>`, despite the fact that `Circle` might be a subtype of `Shape`. This is because the normal subtyping rules are concerned with the subtypes of the collection types, not those of their elements. Thus a `List<Shape>` can be assigned or passed as an argument to a `Collection<Shape>` or even to an `Ordered<Shape>`, but a `List<Circle>` cannot.

This point is unfortunate, because it prevents useful applications. Bracha points out in [27] that because normal subtyping rules are concerned with the subtypes of the *collection* types, not those of their *elements*, it is not possible to assign, say, a `List<circle>` to the *element* supertype, e.g. a `List<shape>`, despite the fact that allowing assignments of this kind creates no problems in cases where the method would merely *read* shapes from the list.

In order to provide a better understanding of Timor's alternative approach, it is helpful first to understand the issues in more detail than the above short outline suggests.

1.1 Why Is a List of Students not a Subtype of a List of Persons?

It becomes clear why a `List<Student>` is not a subtype of a `List<Person>` if, for example, we try to insert an element of type `Person` into a `List<Student>` instance via a `List<Person>` variable. The following (incorrect) Timor example illustrates the problem.

```
Person* p = new Person&s.init();
p.name = "fred";
p.dob = "1/1/1981";
List<Student*>* sList = new List&s<Student*>.init();
// insert some students
List<Person*>* pList = sList;
pList.insert(p); // inserts a Person into a student list
for (s in sList) {if (s.uniName = "Oxford") ...}
```

This example contains three errors.

1) In real Timor the compiler would flag an error in the line containing the assignment `pList = sList`, because a `List<Student>` is not a `List <Person>`. This compile time error prevents the following two errors from occurring at run-time, and is based on the (correct) assumption that a collection of elements of type E is not a supertype of a collection of E2, where E2 is a subtype of E. However if the programmer who naively wrote the above code did not realise this, and if the compiler were simply to flag this as an error this does not explain to the programmer why a `List<Student>` is not a `List<Person>`.

2) If the compiler writer were not aware of the error and incorrectly allowed the program to compile, the next line would cause a run-time error, because a run-time check would discover that the parameter is not of an appropriate type (i.e. in this example it is not a reference for a `Student` or a subtype thereof).

3) Suppose now that the run-time system were faulty and did not raise a run-time error as described in point 2, then we would inevitably have a run-time error in the next line. In the course of iterating through `sList` we would reach `p` and because this is a `Person` without any `Student` details such as `uniName` it would have to generate a run-time error!

A little consideration of the example shows that the error described in 3) only occurs if a new element is inserted into the collection via a supertype variable. It does not necessarily arise with all write operations at the supertype level. For example removing an element from a list (or clearing an entire collection) would be harmless in this respect.

The above example in fact shows that (a) it is unsafe to allow elements to be added via a "supertype" variable, and (b) it is unsafe to allow methods to be called where dynamic binding checks would fail.

This situation is unfortunate. If we simply draw the conclusion that the assignment of a collection of subtype elements to a collection variable of supertype elements is always forbidden, that would rule out many useful situations where no errors would occur and in which the code could be used without causing a run-time error. That raises the question of how a compiler might allow programs to be accepted which avoid the errors described above, and yet appear to break the normal subtyping rule.

1.2 The Wildcard Solution

To make this possible Java introduced upper bounded wildcards, in the form `List<? extends Shape>` to identify those variables or parameters to which such assignments could safely be made, thus modifying the subtyping rules. In the following we refer to such generically defined variables and parameters as "element subtype assignable".

1.3 The First Timor Solution: Generic Co-Types

In fact it is quite possible in Timor to achieve the aim behind this by using co-types, as we have already shown in chapter 11, section 15. That solution eliminates the need for upper bounded wildcards in cases where the co-type adjustment technique automatically produces an additional method which provides the required functionality and has the correct parameter type.

However there are situations in which an appropriate method cannot be automatically generated, due to the nature of the problem. Conversion of types is such a case. The collection co-types in the TCL provide standard makers which can convert any collection of elements to any other type of collection of elements, where the elements have the same type. For example a `List<Person*>` can be converted into a `SortedSet<Person*>` using the following maker[67], defined generically in `SortedSet&s` as follows:

```
op TheType convert(Collection<ELEM>*** c1) throws NullEx;
```

[67] We ignore the issue of choosing an implementation for the converted result to keep the issue as simple as possible.

When actualised using `Person*` elements, there is no problem in converting say a `List<Person*>` into a `SortedSet<Person*>`. To carry out the required conversion the application might use the code fragment:

```
List<Person*> myPersonList = ...;
SortedSet<Person*> myPersonSortedSet =
                     personSortedSet&s.convert(myPersonList);
```

The standard co-type variable `personSortedSet&s` exists automatically and has the type `SortedSet&s<Person*>`.

However if the application programmer wishes to convert a `List <Student*>` into a `SortedSet<Person*>`, he cannot use this method, despite the fact that `Person` is a supertype of `Student`, because it would break the normal subtyping rule.

What alternatives does he have in the context of co-types? He can only use a further co-type of `SortedSet`, because only makers in `SortedSet` can do that.

He might attempt to define a new co-type which extends (or includes, but does not adjust!) the actualised co-type `SortedSet&s<Person*>` as follows:

```
type MySortedSet&s<Person*> expands SortedSet<Person*>{
extends: SortedSet&s<Person*>;
maker:
 op TheType convert(Collection<Student*>*** c1)
                                         throws NullEx;
}
```

This overloads the original method, replacing the parameter with one which will achieve the required aim. So far the attempt is valid, and it would then be possible to provide a new implementation along the following lines:

```
impl MySortedSet&s<Person*>::Impl {
state:
 ^SortedSet&s<Person*> mySortedSetCoType =
                              SortedSet&s<Person*>::Impl();
maker:
 op TheType convert(Collection<Student*>*** c1)
                                         throws NullEx{
 /* new implementation code or the code copied from the
    original */
 }
}
```

This would also be valid. The re-use variable would cause all the methods from the original co-type to be matched. The matched methods thus become implementation code for the corresponding interface methods in the new type. But the overloaded `convert` method cannot be matched for re-use because of the modified parameter type. This is unfortunate, since the code would achieve the aim, and could therefore be copied into the body of the new method.

Since our aim of achieving the equivalent of upper bounded wildcards includes the aim of re-using code, and since the development of new co-types is time consuming, we conclude that this is not an adequate solution. Furthermore such a solution is not a general solution for the basic problem.

1.4 Using Restricted Variables and Parameters as an Alternative to Upper Bounded Wildcards

This alternative Timor approach relies on the fact that the methods which can be called via a variable (or parameter) can be restricted to a particular view or views, in effect forming access rights which can be enforced by the compiler (see chapter 15 section 2). Although this technique was introduced into Timor primarily for protection purposes, it is relevant to the present discussion.

Some views which can be used in restrictors are standard. For example to enforce read-only access to an instance via a particular variable or parameter the standard `enq` view can be used, which then only allows methods defined as enquiries to be invoked. To ensure that protection cannot be side-stepped simply by then assigning the instance to a further variable or parameter which does not have a restricted view, Timor checks that assignments are not permitted to variables or parameters with fewer restrictions than that from which an instance is being assigned.

Using the `enq` view would in fact be sufficient to solve the conversion problem described above, if we were to combine it with a modification of the subtyping rule. However, that does not provide a general solution for the issue. As we saw in subsection 1.1 all that is really necessary to solve the general problem is to ensure that elements are not added and that methods are not invoked which would fail because of the dynamic binding rules.

1.5 The Restrictor OOPS

In fact it is possible to restrict a co-type (or a normal type) from inserting elements into a collection by using the restrictor [: all – insert :], which allows all the methods of the variable (or parameter) to be invoked except the `insert` methods[68]. This would permit all the methods, including for example `remove`, `loopInit` and `getnext` (but excluding all the `insert` methods) to be invoked. We therefore introduce this as a new view restrictor, which for convenience we

[68] see chapter 15 section 2.

call `oops` (an expression for surprise or apology), which indicates that the associated variable or parameter is "element subtype assignable". Like the `enq` view, this is not explicitly defined as a view, because, like `enq`, the methods involved change from type to type.

Let us now return to the example in section 1.1, where the final three lines were:

```
List<Person*>* pList = sList;
pList.insert(p); // inserts a Person into a student list
for (s in sList) {if (s.uniName = "Oxford") ...}
```

As it stands this is still an error, but if the programmer had written the first of these lines as

```
List<Person*>*[:oops:] pList = sList;
```

then it would no longer be an error. On recognising the error in the original first line, the compiler could of course suggest to the programmer that his error could be corrected by using the `oops` restrictor.

2 Unbounded Wildcards

In Java unbounded wildcards are supported in the form `Collection<?>`, which is considered to be the supertype of all collections, or for example `List<?>`, the supertype of all lists, etc. The idea is that a collection with some unknown element type can be declared as a variable or passed as a parameter, and that collection methods which are independent of the type of the element (e.g. methods which clear a collection or return its length) can be invoked on an instance of some real (actualised) collection which has been assigned to such a variable, without the actual type of the elements being known at that point in the program. Furthermore, elements from such a list can be read (as instances of the Java type `Object`) but new elements cannot be inserted, because the actual type of the objects is unknown. This suggests that the unbounded wildcard `Collection<?>` in Java can be viewed as a special case of an actualisation of an upper bounded wildcard, i.e. `Collection<? extends Object>`. This observation provides the clue as to how Timor can implement an equivalent mechanism.

2.1 Handles and Unbounded Wildcards

Torgersen et al. motivate the idea of unbounded wildcards with an example in which the objects in a `List` can be cleared (or for example the length of the list interrogated), since these are methods which require no knowledge of the actual element type. In fact they describe the type `List<?>` as

"a supertype of `List<T>` for any `T`, which means that any type of list can be assigned into the list field. Moreover since we do not know the actual element type we cannot put objects into the list. However, we are allowed to read `Objects` from it — even though we do not know the exact type of the elements, we do know that they will be `Objects`." [34].

In Timor the type `Handle` is the supertype of all references, capabilities and values of all types, and is itself a type. Unlike Java's unbounded wildcard it is not a generic feature, and unlike Java's class `Object` it has no methods. The initial consideration in introducing this type into Timor arose from the aim of allowing operating system software to be developed (see chapter 7, section 8). However, the Timor type `Collection<Handle>`, if used together with an appropriate restrictor, can in practice be used in Timor to achieve almost all the aims of Java's `Collection<?>`, as we now demonstrate.

First, Timor collection methods such as `size` and `remove` can be invoked on any collection instance underlying a `Collection<Handle>[:oops:]`, as was discussed in the previous section.

Second, in conjunction with this restriction it can serve as the destination of an assignment of any collection instance, and consequently it can be regarded as the "supertype" of all collections, analogously to `Collection<?>` in Java.

Third, any element of such a collection can be read, can be assigned to a `Handle` variable and can be cast to its actual type. This follows in that `Handle` is a supertype of every type.

Here is an example:

```
Person* p = new Person&s.init("fred");
// create a new Person object
List<Person*>* myPersonList = new List&s<Person>.init();
// create a Person List
myPersonList.insert(p);
// insert p into the list
List<Handle>[:oops:]* myListSupertype = myPersonList;
// assign the list to a restricted handle list
Int i = myListSupertype.size();
// then discover the size of the list
Handle firstEntry = myListSupertype.get();
// recover the first entry in the list
cast (firstEntry) as {
 (Person myPerson) {...}
// try to cast it to a type Person
}
```

This demonstrates that Java's unbounded wildcards can be easily simulated in Timor without the introduction of wildcards.

2.2 Further Aspects of Working with Handles

Since `Handle` has no methods, the only thing which it can usefully achieve is to store an instance into some structure (e.g. a directory) and/or to cast it to a type which has methods.

A collection of some specific type can be assigned to say a `Collection <Handle>` variable restricted to `[:oops:]` as follows:

```
Collection<Handle>[:oops:]* myHandles =
                                        new List&s<Person*>.init();
```

and the type could be retrieved using a cast statement as follows:

```
cast (myHandles) as {
 (List<Person*>[:oops:] personList) {...}
 (List<Shape*>[:oops:] shapeList) {...}
 else ...
}
```

At first sight it perhaps appears that the `oops` restriction is necessary in the body of the **`cast`** statement (as shown), in view of the restriction rule that an instance cannot be assigned from a restricted variable to another variable unless the restriction on the latter includes at least that of the former.

However, casting is not the same as making an assignment, and the use of a cast ensures that code executed within the cast body is only executed if the cast condition is true. Hence we can, without risk, remove the restrictions within the cast, as follows:

```
cast (myHandles) as {
 (List<Person*> personList) {...}
 (List<Shape*> shapeList) {...}
 else ...
}
```

This then raises the further issue whether say a `List<Student*>` which has been assigned to a `Collection<Handle>` can be cast to a `List<Person*>`. Since this would create the risk that within the cast statement `Person` references could be inserted into the list, the cast can only be successful if the restriction is applied, i.e.

```
cast (myHandles) as {
 (List<Person*> personList) { /* this cast fails if
                              myHandles is a List <Student*> */}
 (List<Person*>[:oops:] personList) {/* this cast
```

```
                succeeds if myHandles is a List <Student*> */}}
  }
```

The general rule is that a cast will only succeed where a similar assignment would also succeed.

2.3 Casts Involving Co-types

Bracha motivates unbounded wildcards with an example in which the objects in a `Collection<?>` can in turn be manipulated in some way using the methods of Java's type `Object` (e.g. `println`). We now consider how Braccha's example translates to Timor.

A significant difference between Java and Timor comes to play in this example. Whereas in Java `println` is a method of `Object`, in Timor there is nothing quite equivalent to `Object`, since `Handle` has no methods. Instead, generally useful methods such as `println` are included in views.

A view can be inherited in any type, including co-types. The method `println` is defined in a view `StandardIO` which is typically included in co-types designed to carry out input/output operations for their expanded type (see chapter 11, section 15). Standard co-types for input-output are typically included in co-types which have the suffix `&io`.

In order to print an instance of an unknown type (assigned in Timor to a `Handle`) we must first cast it to the view `StandardIO` in its co-type, since there is no guarantee that this has been inherited in the standard `&io` co-type. A normal cast statement will not achieve this for us, for two reasons. First, it will cast us to the type of the instance, not its co-type. Second, because we do not know the type of the objects, we cannot cast it at all. Consequently we need a cast statement which will both find the co-type and will cast to its `StandardIO` view.

This is achieved in a second form of cast statement, known as a co-cast. The following example assumes that a collection of handles is passed to a method which prints each entry in the collection in turn. It further assumes that the appropriate method is held in a standard input-output co-type with suffix `&io`.

```
  enq void printCollection(Collection<Handle>[:oops:] c) {
   for (Handle h in c) {// iterates through each handle in turn
    cocast (Handle h, h&io) as {/* h&io is a pseudo-variable
     which refers to the (unknown) standard io cotype of h */
     (StandardIO p) {h&io.println(p);}
     // the cast is to the view StandardIO in h&io
    }
   }
  }
```

3 Lower Bounded Wildcards

Lower bounded wildcards can be seen as the dual of upper bounded wildcards, allowing a supertype of a known type to be assigned to a variable or passed as a parameter. In this section we consider whether a similar facility is needed in Timor.

3.1 Comparators and Co-Types

Lower bounded wildcards are motivated in [27, 34] by comparators, which are used in Java to provide flexible criteria for making comparisons between the elements of generic objects. The actual example used in both papers is a tree set, which, in the words of Bracha's tutorial, "represents a tree of elements ... that are ordered" (see [27], p.19). The comparator is needed in order to provide a flexible criterion for sorting the elements of the tree set into a defined order.

In Timor `TreeSet` is a possible implementation of the type `SortedSet`, which can be flexibly ordered using the generic static function parameter PRECEDES, as is described in detail in chapter 14, section 3.

Furthermore, Java comparators can be totally eliminated by the use of co-types. In a further example, which motivates the need for lower bounded wild cards in Java via comparators, Bracha discusses the Java `Collections` method

```
T max(Collection<T> coll)
```

which returns the maximum element in the collection `coll`. This method appears in Java's `Collections` rather than simply as an instance method of `Collection`. The problem would simply disappear in Timor if, for example, a method `max` were defined as an instance method of the TCL's type `Collection`, as follows:

```
template <ELEM>
type Collection {
instance:
 ...
 enq ELEM max()throws NullEx();
}
```

The implementation of this would simply use a normal comparison operator (e.g. <) to compare the elements. This works independently of the type of the elements, because of the rules described in Appendix II section 5 (which describe the defaults used by comparison operators) and section 9 (which describe how binary methods in co-types can be used to override the default values).

However, using this approach could lead to a large number of methods being added to the basic `Collection` methods, resulting in lack of clarity, and also making it difficult to add further methods in a component-oriented way. To avoid these problems Timor adds a further category of methods to co-types, indicated by the keyword **library**[69]. Like other co-type method categories these can appear in any co-type, making it easy for applications (and for software component developers) to add new library methods. However there is a standard library co-type suffix `&lib` for standard libraries, in which `max` and similar methods typically appear:

```
template <ELEM>
type Collection&lib expands Collection {
library:
 enq ELEM max throws NullEx();
 ...
}
```

Similar considerations apply to other methods in Java's class `Collections`.

3.2 Type Matching and Casts

Bracha provides a further motivation for lower bounded wildcards which is unrelated to comparators, i.e. a method `writeAll` which iterates through a generically defined collection and "sinks" the contents into a generically defined sink, returning the last element to the caller of the method. Lower bounded wildcards are needed to ensure that the types of the two generic parameters are correctly matched. As Bracha himself hints, this is a somewhat unrealistic example (p.19).)

In part the unrealistic nature of this example, the lack of a clear definition of the purpose of the example together with the fact that there are major structural differences between Java and Timor, make it difficult to present a one-to-one translation of the problem. Instead we have concentrated on what we consider to be the central issue, viz. that the content of a generically defined input parameter (in Bracha's example a `Collection<T>`) can be output to a generically defined output parameter involving a supertype of the generic parameter (Bracha's `Sink<? super T>`).

Exactly this can be achieved in Timor by invoking makers based on `Collection&s`. To help make the comparison clearer we have avoided the use of the covariant adjustment keyword `TheType`, replacing it by actual types:

```
maker:
 enq Collection<ELEM> convert
```

[69] These are not directly related to *modules* defined as **library** *types* (see chapter 3 section 3.

```
          (Collection<ELEM>***[:oops:] c) throws NullEx;
```

Using this maker method in the appropriate adjusted co-type, one could for example copy the content of a List<String> into a Collection<Handle>, as follows:

```
Collection<Handle>* handleColl;
List<String>* stringList = new List<String>::LinkedImpl();
... // put strings into stringList
handleColl = handleList&s.convert(stringList);
```

The only substantial difference from the Java example is that this method returns a Collection<Handle>* rather than the last element in the collection. However, it is not problematic in Timor to read out the last element and cast it.

Finally, we note that in Timor it would be easy to retrieve a List <String>* from an appropriate Collection<Handle>* via a cast statement. In Java this is not possible, because through the use of erasure as the generic implementation technique all generic information is erased at run-time. However in Java it would be possible to define a method for achieving an almost equivalent effect by using a lower bounded wildcard.

3.3 Complementing Casts with Restrictions

There are however some situations in which the Timor mechanisms which we have so far found useful are insufficient to allow a method to be adequately defined generically in a situation where lower bounded wildcards could be used. Consider a method, which in Timor one would expect to appear as a maker in a co-type, viz. a method which creates a collection of some specified type by selecting elements of this type out of a collection of elements of a supertype. In the following example we wish to create a List<Student> by iterating through a List the elements of which are of some supertype of Student (e.g. Person and/or Handle).

Such a method could of course be defined, but with the techniques so far mentioned it could not be defined generically. Here is how a non-generic version might look:

```
enq List<Student> convert(Collection<Handle> c){
 List<Student> myList = List<Student>::LinkedImpl();
  {for (Handle h in c){
    cast (h) as {
    (Student s){myList.insert(h)}
 return myList;
  }
}
```

The key restriction which applies is that the input parameter must be "element supertype assignable" – in this case a collection of a supertype of `Student`.

That can be easily checked by the compiler if we have a restriction to advise it of this condition. Such a restriction has the name `wow` (an exclamation expressing admiration), allowing us to define the method generically in `List` as follows:

```
enq List<TT> convert(Collection<TT>[:wow:] c);
```

An implementation in `List&s` could be as follows:

```
enq List<ELEM> convert(Collection<ELEM>[:wow:]*** c){
 List<ELEM> myList = List<ELEM>::LinkedImpl();
  for (Handle h in c){
  // Any element of any type can be treated as a Handle!
    cast (h) as {
    (ELEM s){myList.insert(s);}
    }
    return myList;
  }
}
```

The relevant rules for the cast restriction are that:

– only "element supertype assignable" instances of the actualised type can be assigned to variables or parameters restricted by the `wow` restriction.

– variables and parameters restricted by `[:wow:]` can only be used in a cast statement, and only then without the `[:wow:]` restriction.

The reason for restricting its use in cast statements (in which the normal type rules apply) is to ensure that the type rules are always upheld. This rule would for example prevent a programmer from writing a statement such as

```
List<Student>[:wow:] studentlist = List<Person>::Impl();
```

With the rule that cast statements are necessary this statement would be flagged as a compile time error if it appeared outside a cast statement. If the programmer were to attempt to achieve the same aim in a cast statement, as follows,

```
cast (List<Student>[:wow:] studentlist) as
 (List<Student> slist) {slist = List<Person>::Impl();}
 (List<Person> plist)  {plist = List<Person>::Impl();}
```

then the first cast option would be flagged by the compiler as an illegal assignment according to the normal subtyping rules, while the second would be legal.

Notice that the restriction requires either that the entire instance underlying the `wow` variable or parameter is cast (as just illustrated) or that the individual elements are cast, as in the convert example above.

The advantage of the `wow` restriction is that the programmer knows that the instance behind the variable must have elements which are supertypes of the statically defined elements, thus limiting the list of cast options.

Chapter 17
Concluding Remarks

Chapter 1 indicated that Timor has been designed with four primary aims in mind, i.e.

i) to provide a suitable programming language for developing application programs for the ModelOS system;

ii) to provide a programming language which facilitates the development of software components that can be easily re-used in many individual systems, both as large and small components;

iii) to eliminate certain problems which arise in the context of other popular object-oriented programming languages;

iv) to provide strong support for database applications.

In this final chapter we briefly review how Timor achieves these aims.

1 Support for ModelOS Applications

Timor supports the idea of direct addressability in a persistent virtual memory in the simplest way possible, i.e. by simply ignoring the issue. The programmer just assumes that any value, object or module is persistent and can be addressed directly. (ModelOS is responsible for ensuring that this is the case.)

ModelOS requires Timor to support the idea of information hiding in a very strict way, based on entirely procedural interfaces, in order to ensure that some of its key protection and synchronisation mechanisms function correctly. In fact information hiding units are the only basic structure in Timor; there is no separate mechanism for writing programs nor for organising data structures into files[70]. Nevertheless its idea of abstract variables (see chapter 5) offer the programmer a very comfortable mechanism for achieving information hiding.

[70] If Timor is to be used in conventional systems conventional programs and files can be hidden behind a module interface.

Timor makes at least three important contributions to ModelOS's aim of providing a very secure system:

- It distinguishes between reader and writer methods, via `enq` and `op` keywords.
- It introduces qualifying types with bracket routines (see chapter 10).
- It permits restrictors to be associated with variables and parameters (see chapter 15, section 2).

Distinguishing between reader and writer methods is particularly important to support both the synchronisation of threads and various ModelOS security mechanisms.

Qualifying types inter alia provide a mechanism which allows the programmer to confine information to a module, without letting it escape via intermodule calls (thus providing a general solution to one of the most serious problems facing the modern computing industry) and it also allows the same mechanism to be used for calls between the individual objects in a single module.

Similarly, restrictors allow programmers in effect to reduce access rights in capabilities but also (by associating them with individual variables and parameters within a module) to ensure better access control even at the programming-in-the-small level.

Chapter 15 describes how Timor also provides support for other aspects of the ModelOS operating system at a technical level. For example Timor can be used to call the ModelOS kernel instructions directly, and thus allows it to operate as a lower level language than is usual in many other programming languages, including the setting and reducing of access rights and finer control of the invocation of modules as well as access to synchronisation variables.

2 Flexible Support for the Design and Development of Software Components

The use of a single information hiding structure to create value variables, objects and modules allows a programmer (or better, a software component manufacturer) to create standardised small variables which can be used in many frequently occurring situations. For example it becomes possible to standardise such items as names, addresses, telephone numbers, dates, passport numbers and much more, and then build these into larger level items, such as persons, telephone directories, marriage documents, etc.

There are advantages in developing these small units as separate components. First, assuming that software component manufacturers sell such items off the shelf (and provide the buyers with specifications) this would reduce the work of applications designers and programmers, and – just as important – it would save the need for applications programmers to test them (which becomes the job of the original component manufacturer).

Second, the component manufacturer can afford to spend more time than the average applications programmer on a particular task and thus provide a more flexible and more complete product. A good example to illustrate this point is a *date* item. Often applications programmers, who are usually under pressure to complete their work quickly, will just produce a trivial item, without thinking of leap years and century changes, etc. Those old enough will recall the panic which occurred throughout the software industry as the beginning of the 21st century approached! And the fact that the information hiding modules have procedural interfaces (which of course can be partly hidden by the idea of abstract variables (see chapter 5)) means that not only the actual data (in this case the date) can be programmed, but for example enquiries which calculate the day (Monday to Sunday) on which a future or past date occurs, etc. can be provided.

The idea of co-types as such is also a very flexible idea for adding components. These are add-on units which can provide useful extra facilities, such as an enquiry which calculates the number of days (and hours and minutes and seconds) between two dates, etc. Small components can be quite complicated and are very useful.

Components provide a very important way of standardising software. Given this approach, many application systems can be built largely from standardised units. This applies not only to the very small components which we have so far discussed, but also to composite objects such as person records, which, when declared as objects, can cross-reference each other (e.g. in a spouse field) and can be further organised into more substantial structures, e.g. a family tree. Similarly they can reference separate modules (e.g. a birth certificate, a marriage certificate), which could also be accessed remotely (via remote inter-module calls) in various government departments and in company databases.

3 Improving the Object Oriented Paradigm

Here are some of the improvements which Timor makes to the OO paradigm[71].

i) Co-Types:

Timor's insistence on declaring only instance methods in types and defining co-types as instance methods which function as class methods, binary methods and makers (application oriented constructors) and even as input-output add-ons

[71] It is clear that some purist OO fans will not regard some of these as "improvements", but may see them as "heretical".

solves a number of issues which can arise in the conventional OO paradigm. Here are some of them:

a) a number of issues which arise with binary methods to which Bruce et. al. have drawn attention [26];

b) the problem of combining both application-oriented and implementation-oriented parameters in conventional constructors [3].

Timor also shows how a co-type hierarchy can be automatically *adjusted* to produce a covariant parallel hierarchy for the subtypes of the original base type.

ii) Separating Types and Implementations:

The separation of types and implementations leads to the concept of re-use variables, which provide an efficient mechanism for re-using code, even in implementations of formally unrelated types (e.g. re-using a queue implementation in a double-ended queue – or vice versa –, although these do not have a genuine type-subtype relationship).

iii) Views:

Timor distinguishes *views* from abstract types, allowing them easily to be incorporated into larger objects in an adjectival way (often based on adjectives which end in "-able", such as *switchable*, *openable*, etc.

iv) Types which are often difficult cases in normal object orientation:

Timor provides mechanisms (e.g. 'parts') which simplify the modelling and implementing of problem areas such as repeated inheritance and diamond inheritance.

4 Support for Database Applications

The vast majority of real world applications are based on large – and ever growing – databases (e.g. in government departments and in commercial organisations). For this reason an important aim of Timor has been to provide an adequate support for database applications, based on the Timor Collection Library (TCL).

Unlike Timor, most other OO programming languages are not designed as persistent programming languages. This issue has been clearly emphasized in the work of groups around Malcolm Atkinson and Ron Morrison [35], which have emphasized the need for "persistent programming", developing a number of persistent programming languages starting with PS-Algol. However, they did not have the advantage of a persistent virtual memory system such as ModelOS. Consequently they aimed to create a persistent programming language which runs in a non-persistent environment. As a result, their emphasis was quite different from that found in Timor. In contrast, Timor is fortunate enough to be able to ignore some of the problems which they tackled, and the TCL is programmed without the constraints which faced them.

We now consider how Timor supports database programming.

4.1 Remote Databases

In view of the ModelOS approach to remote processing, which is achieved automatically via remote inter-module calls, the programmer can ignore this issue entirely.

4.2 Separating Types from Implementations

An advantage of Timor from the viewpoint of databases is that type definitions in the TCL present a logical interface which is used in application programs to access a database independently of how it is actually organised, while implementations of a type provide the actual accessing mechanisms. For the same type various different implementations can exist, and these can be equivalent to 'normal' programming structures which computer science students learn in a data structures course, e.g. arrays, trees, hash tables, linked lists, circular lists, doubly linked lists, etc. but implementations can also provide structures typically associated with file systems and database systems, such as indexed sequential, B-trees, etc.

Since application program accesses to a database are all framed at the type level, it is possible to convert a TCL database from one implementation to another without needing to change the application programs which use it. This is easily achieved using the `convert` maker (see chapter 13, section 3.1).

It is also possible of course to use different implementations for different databases containing the same type of elements.

4.3 Kinds of Database

4.3.1 Relational Database Model

While Timor is not based on the relational database model[72] [36], it can be used to create persistent files corresponding to tables (relations), in which its records can be seen as tuples. Using Timor's 'selection by predicate' facility for subcollections (see chapter 13 section 4.2) new relations can be created which select specific tuples from an existing relation (file) according to defined criteria, and using the 'selection by predicate' facility for elements an individual tuple can similarly be selected (chapter 13 section 4.3) which conforms to specified criteria.

[72] see https://en.wikipedia.org/wiki/Relational_model

It would be possible for an SQL[73] (or similar) module to be developed which builds upon these facilities and on the collection operators (chapter 13 section 4.5 and 4.6) to create a database query language environment.

4.3.2 Network Databases

Similarly Timor is not explicitly oriented to the network database model[74], but it has basic facilities which can be used to support such a database. In particular the explicit use of references for objects (see chapter 4 section 3; Appendix I) provides the necessary base for a network model.

4.4 Flexible Database Record Entries

One further advantage of Timor over other OO languages is that it offers a new construct, viz. *attribute types* (see chapter 9). This harmonises well with the modelling of a dynamically changing world, since it provide mechanisms which easily allow the objects in a database to change in the database as the real items (persons, cars, books, etc.) change or are updated in the real world. A person can become a student then later an office worker, can marry, etc. Similarly a car can be sold to a new owner and can have accidents and (which are attached as new accident records) etc. A library book can be moved from one library branch to another and/or can have a history of borrowers which can be crossed referenced to produce lists of readers, etc. The possibilities are endless.

4.5 Persistence

Because Timor (in conjunction with ModelOS) is designed as a persistent system it has the important advantages (a) that it simplifies programming, because it does not need recourse to a separate file system, and (b) for the same reason it is inherently more efficient.

4.6 Big Data

The features mentioned in this section suggest that a combination of ModelOS and Timor would be helpful in the world of big data[75]. We have already mentioned remote databases. Persistence means that algorithms execute directly on (big) data in the main memory. The separation of types and their implementations makes it relatively easy to design (and experiment with) implementations suited for big data, e.g. using gather-scatter techniques[76]) without modifying the applications which use the data. Parallelism is achieved within a node by the use

[73] see https://en.wikipedia.org/wiki/SQL
[74] see https://en.wikipedia.org/wiki/Network_model
[75] see https://en.wikipedia.org/wiki/Big_data
[76] see https://en.wikipedia.org/wiki/Gather-scatter_(vector_addressing)

of multiple threads[77], and between nodes by the Timor/ModelOS remote inter-module call facilities. Pragmas (see chapter 2 section 6) based on OpenMP[78] or using similar techniques can be added at the compiler level to support multi-platform shared-memory multiprocessing programming. Individual files in ModelOS can be up to 4 TB in length (see [6] volume 2). Since the translation from virtual address to page address can be individualised for each file the address translation can be optimised for large files (e.g. with pages of 128 MB or more), and a pre-paging strategy could be used to increase efficiency of the paging process.

Finally, all the extensive support mechanisms in ModelOS/Timor are available in the context of big data, as of course in all other ModelOS/Timor applications.

[77] At the hardware level the implementation of ModelOS in [6] describes a single processor system (to keep the description simple), but in practice tightly coupled multi-processor systems could also be designed and built.

[78] see https://en.wikipedia.org/wiki/OpenMP

APPENDIX I
A Timor Object Model

In this appendix a partial model is presented which hints at how a Timor compiler might organise values, objects and their references, and capabilities for files at run-time. It is not the intention to provide a detailed description, but merely to present a simple model. We begin by describing how *instance records* can be organised. It is assumed that the reader has familiarised himself with the various features of a Timor module, including the terminology used in the various chapters of the book.

1 Instance Records

An instance record is a run-time data structure which corresponds to an implementation of a structured type.

Because a structured type can have different implementations (which can co-exist in a single program) the space occupied by different instances of the same type can vary in size. For example consider some possible implementations of a type `Date`. An actual `Date` instance might be represented by three integer variables, or by a single string variable, or an array of string variables, etc. Consequently the compiler cannot pre-allocate the exact amount of space needed by an instance of a type. Since this issue should remain invisible to programmers, the run-time model for an instance of a structured type is that its content is represented by a fixed size internal pointer (not visible to programmers) which locates the actual instance record (that can vary in size from implementation to implementation). The instance record (see Figure AI.1) includes

- a methods pointer, which points to the implementation code of the type (i.e. a list of instance methods),
- an array of slots which describe its state. Each slot is a 64-bit word which holds a *pointer* to a value, an object, or a file capability (see chapter 4).

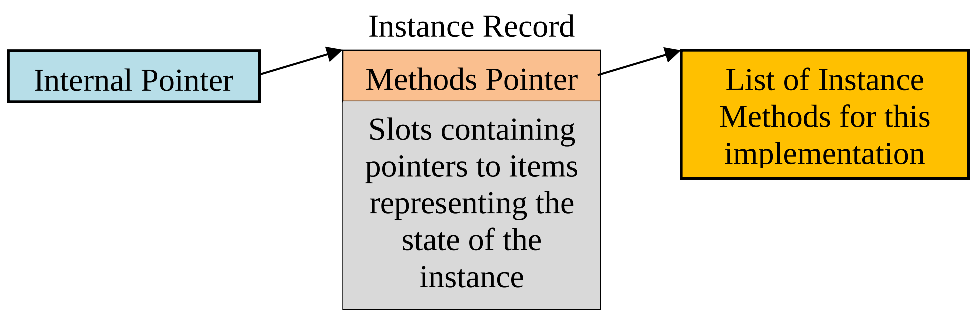
Instance Record
Internal Pointer
Methods Pointer
Slots containing pointers to items representing the state of the instance
List of Instance Methods for this implementation

Figure AI.1: An Instance Record for a Structured Type

The code of the instance methods is shared by all instances of the type which have the same implementation. Hence several instance methods in the same module may point to the same method list.

Since a structured type can be a component of some other structured type a slot in an instance record can point to a further instance record. For example a `Person` record might contain a `Date`, so that a slot in the `Person` instance record will point to a `Date` instance record. (To keep the diagrams simple this is not illustrated; also, in the diagrams all instance records look the same, but in fact the slots vary corresponding to the data declarations in the state data.)

The slots representing the state of the instance point to entries in a general heap for the module. The individual items in the heap may be values, pointers to an object or capabilities. (Optimisations are here possible, e.g. simple values may be directly held in the instance record, as is illustrated in Figure AI.2.)

When a type is instantiated, the *constructor* for the selected implementation of the type creates a new instance record and links this into a tree of instance records which together represent the state of the entire module. From the viewpoint of this model Timor *makers* are simply instance methods and have no special role in the object model. Since abstract variables are defined as a pair of methods, they too have no special role in the model.

2 The Object Table

An instance of a type can be declared as a *reference* for an object, using the single * notation. In this case the constructor creates an entry for an object in the module's object table and creates a reference to this in the appropriate instance record of the caller. Multiple references can point to the same object table entry, thus allowing the object to be shared, see Figure AI.2.

When assignment operators are applied to references, only the reference is copied. When the assignment operator is applied to a value (including a dereferenced object reference such as `*spouse`) a deep copy of the value is made. For a basic value this means that the value itself is copied to the slot indicated by the left hand side of the assignment expression. For an instance of a structured type it means that a new instance record is created and its internal pointer is assigned to the instance record slot indicated by the left hand side. For each internal pointer in the instance record being copied a copy of the corresponding instance record is made (recursively). The value of each reference and of each basic value in the instance record and in its nested instance records is copied into the new instance record.

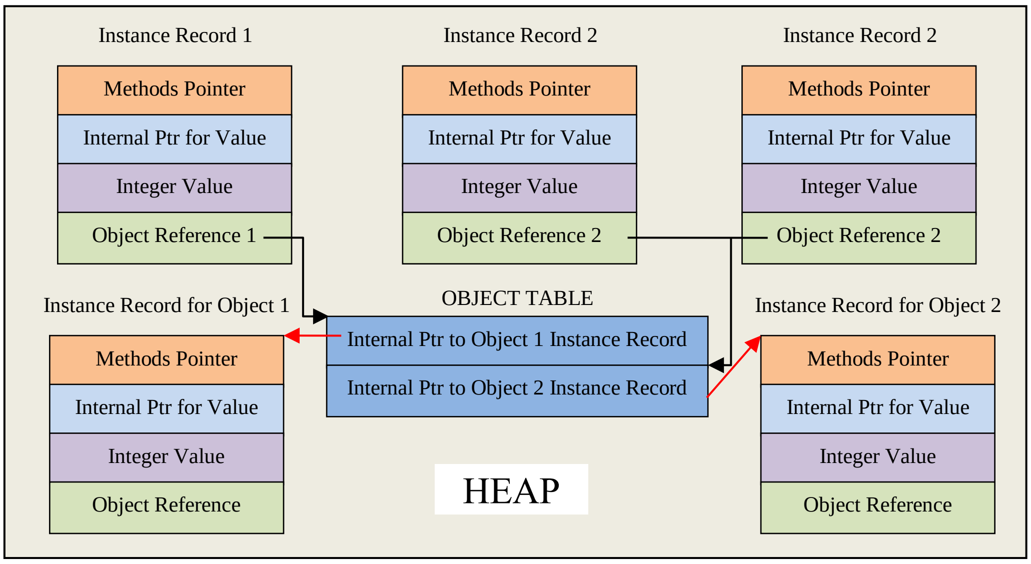

Figure AI.2 Sharing an Object

3 Capabilities

Capabilities in ModelOS are protected entities which provide the holder with access to other multiple entrypoint modules. Each capability has access rights associated with it. These determine inter alia the entry points of the target module which the caller can activate but also provide a large variety of protection measures relating to the module and/or the thread which holds the capability (see chapter 26 of [6]).

Capabilities are stored in instance records[79] (see Figure AI.3); they allow a thread to move between different modules. In conventional terms modules can be thought of as something akin to conventional files, although the underlying mechanisms are quite different. The modules which can be accessed via capabilities always hold code for the entrypoint routines (and where appropriate for internal routines) and they may also hold persistent data. Hence they can serve as multiple entrypoint programs, as files with semantic routines, as subroutine libraries, as directories, etc. The ModelOS environment eliminates the need for

[79] In ModelOS they are stored in a protected area of ModelOS segments, and are used in inter-module calls (see chapter 15 section 3.3).

special features, such as the special `public static void main (...)` mechanism for starting programs. Instead a "program" is started simply by using a capability to activate any permitted entrypoint routine of any module for which the caller has a capability. To understand this in more detail one should familiarise oneself with the ModelOS concepts described in [6].

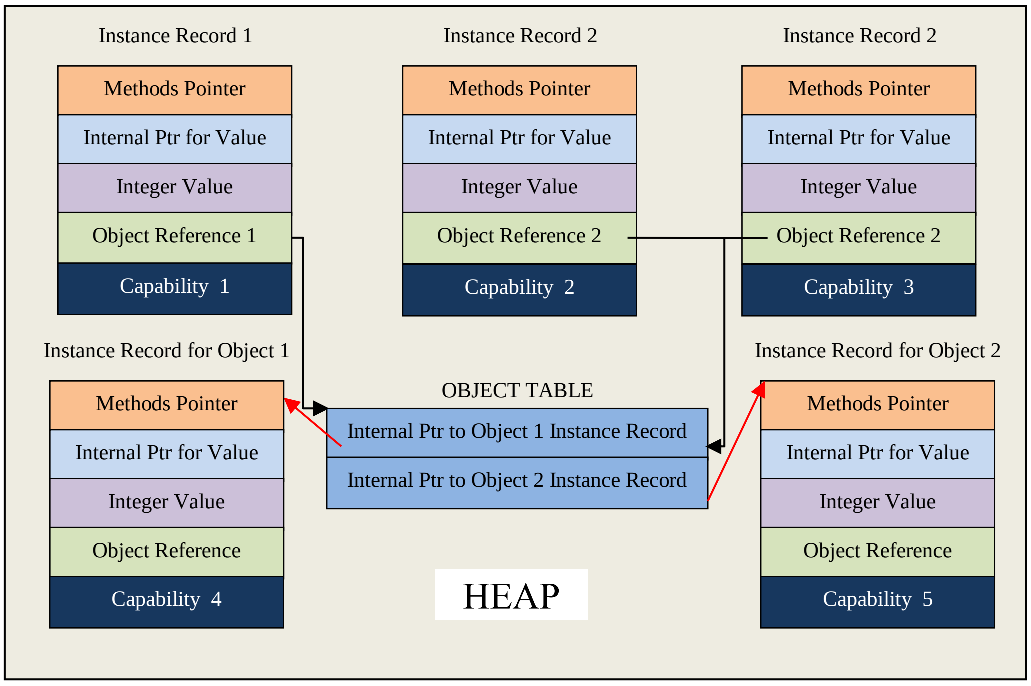

Figure I.3: Instance Records with Capabilities

APPENDIX II
The Timor Operators

This appendix summarises the operators in Timor.

1 The Null Value and Null Exceptions

There is a special value `null` which can be used in comparisons and assignment statements. This value represents an invalid value, reference or capability, depending on the context in which it is used. A null exception (called `NullEx`) is automatically raised by the Timor run-time system if a method is invoked on an invalid value, reference or capability.

2 Arithmetic Operators

2.1 Operations on Numerical Values

The arithmetic operators `+`, `-`, `*`, `/`, `%` are defined for numerical values as in Java.

In view of the difficulties which programmers might have in really understanding the *prefix* notations of the C++ and Java increment (++) and decrement (--) operators, these are not supported in Timor.

2.2 Collection Operators

These are described in detail in chapter 13 section 4.

2.3 Operations on Dynamic Attributes[80]

The `+` operator is used to attach an attribute object to an object of its base type, e.g.

[80] see Chapter 9.

```
Person* p = new Person::Impl();
Studying* s = new Studying Impl();
p = p + s;
```

If an actual attribute object is already attached to an object, the exception `AlreadyAttachedEx` is thrown. (However, attributes of the same type can be attaches, if they are not identical). If the attribute is not compatible with the base object, the exception `WrongTypeEx` is thrown.

The `-` operator is used to detach an attribute object from its current base object, e.g.

```
p = p - s;
```

If the attribute is not currently attached to the nominated base object the exception `NotAttachedEx` is thrown. The attribute remains reachable via its own reference(s).

There is a shorthand version of these operators, see section 3.4.

3 Assignment Operators

The assignment operator `=` can assign a value, a reference or a capability to a variable of the same type or of a supertype thereof (but not of an "including" type, i.e. not of a type defined using **`includes`**[81]). This is in principle a *copy* operation, i.e. the value, reference or capability is copied (but not the object or file to which a reference or capability refers).

Any value, reference or capability can be assigned to a "handle"[82]. The value `null` can be assigned to any variable.

3.1 Value Assignments

In the case of a value assignment, the new value is copied from the value on the right side of the assignment statement. It might have a different implementation from the value being replaced[83]. In this case, as in the case of an assignment of a subtype value, the amount of space required for the value may vary from that of the previous value; however, this is unproblematic, since the compiler always uses internal pointers[84].

If the instance record of a deleted or replaced value includes reference and/or capability slots the corresponding references and/or capabilities are delet-

81 see chapter 7.

82 see chapter 7 section 8.

83 see chapter 8.

84 See Appendix I. The heap entry associated with the internal pointer for a replaced value can be deleted when a new value (or null) is assigned to the slot. This becomes a recursive operation if the value includes pointers to further value slots.

ed, but the underlying objects or persistent files are not deleted.

Note: If the compiler uses reference counts to determine when a local object is deleted, the deletion of a reference embedded in a value variable must of course lead to the corresponding reference count being decremented. In the case of a `null` assignment to a capability the ModelOS operating system or emulator must be informed. If a capability has a special status (e.g. owner capability) the assignment may fail and the exception `InvalidAssignmentEx` is thrown to indicate that the assignment operation has failed.)

3.2 Reference and Capability Assignments

The new reference or capability is copied from the value on the right side of the assignment statement. (Note: If the compiler uses reference counts to determine when a local object is deleted, the deletion of a reference embedded in a value variable must of course lead to the corresponding reference count being decremented. In the case of a `null` assignment to a capability the ModelOS operating system must be informed.)

3.3 Assignments Involving Restrictions

If the assignment is being made from a restricted variable or parameter, the operation can only take place if the left side contains at least the same restrictions. Restrictors are described in chapter 15 section 2.

3.4 Shorthand Assignment Operators

The following operators are provided to allow programmers to shorten their programs. The left column shows an example of the operator's use, the second column provides a name and the third column explains the operator in terms of the normal assignment operator.

```
a += b   add assign          a = a + b
a -= b   subtract assign     a = a - b
a *= b   multiply assign     a = a * b
a /= b   divide assign       a = a / b
a %= b   modulus assign      a = a % b
```

The shorthand assignment operators "add assign" and "subtract assign" can also be used to shorten statements relating to dynamic attributes.

4 New, Create, Delete and Dereferencing Operators

The **`new`** operator accepts and copies an internal value and places it in the Object Table (see Appendix I). It returns a reference to the entry in the Object Table.

The **`create`** operator accepts and copies an internal value and passes it to the ModelOS system. This returns a capability for the new file.

The **`delete`** operator can be used only on internal *objects* and on *capabilities* for files. In the case of files the exception `NotAuthorisedEx` may be thrown if the ModelOS kernel indicates to the run-time system that the operation is not authorised.

The *dereferencing* operator accepts a reference or capability and returns a copy of the content of the object or persistent file as a *value*, e.g.

```
Date* d = new Date::Impl();
...
Date dob = *d;
```

Dereferencing can take place only if the reference or capability involved includes a `copy` access right (see chapter 15 section 2).

5 Comparison Operators

There are a number of standard operators which can in principle be used to compare instances of any Timor type and/or mode.

5.1 Comparing Values

These include the value comparison operators. They always compare values and provide a boolean result. They are:

```
== compare for equality
!= compare for inequality
<  compare for less than
<= compare for less than or equal
>  compare for greater than
>= compare for greater than or equal
```

For the primitive types (e.g. `Integer`, `Long`, `Boolean`, `Real`, `Float`, `Char`) the meaning of these operators is fully defined in Timor. (Note that `String` and `Array` are treated as examples of the type `List`, not as primitive types. The definitions for comparing lists apply to them.)

An equality/inequality comparison with `null` can be used to test whether a value is initialised. If two `null` values are compared using == the result is `true`.

The value comparison operators always compare values, regardless of the mode(s) of the operands. Thus it is possible in a single operation to compare say the value of a `Date` value with that of a reference to a `Date` object, e.g.

```
Date* d = new Date::Impl();
...
Date dob = "4/11/10";
if (d == dob) ...;
```

This rule allows instances of mixed modes, including instances addressable via type handles (e.g. `Date***`), to be compared without the application programmer having to cast them to the appropriate type, then dereference them before making a comparison. This is especially useful in co-types containing binary methods, but can of course be used in other contexts.

However, instances addressable via a `Handle` variable or parameter cannot be compared using the value operators. A compile time error occurs if an attempt is made to compare instances of different types (unless a subtyping relationship exists).

5.2 User-Defined Types: Comparing Using the Comparison Operators

For user-defined types the comparison operators have default definitions. However, the default definitions can be overridden in co-types (see section 9 of this appendix).

The default definitions compare all the values, references and capabilities defined in the instances to be compared. In cases where a type (or a component type) may have multiple implementations the comparison must be based on the results of the interface instance methods of that type. The comparison is a "shallow" comparison in the sense that the values of references and capabilities encountered are not checked. Instead the references and capabilities are themselves compared as pointers.

5.3 Comparing Subtypes via the Value Operators

If an attempt is made to compare a supertype instance with a subtype instance (by extension, *not* by inclusion) the comparison is made on the basis of the definition of the supertype. (The order of the operands is not significant.) Thus if an instance of type `Student` is compared with one of type `Person`, then the comparison is carried out solely on the basis of the `Person` details of both instances.

If an attempt is made to compare instances which are different subtypes of a common supertype (e.g. a `Student` and an `Employee`) then the comparison is made on the basis of the lowest common supertype (here `Person`). `Handle` is not considered a supertype in this sense, as it has no methods. A compile time error is raised if instances of incompatible types are compared.

5.4 Comparing References and Capabilities

The following boolean operators exist for comparing references and/or capabilities for *identity*:

```
~~ compare for identity
!~ compare for non-identity
```

A warning is raised at compile time if the compiler can establish that the comparison involves mixed modes. If an attempt is made to compare values using these operators, a compiler error occurs if this can be recognised at compile time.

However, no exceptions are automatically thrown at run time, even where a handle or a type handle `***` is involved. If a mixed mode comparison is attempted, the value returned by the operator `~~` is `false` (and the `!~` operator `true`). The `~~` operator returns `true` (and the `!~` operator false) if two `null` references or two invalid capabilities are compared.

6 Logical Operators

The logical operators `&`, `|`, `^`, `!` `&&`, `||` are defined as in Java. (Since `^` is only rarely used, we remind readers that this is the *exclusive or* operator, which returns `true` if one but not both of its operands evaluates to `true`.)

7 The Conditional Operator

The conditional or ternary operator `?:` found in Java is not supported in Timor, since this can easily lead to unclear programs.

8 Bit Manipulation Operators

To support system programming Timor provides the same bitwise operators as are found in Java. These can only be used on integer values.

9 Defining/Redefining Operators in Co-Types

Some operators can be redefined by an application in co-types. The scope of such a definition is the persistent module in which the co-type resides.

9.1 Associating Binary Comparison Methods with Operators

If a co-type contains a binary method with the identifier `equal` (with exactly two parameters of the type being expanded by the co-type and returning a boolean result) the compiler implements the operator == by calling this method.

If a co-type contains a binary method with the identifier `less` (with exactly two parameters of the type being expanded by the co-type and returning a boolean result) the compiler implements the operator < by calling this method.

If both methods are present in the same co-type, the following further operators can be automatically implemented using them in combination:

```
<= less or equal
>  not (less or equal)
>= not less
```

If the co-type also defines methods with identifiers which might appear to be directly associated with these (e.g. using names such as `lessequal`, `greater` and/or `greaterequal`), they are of no special significance to the compiler.

9.2 Associating Binary Operations with Operators

If a co-type contains a binary method with the identifier `plus`, `minus`, `multiply`, `divide`, `remainder` with exactly two input parameters of the same type and a return value (also of the same type) the compiler implements the operators `+`, `-`, `*`, `/`, `%` by calling the corresponding method.

The compiler is not concerned with relationships between any of these methods.

9.3 Definitions in Multiple Co-Types

Where a program contains multiple co-types for a type and these contain appropriate methods for use as operators, then the normal rules for selecting a co-type are used. However, all the methods which the compiler uses for implementing comparison operators must be taken from a single co-type, i.e. it is not possible to take `equal` from one co-type and `less` from a different co-type. Similarly all the methods which the compiler uses for implementing binary operation operators must be taken from a single co-type (which need not be that used for the comparison operators).

Appendix III: EBNF for the Timor Programming Language

M. Evered, June 2021

// Program structure syntax

```
compilation_unit = compilation_item { compilation_item }
compilation_item = [ "template" [ "<" big_id { "," big_id } ">" ] [ "func" "<" generic_func
        { ";" generic_func } ">" ] ] { unit_qualifier } ( type_defn | implementation_defn )
type_defn = enum_type | object_type
enum_type = enum_kind big_id "{" small_id { "," small_id } "}"
enum_kind = "enum" | "seq" | "circ"
object_type =  object_type_head [ "expands" big_id ] "{" { derivation_section } { type_section } "}"
object_type_head =  ( "type" big_id [ "&" small_id ] [ "for" attach_type ] ) | ( "view" big_id )
unit_qualifier = "abstract" | "singleton" | "library" | "comod" | "callback"
attach_type = big_id | "any"
derivation_section = ( "extends" | "includes" | "adjusts" ) ":" { inherited_item }
inherited_item = [ qualifying_list ] ( single_item | ( "(" { inherited_item } ")" ) ) ";"
qualifying_list = "{" { inherited_item } "}"
single_item = big_id [ "&" small_id ] [ "<" big_id { "," big_id } ">" ] [ small_id ] { "," small_id }
type_section = redefines_section | type_instance_section | type_protected_section
        | type_callback_section | type_qualifies_section | type_callout_section | type_maker_section
        | type_binary_section | type_inout_section
redefines_section = "redefines" ":" { redefines_item ";" }
redefines_item = [ "[" small_id { "," small_id } "]" ] ( big_id | method_signature )
type_instance_section = [ "predefines" ] "instance" ":" { method_signature ";" }
type_protected_section = "protected" ":" { method_signature ";" }
type_callback_section = "callback" ":" { method_signature ";" }
type_qualifies_section = "qualifies" ( big_id | "any" ) ":" { method_signature ";" }
type_callout_section = "callout" ( big_id | "any" ) ":" { method_signature ";" }
type_maker_section = [ "predefines" ] "maker" ":" { method_signature ";" }
type_binary_section = [ "predefines" ] "binary" ":" { method_signature ";" }
type_inout_section = [ "predefines" ] "inout" ":" { method_signature ";" }
method_signature = [ method_qualifier ] ( abstract_var | op_signature )
method_qualifier = "final"
abstract_var = type_spec small_id { "," small_id }
op_signature = op_kind ( type_spec | "bracket" ) ( small_id | op_kind | "all" ) "(" [ par_list ] ") [ throws_clause ]
op_kind = "op" | "enq" | "open" | "close"
```

```
throws_clause = "throws" big_id { "," big_id }
par_list = "..." | ( par_sublist { ";" par_sublist } )
par_sublist = type_spec par_id { "," par_id }
par_id = small_id [ "=" expression ]
identifier = small_id | big_id
implementation_defn = "impl" [ big_id [ "&" small_id ] ] "::" identifier [ "expands" big_id ]
        "{" { impl_section }"}"
impl_section = state_section | retained_section | constr_section | instance_section | protected_section
        | callback_section | internal_section | with_section | qualifies_section | callout_section | maker_section
        | binary_section | inout_section
state_section = "state" ":" { ( var_declaration | reuse_declaration ) }
reuse_declaration = ( "^" | "^^" ) ( type_spec [ "::" identifier ] [ var_id ] | "::" identifier ) ";"
retained_section = "retained" ":" { var_declaration }
constr_section = "constr" ":" constructor
instance_section = [ "predefines" ] "instance" ":" { method }
protected_section = "protected" ":" { method }
callback_section = "callback" ":" { method }
internal_section = "internal" ":" { method }
qualifies_section = "qualifies" ( big_id | "any" ) ":" { method }
callout_section = "callout" ( big_id | "any" ) ":" { method }
maker_section = [ "predefines" ] "maker" ":" { method }
binary_section = [ "predefines" ] "binary" ":" { method }
inout_section = [ "predefines" ] "inout" ":" { method }
with_section = "with" "(" small_name ")" [ "as" small_id ] "{" { impl_section } "}"
constructor = constr_signature block
constr_signature = [ big_id ] "::" identifier [ "<" generic_func { ";" generic_func } ">" ]
        "(" par_list ")" [ throws_clause ]
generic_func = type_spec big_id "(" [ par_list ] ")"
method = [ method_qualifier ] op_signature block
block =  "{" { block_statement } "}"

// Variable declaration syntax
var_declaration = [ "final" | "const" | "fixed" ] type_spec var_id { "," var_id } ";"
var_id = small_id [ "=" expression ]
type_spec = ( "void" | name ) [ "&" small_id ] [ "<" type_spec { "," type_spec } ">" ]
        [ "[:" restrictor_expr { "," restrictor_expr } ":]" ] { "[" "]" } [ mode_modifier ]
restrictor_expr = restrictor_id { ( "+" | "-" | "*" ) restrictor_id }
restrictor_id = identifier | "all" | "op" | "enq" | "body" | "call"
mode_modifier = "*" | "**" | "***"
name = big_id { "." big_id }
small_name = small_id { "." identifier }
```

```
// Statement syntax
statement = while_statement | repeat_statement | for_statement | if_statement | case_statement
        | with_statement | try_statement | block | ( expression ";" ) | ( throw_statement ";" )
        | ( return_statement ";" ) | ( delete_statement ";" ) | cast_statement | cocast_statement
while_statement = "while" "(" expression ")" statement
repeat_statement = "repeat" statement { "until" "(" expression ")" ]
for_statement = "for" "(" [ type_spec ] small_id "in" expression ")" statement [ "else" statement ]
if_statement = "if" "(" expression ")" statement { "elsif" "(" expression ")" statement } [ "else" statement ]
case_statement = "case" "(" expression ")" "of" "{" { "(" expression ")" block } [ "else" statement ] "}"
with_statement = "with" "(" small_name ")" [ "as" small_id ] block
try_statement = "try" statement { "catch" "(" exception_list small_id ")" statement } [ "finally" statement ]
block_statement = ( var_declaration ";" ) | statement
exception_list = "big_id" { "|" big_id }
throw_statement = "throw" expression
return_statement = "return" expression
delete_statement = "delete" "(" expression ")"
cast_statement = "cast" "(" expression ")" "as" "{" { cast_selector block } [ "else" statement ] "}"
cast_selector = "(" type_spec small_id ")" | "[" type_spec small_id "]"
cocast_statement = "cocast" "(" type_spec small_id "," small_id "&" small_id ")" "as"
        "{" { cast_selector block } [ "else" statement ] "}"

// Expression syntax
expression = conditional_expression [ assignment_operator expression ]
assignment_operator = "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | ">>>=" | "&=" | "^=" | "|=" | "=*"
conditional_expression = conditional_and_expression { "||" conditional_and_expression }
conditional_and_expression = inclusive_or_expression { "&&" inclusive_or_expression }
inclusive_or_expression = exclusive_or_expression> { "|" exclusive_or_expression }
exclusive_or_expression = and_expression { "^" and_expression }
and_expression = equality_expression { "&" equality_expression }
equality_expression = instanceof_expression { ( "==" | "!=" | "~~" | "!~" ) instanceof_expression }
instanceof_expression = relational_expression [ "instanceof" big_id ]
relational_expression = shift_expression { ( "<" | ">" | "<=" | ">=" | "in" ) shift_expression }
shift_expression = additive_expression { ( "<<" | ">>" | ">>>" ) additive_expression }
additive_expression = multiplicative_expression { ( "+" | "-" ) multiplicative_expression }
multiplicative_expression = unary_expression { ( "*" | "/" | "%" ) unary_expression }
unary_expression = preincrement_expression | predecrement_expression | ("+" <unary_expression>)
        | ("-" unary_expression) | unary_expression_not_plus_minus | new_expression
        | create_expression | dereference_expression
predecrement_expression = "--" unary_expression
preincrement_expression = "++" unary_expression
new_expression = "new" [ primary_expression ] primary_expression
create_expression = "create" [ primary_expression ] primary_expression
```

```
dereference_expression = "*" unary_expression
unary_expression_not_plus_minus = postfix_expression | ("~" unary_expression) | ("!" unary_expression)
        | cast_expression
postfix_expression = primary_expression [ ( "++" | "--" ) ]
cast_expression = "(" ( type_spec | "reference" | "capability" ) ")" unary_expression
primary_expression = primary_prefix { primary_suffix }
primary_prefix = literal | "this" | ( "(" expression ")" ) | impl_allocation | allocation_expression
        | ( small_name [ "&" small_id ] ) | collection | ("callback" "." identifier)
        | "base" | "body" | "call" | big_id
impl_allocation = name [ "&" small_id ] [ "<" type_spec { "," type_spec } ">" ] [ "[" "]" ] "::" identifier
        [ generic_func_par_list ] arguments
generic_func_par_list = "<" generic_func_par { "," generic_func_par } ">"
generic_func_par = "(" expression ")"
allocation_expression = "new" name arguments
collection = "{" [ type_spec ":" ] [ collection_range ] { "," collection_range } "}"
collection_range = expression [ ".." expression ]
primary_suffix = ("[" expression "]") | ("." identifier) | arguments
         "{" type_spec small_id ":" expression "}" | "{" expression ".." expression "}"
        | "[" type_spec small_id ":" expression "]"
literal = integer_literal | floating_point_literal | character_literal | string_literal | boolean_literal | "null"
boolean_literal = "true" | "false"
arguments = "(" [ argument_list ] ")"
argument_list = "..." | ( argument { "," argument } )
argument = "*" | expression
```

References

[1] M. Evered, LEIBNIZ - A Language to Support Software Engineering, Darmstadt: Dr.Ing thesis, Technical University of Darmstadt, Faculty of Computer Science, 1985.

[2] G. Menger, Unterstützung für Objektsammlungen in statisch getypten objektorientierten Programmiersprachen, Dr. rer.nat thesis, University of Ulm Germany, 2000.

[3] A. Schmolitzky, Ein Modell zur Trennung von Vererbung und Typabstraktion in objektorientierten Sprachen, Dr. rer.nat, thesis, University of Ulm, Germany, 1999.

[4] K. Espenlaub, Design of the SPEEDOS Operating System Kernel, Ulm, Germany: Ph.D. thesis, The University of Ulm, Department of Computer Structures, Computer Science Faculty, 2005.

[5] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM,* vol. 15, no. 12, pp. 1053-1058, 1972.

[6] J. L. Keedy, Making Operating Systems Secure, 2021.

[7] M. D. McIlroy, "Mass Produced Software Components," in *Software Engineering: Concepts and Techniques, Petrocelli-Charter*, New York, 1968.

[8] J. L. Keedy, G. Menger and C. Heinlein, "Inheriting from a Common Abstract Ancestor in Timor," *Journal of Object Technology,* vol. 1, no. 1, pp. 81-106, May-June 2002.

[9] J. L. Keedy, G. Menger and C. Heinlein, "Support for Subtyping and Code Re-use in Timor," in *40th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, 2002.

[10] J. L. Keedy, C. Heinlein, G. Menger and M. Evered, "Diamond Inheritance and Attribute Types in Timor," *Journal of Object Technology,* vol. 3, no. 10, pp. 121-142, Nov-Dec 2004.

[11] J. L. Keedy, G. Menger and C. Heinlein, "Inheriting Multiple and Repeated Parts in TIMOR," *Journal of Object Technology,* vol. 3, no. 10, pp. 99-120, Nov-Dec 2004.

[12] J. L. Keedy, C. Heinlein and G. Menger, "Reuse Variables: Reusing Code and State in Timor," in *8th International Conference on Software Reuse, ICSR 2004, Lecture Notes in Computer Science 3107*, Madrid, 2004.

[13] J. L. Keedy, K. Espenlaub, G. Menger and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," *Journal of Object Technology,* vol. 3, no. 1, pp. 101-121, 2004.

[14] J. L. Keedy, G. Menger, C. Heinlein and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Net.ObjectDays, 2002*, Erfurt, Germany, 2003.

[15] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, "Persistent Objects and Capabilities in Timor," *Journal of Object Technology,* vol. 6, no. 4, pp. 103-123, May-June 2007.

[16] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, "Persistent Processes and Distribution in Timor," *Journal of Object Technology,* vol. 6, no. 6, pp. 91-108, 2007.

[17] J. L. Keedy, G. Menger and C. Heinlein, "Types and Co-Types in Timor," *Journal of Object Technology,* vol. 8, no. 7, 2009.

[18] J. Keedy, G. Menger and C. Heinlein, "Covariantly Adjusting Co-Types in Timor," vol. 9, no. 1, pp. 35-55, 2010.

[19] F. DeRemer and H. Kron, "Programming-in-the large versus programming-in-the-small," in *Proceedings of the International Conference on Reliable Software*, Los Angeles, 1975.

[20] K. Arnold, J. Gosling and D. Holmes, The Java Programming Language, 3rd ed., Addison-Wesley, 2000.

[21] J. L. Keedy, M. Evered, A. Schmolitzky and G. Menger, "Attribute Types and Bracket Implementations," in *25th International Conference on Technology of Object Oriented Systems, TOOLS 25*, Melbourne, 1997.

[22] J. L. Keedy, K. Espenlaub, G. Menger and C. Heinlein, "Call-out Bracket Methods in Timor," *Journal of Object Technology,* vol. 5, no. 1, pp. 51-67, 2006.

[23] J. L. Keedy, K. Espenlaub, G. Menger, C. Heinlein and M. Evered, "Statically Qualified Types in Timor," *Journal of Object Technology,* vol. 4, no. 7, 2005.

[24] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, "Security and Protection in Timor Programs," *Journal of Object Technology,* vol. 7, no. 4, pp. 123-138, 2008.

[25] J. L. Keedy, K. Espenlaub, C. Heinlein, G. Menger, F. Henskens and M. Hannaford, “Support for Object Oriented Transactions in Timor,” *Journal of Object Technology,* vol. 5, no. 2, pp. 103-124, 2006.

[26] K. B. Bruce, L. Cardelli, G. Castagna, G. Leavens and B. Pierce, “On Binary Methods,” *Theory and Practice of Object Systems,* vol. 1, pp. 221-242, 1995.

[27] G. Bracha, “Generics in the Java Programming Language,” *PDF version: java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, http://java.sun.com/docs/books/tutorial/extra/generics/index.html, 2004.*

[28] M. Evered, J. L. Keedy, G. Menger and A. Schmolitzky, “Genja - A New Proposal for Genericity in Java,” in *25th International Conf. on Technology of Object-Oriented Languages and Systems, 25*, Melbourne, 1997.

[29] M. Evered, J. L. Keedy, G. Menger and A. Schmolitzky, “Toward Zero Overhead Genericity in Java,” in *Fachkongreß, Smalltalk und Java in Industrie und Ausbildung*, Erfurt, 1997.

[30] E. W. Dijkstra, “Cooperating Sequential Processes,” in *Programming Languages*, E. Genuys, Ed., Academic Press, 1968, pp. 43-112.

[31] J. L. Keedy, K. Ramamohanarao and J. Rosenberg, “On Implementing Semaphores with Sets,” *The Computer Journal,* vol. 22, no. 2, pp. 146-150, 1979.

[32] P. Courtois, F. Heymans and D. L. Parnas, “Concurrent control with readers and writers,” *Comm.ACM,* vol. 14, no. 10, pp. 667-668, 1971.

[33] J. L. Keedy, J. Rosenberg and K. Ramamohanarao, “On Synchronising Readers and Writers with Semaphores,” *The Computer Journal,* vol. 25, no. 1, pp. 121-125, 1982.

[34] E. E. M. Torgersen, C. Hansen, P. von der Ahe, G. Bracha and N. Gafter, “Adding Wildcards to the Java Programming Language,” *Journal of Object Technology, 3 (2004),* vol. 3, pp. 97-116, 2004.

[35] M. Atkinson, P. Bailey, K. Chisholm, W. Cockshott and R. Morrison, “An Approach to Persistent Programming,” *The Computer Journal,* vol. 26, no. 4, pp. 360-365, 1983.

[36] E. Codd, “A Relational Model of Data for Large Shared Data Banks,” *Communications of the ACM,* vol. 13, no. 6, pp. 377-387, June 1970.

[37] R. C. Daley and J. B. Dennis, “Virtual Memory, Processes, and Sharing in MULTICS,” *Communications of the ACM,* vol. 11, no. 5, pp. 306-312, 1968.

[38] A. Bensoussan, C. T. Clingen and R. C. Daley, “The MULTICS Virtual Memory: Concepts and Design,” *Communications of the ACM,* vol. 15, no. 5, pp. 308-318, May 1972.

[39] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott and R. Morrison, “PS-Algol: A Language for Persistent Programming,” in *Proceedings of the 10th Australian National Computer Conference*, Melbourne, Australia, 1983.

[40] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle and M. P. Atkinson, “The Napier Type System,” in *Proceedings of the 3rd International Workshop on Persistent Object Systems*, 1989.

Bibliography

Arnold, K., Gosling, J. & Holmes, D. (2000). The Java Programming Language (3rd ed.). Addison-Wesley.

Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. & Morrison, R. (1983). PS-Algol: A Language for Persistent Programming. Proceedings of the 10th Australian National Computer Conference, (pp. 70-79). Melbourne, Australia.

Atkinson, M., Bailey, P., Chisholm, K., Cockshott, W. & Morrison, R. (1983). An Approach to Persistent Programming. The Computer Journal, 26(4), 360-365.

Bensoussan, A., Clingen, C. T. & Daley, R. C. (1972, May). The MULTICS Virtual Memory: Concepts and Design. Communications of the ACM, 15(5), 308-318.

Bracha, G. (n.d.). Generics in the Java Programming Language. PDF version: java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, http://java.sun.com/docs/books/tutorial/extra/generics/index.html, 2004.

Bruce, K. B., Cardelli, L., Castagna, G., Leavens, G. & Pierce, B. (1995). On Binary Methods. Theory and Practice of Object Systems, 1, 221-242.

Codd, E. (1970, June). A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, 13(6), 377-387.

Courtois, P., Heymans, F. & Parnas, D. L. (1971). Concurrent control with readers and writers. Comm.ACM, 14(10), 667-668.

Daley, R. C. & Dennis, J. B. (1968). Virtual Memory, Processes, and Sharing in MULTICS. Communications of the ACM, 11(5), 306-312.

DeRemer, F. & Kron, H. (1975). Programming-in-the large versus programming-in-the-small. Proceedings of the International Conference on Reliable Software (pp. 114-121). Los Angeles: Association for Computing Machinery.

Dijkstra, E. W. (1968). Cooperating Sequential Processes. In E. Genuys (Ed.), Programming Languages (pp. 43-112). Academic Press.

Espenlaub, K. (2005). Design of the SPEEDOS Operating System Kernel. Ulm, Germany: Ph.D. thesis, The University of Ulm, Department of Computer Structures, Computer Science Faculty.

Evered, M. (1985). LEIBNIZ - A Language to Support Software Engineering.

Darmstadt: Dr.Ing thesis, Technical University of Darmstadt, Faculty of Computer Science.

Evered, M., Keedy, J. L., Menger, G. & Schmolitzky, A. (1997). Genja - A New Proposal for Genericity in Java. 25th International Conf. on Technology of Object-Oriented Languages and Systems, 25, (pp. 181-189). Melbourne.

Evered, M., Keedy, J. L., Menger, G. & Schmolitzky, A. (1997). Toward Zero Overhead Genericity in Java. Fachkongreß, Smalltalk und Java in Industrie und Ausbildung, (pp. 62-67). Erfurt.

Keedy, J. L. (2021). Making Operating Systems Secure.

Keedy, J. L., Espenlaub, K., Heinlein, C. & Menger, G. (2007). Persistent Processes and Distribution in Timor. Journal of Object Technology, 6(6), 91-108.

Keedy, J. L., Espenlaub, K., Heinlein, C. & Menger, G. (2007, May-June). Persistent Objects and Capabilities in Timor. Journal of Object Technology, 6(4), 103-123.

Keedy, J. L., Espenlaub, K., Heinlein, C. & Menger, G. (2008). Security and Protection in Timor Programs. Journal of Object Technology, 7(4), 123-138.

Keedy, J. L., Espenlaub, K., Heinlein, C., Menger, G., Henskens, F. & Hannaford, M. (2006). Support for Object Oriented Transactions in Timor. Journal of Object Technology, 5(2), 103-124.

Keedy, J. L., Espenlaub, K., Menger, G. & Heinlein, C. (2004). Qualifying Types with Bracket Methods in Timor. Journal of Object Technology, 3(1), 101-121.

Keedy, J. L., Espenlaub, K., Menger, G. & Heinlein, C. (2006). Call-out Bracket Methods in Timor. Journal of Object Technology, 5(1), 51-67.

Keedy, J. L., Espenlaub, K., Menger, G., Heinlein, C. & Evered, M. (2005). Statically Qualified Types in Timor. Journal of Object Technology, 4(7).

Keedy, J. L., Evered, M., Schmolitzky, A. & Menger, G. (1997). Attribute Types and Bracket Implementations. In C. D. Mingins (Ed.), 25th International Conference on Technology of Object Oriented Systems, TOOLS 25, (pp. 325-337). Melbourne.

Keedy, J. L., Heinlein, C. & Menger, G. (2004). Reuse Variables: Reusing Code and State in Timor. 8th International Conference on Software Reuse, ICSR 2004, Lecture Notes in Computer Science 3107, (pp. pp. 205-214). Madrid.

Keedy, J. L., Heinlein, C., Menger, G. & Evered, M. (2004, Nov-Dec). Diamond Inheritance and Attribute Types in Timor. Journal of Object Technology, 3(10), 121-142.

Keedy, J. L., Menger, G. & Heinlein, C. (2002, May-June). Inheriting from a Common Abstract Ancestor in Timor. Journal of Object Technology, 1(1), 81-106.

Keedy, J. L., Menger, G. & Heinlein, C. (2002). Support for Subtyping and Code Re-use in Timor. In J. Potter & J. Noble (Ed.), 40th International Conference on Technology of Object-Oriented Languages and Systems (pp. 35-43). Sydney: Australian Computer Society, Inc.

Keedy, J. L., Menger, G. & Heinlein, C. (2004, Nov-Dec). Inheriting Multiple and Repeated Parts in TIMOR. Journal of Object Technology, 3(10), 99-120.

Keedy, J. L., Menger, G. & Heinlein, C. (2009). Types and Co-Types in Timor. Journal of Object Technology, 8(7).

Keedy, J. L., Menger, G., Heinlein, C. & Henskens, F. (2003). Qualifying Types Illustrated by Synchronisation Examples. In M. M. Aksit (Ed.), Net.ObjectDays, 2002. LNCS 2591, pp. 330-344. Erfurt, Germany: Springer.

Keedy, J. L., Ramamohanarao, K. & Rosenberg, J. (1979). On Implementing Semaphores with Sets. The Computer Journal, 22(2), 146-150.

Keedy, J. L., Rosenberg, J. & Ramamohanarao, K. (1982). On Synchronising Readers and Writers with Semaphores. The Computer Journal, 25(1), 121-125.

Keedy, J., Menger, G. & Heinlein, C. (2010). Covariantly Adjusting Co-Types in Timor. 9(1), 35-55.

McIlroy, M. D. (1968). Mass Produced Software Components. In P. R. Naur (Ed.), Software Engineering: Concepts and Techniques, Petrocelli-Charter. New York: Petrocelli-Charter.

Menger, G. (2000). Unterstützung für Objektsammlungen in statisch getypten objektorientierten Programmiersprachen. Dr. rer.nat thesis, University of Ulm Germany.

Morrison, R., Brown, A., Carrick, R., Connor, R., Dearle, A. & Atkinson, M. P. (1989). The Napier Type System. Proceedings of the 3rd International Workshop on Persistent Object Systems (pp. 3-18). Springer-Verlag,.

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12), 1053-1058.

Schmolitzky, A. (1999). Ein Modell zur Trennung von Vererbung und Typabstraktion in objektorientierten Sprachen. Dr. rer.nat, thesis, University of

Ulm, Germany.

Torgersen, E. E., Hansen, C., von der Ahe, P., Bracha, G. & Gafter, N. (2004). Adding Wildcards to the Java Programming Language. Journal of Object Technology, 3 (2004), 3, 97-116.

Acknowledgements

My very special thanks are due to Mark Evered, who prepared the EBNF in Appendix III of this book. His help was very much appreciated.

I also draw special attention to David Abramson for recently contributing some relevant ideas to this book. His advice was very much appreciated.

Design of Timor

I would also like to thank all my PhD students who have contributed to the design of Timor. Key contributions, initially carried out in Ulm, were made by

– Mark Evered, later Senior Lecturer at the University of New England, NSW Australia and Researcher at the Department of Primary Industries NSW.

– Axel Schmolitzky, later Professor at the University of Applied Sciences, Hamburg, Germany.

– Gisela Menger, now retired.

– Christian Heinlein, later Professor at the University of Applied Sciences, Aalen, Germany and Dean of Studies.

The design was continued after my retirement with the help of Gisela Menger, who graciously offered several years of her retirement to assisting me.

MONADS Design and Implementations

I always saw programming language design as an important contribution to my operating system work on the MONADS and SPEEDOS projects, but not infrequently my assistants on those projects also made contributions the programming language work, for which I am also thankful. In this context I especially mention my following former PhD students:

– John Rosenberg, who later became Professor at the University of Sydney, Dean of the Information Technology Faculty at Monash University, Deputy Vice-Chancellor at the Universities of Deakin and then Latrobe.

– David Abramson, later Professor and Head of Department at Monash University, then Director of Research at the Research Computer Centre of the University of Queensland (Co-supervisor Professor Chris Wallace).

– Kotagiri Ramamohanarao, later Professor of Computer Science at the University of Melbourne; Head of Computer Science and Software Engineering, Head of the School of Electrical Engineering and Computer Science at the University of Melbourne and Research Director for the Cooperative Research Centre for Intelligent Decision Systems.

– Frans Henskens, later Associate Professor at the University of Newcastle,

NSW; Head of the Discipline of Computer Science and Software Engineering, Deputy Head of School of Electrical Engineering and Computer Science, Assistant Dean (IT) in the Faculty of Engineering and Built Environment and subsequently Professor in the Faculty of Health and Medicine at the University of Newcastle (Supervisor Prof. John Rosenberg).

Further Work on Operating Systems Design

During my period as Professor of Operating Systems at the University of Darmstadt in Germany some advanced synchronisation techniques which are reflected in Timor were made by my PhD student

– Bernd Freisleben, later Professor of Distributed Systems at the University of Marburg in Germany.

At the University of Bremen in Germany the following contributed further ideas to the design of operating systems and database systems:

– Karin Vosseberg, later Professor of Software Technology at the University of Applied Sciences, Bremerhaven in Germany and Deputy Director for Study and Teaching.

– Peter Brössler, later a manager in various companies and then a Freelance Management Adviser in Munich in Germany.

Engineering Support for the MONADS Systems

Special mention is due on the engineering side to David Koch at Monash University and the University of Newcastle, and to Jörg Siedenburg at the Universities of Bremen and Ulm.

Design of SPEEDOS

Many important contributions to SPEEDOS were made by my PhD student

– Klaus Espenlaub, now Software Development Director, Oracle VM VirtualBox, Oracle Corporation.

Finally I would also like to thank all the undergraduate students who worked on MONADS, SPEEDOS and/or Timor.

My work has been supported over the years by several competent secretaries, and my special thanks in this respect are due to Renate Post-Gonzales for organising the 'International Workshop on Computer Architectures to Support Security and Persistence' in Bremen in 1990.

Thanks are also due to the Australian Research Grants Committee for their financial support of the MONADS Project at Monash and Newcastle.

Above all I am enormously grateful for the love, patience and support which I have received from my wife Ulla and also from my son Nicolas