The United Nations
University

UNU/IIST

International Institute for
Software Technology

Introduction to RAISE

Chris George

March 2002

UNU/IIST Report No. 249

UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2003

The United Nations
University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

Introduction to RAISE

Chris George

Abstract

This report provides a brief, technical introduction to RAISE. It first provides an introduction to the RAISE Specification Language (RSL). Then there are two sections on the method, covering both the writing of in initial specifications and their development into executable software.

Chris George is a Senior Research Fellow at UNU/IIST, 1 September 1994 - 31 August 2003. He is one of the main contributors to RAISE, particularly the RAISE method, and that remains his main research interest. Before coming to UNU/IIST he worked for companies in the UK and Denmark.

Copyright © 2002 by UNU/IIST, Chris George

Contents

1 Introduction **1**

2 The RAISE Specification Language **1**
2.1 Basic Class Expressions 1
2.2 Types 2
2.2.1 Built-in types 2
2.2.2 Type Constructors 3
2.2.3 Type Expressions 6
2.2.4 Type Definitions 7
2.2.5 Abstract types 7
2.2.6 Records 8
2.2.7 Variants 8
2.2.8 Unions 9
2.3 Values 10
2.3.1 Overloading and Distinguishable Types 12
2.4 Logic 12
2.4.1 Quantifiers 15
2.4.2 Typings 15
2.4.3 Bindings 15
2.5 Value Expressions 16
2.5.1 Set Expressions 16
2.5.2 List Expressions 17
2.5.3 Map Expressions 17
2.5.4 Let Expressions 17
2.5.5 Case Expressions 18
2.6 Axioms 19
2.7 Test Cases 19
2.8 Imperative Constructs 20
2.8.1 If Expressions 21
2.8.2 Iterative Expressions 22
2.8.3 Local Expressions 23
2.9 Concurrency 23
2.9.1 Comprehended Expressions 27
2.10 Modules 28
2.10.1 Basic Class Expressions 28
2.10.2 Extending Class Expressions 28
2.10.3 Renaming Class Expressions 28
2.10.4 Hiding Class Expressions 28
2.10.5 With Class Expressions 29
2.10.6 Scheme Instantiations 29
2.11 Parameterised Schemes 29
2.12 Object Declarations 31
2.13 Comments 32

3 The RAISE Method: Writing Initial Specifications **32**
3.0.1 Be Abstract 33
3.0.2 Use Users' Concepts 33
3.0.3 Make it Readable 33
3.0.4 Look for Problems 34

3.0.5 Minimise the State . . . 34
3.0.6 Identify Consistency Conditions . . . 34
3.1 Kinds of Module . . . 36
3.1.1 Global Objects . . . 36
3.1.2 State Components . . . 36
3.2 Abstract and Concrete Modules . . . 36
3.3 Module Hierarchies . . . 40
3.4 Sharing Child Modules . . . 41

4 The RAISE Method: Developing Specifications **43**
4.1 Imperative Modules . . . 43
4.2 Concurrent Modules . . . 47
4.3 Development Route . . . 49
4.4 Asynchronous Systems . . . 49
4.5 Validation and Verification . . . 50
4.6 Refinement . . . 51
4.7 Lightweight Formal Methods . . . 52
4.8 Confidence Conditions . . . 53
4.8.1 Theorems . . . 55
4.9 Generating the Executable Program . . . 56

5 When Not to Use RAISE **56**
5.1 There is a Special-Purpose Formalism . . . 57
5.2 The Effort is Not Worth the Gain . . . 57

References **58**

A RSL Syntax **60**
A.1 Conventions . . . 60
A.2 Changes from RSL Book . . . 60
A.3 Modules and Declarations . . . 60
A.4 Class Expressions . . . 61
A.5 Object Expressions . . . 61
A.6 Type Expressions . . . 62
A.7 Value Expressions . . . 62
A.8 Bindings . . . 64
A.9 Typings . . . 64
A.10 Patterns . . . 64
A.11 Names . . . 65
A.12 Identifiers and Operators . . . 65
A.13 Theories and Development Relations . . . 65

B Symbols and Keywords **66**
B.1 ASCII Versions of RSL Symbols . . . 66
B.2 RSL Keywords . . . 66

1 Introduction

RAISE (Rigorous Approach to Industrial Software Engineering) was originally developed during 1985–90 by a European collaborative project in the ESPRIT-I programme involving four companies, two in Denmark and two in the UK. A second project, LaCoS (Large-scale Correct Systems using Formal Methods) was a continuation ESPRIT-II project (1990–95) involving nine companies in seven European countries. LaCoS further developed the RAISE technology, particularly the method and tools [1], and tested RAISE on a wide range of software development projects [2].

The RAISE Specification Language (RSL) is a formal specification language, i.e. a language with a formal, mathematical basis [3, 4, 5] intended to support the precise definition of software requirements and reliable development from such definitions to executable implementations. Particular aims of the language were to support large, modular specifications, to provide a range of specification styles (axiomatic and model-based; applicative and imperative; sequential and concurrent), and to support specifications ranging from abstract (close to requirements) to concrete (close to implementations).

In this paper we provide an introduction to the RAISE Specification Language and to the RAISE method. Complete information can be found in the books on RSL [6] and the method [5].

There were a few minor changes made to RSL between the RSL book and the method book, and we follow the latter. We also include a few extensions to RSL that have been introduced since the method book was published, and that are supported by the tools available from UNU/IIST [7].

2 The RAISE Specification Language

The RAISE Specification Language (RSL) is a modular language. Specifications are in general collections of (related) modules. There are two kinds of modules: *schemes* and *objects*. Schemes are (possibly parameterised) *class expressions*, and objects are instances of classes. We return to schemes and objects later in Sections 2.10, 2.11, and 2.12. For now, if you have an intuition about classes and objects in object-oriented programming languages, then this intuition largely carries over into RSL.

2.1 Basic Class Expressions

There are several ways of making class expressions, but the most common is the *basic class expression* that consists of the keywords **class** and **end** around some *declarations* of various kinds. Each declaration is a keyword followed by one or more *definitions* of the appropriate kind (Table 1).

No declarations are compulsory: many classes just contain type and value declarations. The order in the table is a common one to use, but any order is allowed, and there may be more than one occurrence of a kind of declaration.

Table 1: Declarations and their definitions.

Declaration	Kind of definition
object	Embedded modules
type	Types
value	Values: constants and functions
variable	Variables that may store values
channel	Channels for input and output
axiom	Axioms: logical properties that must always hold
test_case	Test cases: expressions to be evaluated by a translator or interpreter

2.2 Types

RSL, like most specification and also programming languages, is a *typed* language. That is, it must be possible to associate each occurrence of an identifier representing a value, variable or channel with a unique type, and to check that the occurrence of the identifier is consistent with a collection of typing rules. Such rules, such as that typically prohibiting expressions like "1 + **true**", are well known from programming languages and we will not describe them further here.

2.2.1 Built-in types

In order to be able to define the types of values etc. we need a collection of types to use. RSL has seven built-in types (Table 2), and a number of ways of constructing other types from these.

Table 2: Built-in types.

Type	Example values	Operators
Bool	**true**, **false**	$\wedge$, $\vee$, $\Rightarrow$ $\sim$
Int	..., -1, 0, 1, ...	$+$, $-$, $*$, $/$, $\backslash$, $\uparrow$, $<$, $\leq$, $>$, $\geq$, **abs**, **real**
Nat	0, 1, ...	Same as for **Int**
Real	..., -4.3, ..., 0.0, ...	$+$, $-$, $*$, $/$, $\uparrow$, $<$, $\leq$, $>$, $\geq$, **abs**, **int**
Char	'a', ...	
Text	"", "Alice", ...	As for lists of **Char**
Unit	()	

Equality = and inequality ≠ are also defined for all types.

Technically, the operators for **Bool** are properly referred to as *connectives*. They differ from operators in that a "lazy" or "conditional" evaluation is used for them: see Section 2.4. $\sim$ is negation. There is no need for $\Leftrightarrow$ as it would be the same as =.

Nat is a *subtype* of **Int**: all **Nat** values are also **Int** values. The operators are mostly conventional: / for **Int** is *integer division*, and \ is *remainder*. $\uparrow$ for both **Int** and **Real** is exponentiation; **abs** for both **Int** and **Real** gives the absolute value. **Int** is not a subtype of **Real**: the operator **real** converts from **Int** to

Real, and the operator **int** from **Real** to **Int**, truncating towards zero.

Unit is a type with just one value "()", also written as **skip**. It is used mainly in imperative and concurrent specifications to provide a parameter type for functions that do not need parameters, and to provide a return type for functions that do not return values.

The operators and other symbols used to construct value expressions (which we will see later in this paper) are listed in Table 3. They are listed in increasing order of precedence (P), so the prefix operators bind most tightly. The column headed A indicates those that are associative, either right (R) or left (L).

Table 3: Value expression precedences.

P	Symbols	A
14	λ $\forall$ $\exists$ $\exists!$	R
13	$\equiv$ **post**	
12	$\Box$ $\sqcap$ $\parallel$	R
11	;	R
10	:=	
9	$\Rightarrow$	R
8	$\vee$	R
7	$\wedge$	R
6	$= \neq > < \geq \leq \subset \subseteq \supset \supseteq \in \notin$	
5	$+ - \backslash \hat{} \cup \dagger$	L
4	$* / \circ \cap$	L
3	$\uparrow$	
2	:	
1	$\sim$ prefix operators	

2.2.2 Type Constructors

There are a number of type constructors for creating types from other types, illustrated in Table 4.

Table 4: Type constructors.

Ctr	P	A	Example expressions	Operators
$\times$	2		(1,true,'a')	
-set	1		{}, {1,2}	**hd**, $\in$, $\notin$, $\cup$, $\cap$, $\subset$, $\subseteq$, $\supset$, $\supseteq$, **card**, $\backslash$
*	1		$\langle\rangle$, $\langle 1,2 \rangle$	**hd**, **tl**, $\in$, $\notin$, $\hat{}$, **len**, **elems**, **inds**
$\overrightarrow{m}$	3	R	[ ], ['a' $\mapsto$ **true**, 'b' $\mapsto$ **false**]	**dom**, **rng**, **hd**, $\in$, $\notin$, $\cup$, $\dagger$, $\backslash$, /, $\circ$
$\rightarrow$	3	R	λ x : **Int** • x + 1	$\circ$
$\tilde{\rightarrow}$	3	R	λ (x,y) : **Int** $\times$ **Int** • x / y	$\circ$

The column headed P indicates the binding precedence of the type constructors, where 1 is the highest. The column headed A indicates the constructors that are right (R) associative; the others do not associate.

So, for example:

$\mathbf{Int} \times \mathbf{Real\text{-}set} \rightarrow \mathbf{Real}^* \rightarrow \mathbf{Bool}$
means
$(\mathbf{Int} \times (\mathbf{Real\text{-}set})) \rightarrow ((\mathbf{Real}^*) \rightarrow \mathbf{Bool})$

The product constructor $\times$ is used to form tuples. These may be pairs, triples, ... of any types. This constructor is not associative. For example, **Int** $\times$ **Text** $\times$ **Char** and **Int** $\times$ (**Text** $\times$ **Char**) are different types: the first is a triple, the second a pair containing a singleton and a pair.

-set, * and $\overrightarrow{m}$ create finite sets, list and maps respectively. There are also the potentially infinite set (**-infset**), infinite list ($^\omega$) and infinite map ($\widetilde{\overrightarrow{m}}$) constructors, but they are rarely used.

card gives the number of elements in a (finite) set; **len** gives the length of a (finite) list. For example:

card $\{\} = 0$
len $\langle$'a', 'b', 'a'$\rangle = 3$

The operator $\frown$ is the concatenation operator for lists. For example:

$\langle 1, 2\rangle \frown \langle 2, 3\rangle = \langle 1, 2, 2, 3\rangle$

Maps are relations, or associations, between pairs of values. Values on the left of the pairs forming the association are said to form the *domain*, and those on the right are said to form the *range*. Finite maps ($\overrightarrow{m}$) are required to be one-one or many-one, not one-many or many-many. In other words, a value in the domain must not be associated with more than one value in the range. For example, the type **Int** $\overrightarrow{m}$ **Int** contains the value $[1 \mapsto 2, 2 \mapsto 4]$ but not the value $[1 \mapsto 2, 1 \mapsto 4]$. The **dom** operator returns the domain (a set) and **rng** returns the range (also a set). For example:

dom $[\,] = \{\}$
dom ['a' $\mapsto$ **true**, 'b' $\mapsto$ **true**] = {'a', 'b'}
rng ['a' $\mapsto$ **true**, 'b' $\mapsto$ **true**] = {**true**}

The union ($\cup$) of two maps is formed as if the maps were two sets of pairs and the union of the two sets were the result. But it only gives a finite, many-one map if the domains are disjoint: see below. The override operator $\dagger$ forms a map by taking the union of the two domains, and associating each domain value with the appropriate range value from the second map, if any, otherwise that from the first map. So the second takes precedence over, or "overrides", the first. For example:

['a' $\mapsto$ **true**, 'b' $\mapsto$ **true**] $\cup$ ['a' $\mapsto$ **false**, 'c' $\mapsto$ **false**] =
  ['a' $\mapsto$ **true**, 'a' $\mapsto$ **false**, 'b' $\mapsto$ **true**, 'c' $\mapsto$ **false**]
['a' $\mapsto$ **true**, 'b' $\mapsto$ **true**] $\dagger$ ['a' $\mapsto$ **false**, 'c' $\mapsto$ **false**] =
  ['a' $\mapsto$ **false**, 'b' $\mapsto$ **true**, 'c' $\mapsto$ **false**]

We see that the union of two deterministic maps can be non-deterministic (and hence in the type of possibly infinite maps constructed by $\underset{m}{\tilde{\rightarrow}}$), unless their domains are disjoint, while override preserves determinacy. So it is good practice either to never use union, or to only use it when the domains are disjoint.

There are two ways of reducing, or restricting a map. \ (the operator also used for set difference) subtracts a set of elements from the domain. / restricts the domain to values in its second argument. For example:

$[1 \mapsto \textbf{true}, 2 \mapsto \textbf{false}] \setminus \{2,3\} = [1 \mapsto \textbf{true}]$
$[1 \mapsto \textbf{true}, 2 \mapsto \textbf{false}] \; / \; \{2,3\} = [2 \mapsto \textbf{false}]$

$\rightarrow$ is the constructor for forming *total* functions. A total function is one that always returns a value when it is applied and always returns the same value for the same argument. If a function returns some value, we say it *terminates*, and if a function always returns the same value for the same argument we say it is *deterministic*. So a total function is one that terminates and is deterministic for all arguments. Consider tossing coins on a low-gravity planet as a function, with the coin as an argument. It is non-deterministic, because each coin sometimes lands one way up, sometimes the other. If gravity is so low that very light coins are tossed into orbit, then the function does not terminate for some arguments, as we wait for ever for the coin to land. A function that is not known to be total for all arguments is called *partial*, and $\tilde{\rightarrow}$ is the constructor for partial functions.

We can define functions using "lambda-expressions" as shown in Figure 4, though these are not often used. The first, total function is the "add one" function for integers. The second, partial function, is the integer division function. This is partial because it is not defined for division by zero.

The operator **hd** applied to a non-empty set returns an arbitrary value from the set. For a non-empty list, **hd** returns the first element. For a non-empty map, **hd** returns an arbitrary element from the domain of the map. **hd** is not defined when its argument is empty, so it is a partial operator. The definition of **hd** for sets and maps was added to RSL after the publication of the two books on RAISE [6, 5].

For non-empty lists, **tl** returns the list obtained by removing the first element. Note that **hd** returns an element, **tl** a list. For example:

$\textbf{hd} \langle 1, 2 \rangle = 1$
$\textbf{tl} \langle 1, 2 \rangle = \langle 2 \rangle$

$\in$ and $\notin$ for sets are conventional. For a list they refer to the element set; for a map they refer to the domain. For example:

$(1 \in \{\}) = \textbf{false}$
$(1 \in \langle 0, 2 \rangle) = \textbf{false}$
$(1 \notin [1 \mapsto \text{'a'}, 2 \mapsto \text{'b'}]) = \textbf{false}$

The definition of $\in$ and $\notin$ for lists and maps was added to RSL after the publication of the two books on RAISE [6, 5].

Lists and maps may be applied like functions. For lists, the argument is an integer in the range one to the length of the list inclusive. So an empty list cannot be applied, a list of length one can be applied only to one, a list of length two to one or two, etc. When the argument can be applied, the result is the corresponding element of the list. For example:

$\langle$'a', 'b'$\rangle$(1) = 'a'

The **elems** of a list is the set of elements of it, and the **inds** (the indexes) of a list is the set of possible integer arguments that it can be applied to. For example:

elems $\langle$'a', 'a'$\rangle$ = {'a'}
inds $\langle$'a', 'a'$\rangle$ = {1, 2}

For maps, the possible arguments that it can be applied to are the values in the domain, and the result is the corresponding value in the range. Since we insist that finite maps are many-one, finite map application to values in the domain is deterministic.

The operator $\circ$ is available for maps and functions, with the basic property that, for two maps or two functions `f` and `g`:

$(f \circ g)(x) = f(g(x))$

2.2.3 Type Expressions

Type expressions are defined as one of the following:

- a built-in type
- a user-defined type
- a type formed from type expression(s) using a type constructor
- a *subtype* of another type expression

Subtypes are types that contain only some of the values of another type, the ones that satisfy a predicate. For example, the type **Nat** is defined as the subtype

$\{|\ i : \mathbf{Int} \bullet i \geq 0\ |\}$

That is, it is a subtype of **Int**, and is the type containing those integers that are at least zero.

Subtypes are commonly defined using functions, which makes them easier to read. For example, suppose we wanted to define dates as triples of the form (day, month, year), then we might use the subtype

{| (d, m, y) : **Nat** × **Nat** × **Nat** • is_date(d, m, y) |}

where the predicate (boolean function) `is_date` is defined elsewhere, to constrain `m` to the range one to twelve, and to constrain `d` according to `m` and whether `y` is a leap year.

2.2.4 Type Definitions

Users can define their own types, and there are two kinds of type definitions. *Abbreviation* definitions just define identifiers that one can use instead of the defining expression. For example, here is a type declaration containing two type abbreviation definitions:

type
  Date_base = **Nat** × **Nat** × **Nat**,
  Date = {| (d, m, y) : Date_base • is_date(d, m, y) |}

Type abbreviation definitions take the form "identifier = type expression" and, like all kinds of definitions, are separated by commas.

The second kind of type definition introduces an identifier for a new type. This kind comes in four forms:

- abstract types, or *sorts*
- record types
- variant types
- union types

2.2.5 Abstract types

These are just type identifiers. An abstract type is a type we need but whose definition we haven't decided on yet. They are commonly used for two purposes:

- There are many simple types, like identifiers for people, bank accounts, books in a library, departments of an organisation, etc., that we expect to implement very easily in the final program, perhaps as numbers, or characters, or strings. All we need is to use = to compare them, and = is defined for all types, even abstract ones. There is a standard piece of advice in specification that you don't choose a design until you have to, so we typically leave such types abstract. We may later discover during design that it is useful to distinguish between identifiers for reference books and those for books that may be borrowed, and we can then design a type with a suitable structure. An added bonus is that different abstract types are regarded as different by the type checker, so we avoid the danger of using a person's identifier for a book: the type checker will report an error.
- Sometimes we want to delay the design of a type not because it is simple, but for the opposite reason: because it is complicated and we don't yet know what the design should be. There is more on this when we discuss the RAISE method, especially in Section 3.2.

2.2.6 Records

Records in RSL are very much like those common in programming languages. Here is an example that might be found in a system for a bookshop:

type
  Book ::
    title : **Text**
    author : **Text**
    price : **Real** $\leftrightarrow$ new_price

This defines a new type `Book` as a record with three components. Each component has an identifier, called a *destructor*, and a type expression. Optionally a record component can have a *reconstructor*. In our example the third component has a reconstructor `new_price`.

Destructors are total functions from the record type to their component's type expression. For example, the type of `price` is

Book $\rightarrow$ **Real**

So we can apply `price` to a value of type `Book` to get its price. For a book value `b`, we write `price(b)`, as `price` is a function, rather than `b.price` as would be found in some languages.

Reconstructors are total functions that take their component's type expression and a record to generate a new record. The type of `new_price` is

Real $\times$ Book $\rightarrow$ Book

When we write, say, `new_price(17.95, b)` we get a new book value with the same title and author as `b`, but with the price component set to 17.95.

A record type definition also provides, implicitly, a *constructor* function for creating a record value from its component values. The identifier of the constructor is formed by putting `mk_` on the front of the identifier of the type, so in our case we have a constructor `mk_Book` of type

Text $\times$ **Text** $\times$ **Real** $\rightarrow$ Book

and we can write, say, mk_Book("`Oliver Twist`", "`Charles Dickens`", 9.95) as a book value.

2.2.7 Variants

Variant types allow us to define types with a choice of values, perhaps with different structures. The simplest case is rather like the enumeration type found in some programming languages, such as:

type
  Colour == red | green | yellow

This defines a new type called `Colour` and three (different) constants (`red`, `green`, and `yellow`) of type `Colour`.

But variant types allow richer structures. For example, the following type defines binary trees holding values of some type `Val`:

type
  Tree == nil | node(left : Tree, val : Val, right : Tree)

This defines a new type `Tree`, a constant `nil` of type `Tree`, a constructor `node` of type

$\text{Tree} \times \text{Val} \times \text{Tree} \rightarrow \text{Tree}$

and destructors `left`, `val` and `right`. The type of left, for example, is

$\text{Tree} \overset{\sim}{\rightarrow} \text{Tree}$

The destructors are partial because they are not defined for `nil` trees.

Records are in fact special cases of variants: single ones. We could have defined the same type `Book` that we used as an example of a record:

type
  Book == mk_Book(title : **Text**, author : **Text**, price : **Real** ↔ new_price)

This illustrates the fact that variants, like records, can optionally include reconstructors.

The type `Tree` is recursive: trees are defined in terms of trees. Variants are the only type definitions that allow recursion.

2.2.8 Unions

Union type definitions allow us to make new types like variants out of existing types. Suppose types `B` and `C` are defined somewhere. Then we can define a type `A` as their union:

type
  A = B | C

This is in fact a shorthand for a variant, in which the identifier `A`, and the type names `B`, and `C` are used to generate constructor and destructor identifiers:

```
type
  A == A_from_B(A_to_B : B) | A_from_C(A_to_C : C)
```

In order for these constructor and destructor identifiers to be generated, the constituents of a union must be names of user-defined types, and not general type expressions.

With union types, implicit (unwritten) *coercions* are allowed from union components to the union type. Suppose, for example, a function `f` has `A` as its parameter type. Then we can apply `f` to a value `c` of type `C`, simply by writing `f(c)`. This is short for `f(A_from_C(c))`. We could similarly apply `f` to values from `B`.

2.3 Values

Having introduced types, we can consider the values that populate the types. We first see how to define values. We define values within value declarations, where a value declaration consists of the keyword **value** followed by one or more value definitions separated by commas.

The simplest value definition takes the form "identifier : type expression", and is called a *typing*, for example:

```
value
  x : Int
```

This may look like a variable declaration in a language like C (though the order of identifier and type is reversed in C) but it is really a constant declaration. `x` is the identifier of a value, not of a variable: a variable is a location where values can be stored, and the stored value can be changed. There is a possible confusion between the way programmers use the term variable (which is the way we use it) and the way a mathematician uses the term. The mathematician means by a variable something whose value is not known, or does not matter, not something whose value may change. The constant `x` defined above is more like a variable in the mathematical sense: it is a constant but we don't know, without more information, what its value is. Such constants are not allowed in programming languages, because there is not enough information about them. They are useful in specification when, for example, we want to describe a lift (elevator) system without saying how many floors the building has: the lift system can be described for an arbitrary building.

Continuing with the same example, we might want to assume that the number of floors is at least two. It is hard to imagine what a lift would do in a one storey building, or what a building with zero or a negative number of floors would look like. So we might use an *implicit value definition*:

```
value
  floors : Int • floors ≥ 2
```

(The type **Int** here could be replaced by **Nat** without changing the meaning.) `floors` is a constant, but it must satisfy the *predicate* (logical expression) that follows the bullet •. The definition is implicit in that we still don't know what the actual value of `floors` is.

Sometimes we know the value of a constant: the constant identifier is just a convenient shorthand (and, as in a program, makes things easier to maintain). We can use an *explicit value definition*:

value
  floors : **Int** = 20

All three forms of value definition start with a typing, an identifier and a type separated by a colon. The same applies if we want to define functions. First, a function definition may just be a typing, as in:

value
  name : Person $\rightarrow$ **Text**

This definition says that there is a total function from the type `Person` to the type **Text**, i.e. "every person has name". It is used typically when we haven't yet decided how to represent a person, i.e. `Person` is still an abstract type. Implicitly, it says there must be enough information in the type `Person` for a name to be extracted.

We can also define functions implicitly, with a *postcondition*:

value
  square_root : **Real** $\stackrel{\sim}{\rightarrow}$ **Real**
  square_root(x) **as** r **post** r $\geq$ 0.0 $\wedge$ r$*$r = x
  pre x $\geq$ 0.0

This defines a function to produce square roots, but without specifying how they should be calculated. It requires that the result `r` should satisfy the predicate following **post**: it should not be negative and its square must equal the parameter `x`. Since **Real** numbers only have **Real** square roots when they are not negative, it is a partial function and we give it a *precondition*.

This function illustrates the fact that the type **Real** in RSL contains the mathematical real numbers. This function is in practice not *implementable* in a programming language using limited precision arithmetic, and we might prefer a specification requiring the result `r` to be within some machine-dependent tolerance of the mathematical square root.

The types **Int** and **Nat** are similarly not implementable in normal computer arithmetic, because their values are unbounded. In practice this is usually not a problem because we can be sure that the values used or generated will not be so large as to cause over- or underflow. If it is a problem we would have to write a specification of how arithmetic in the actual implementation behaves.

The final kind of value definition is the *explicit function definition*. Here is an example:

value

factorial : **Int** $\stackrel{\sim}{\to}$ **Int**
factorial(n) ≡ **if** n = 1 **then** 1 **else** n ∗ factorial(n−1) **end**
pre n > 0

We need a precondition here since our version of `factorial` is non-terminating for 0 or negative numbers. The definition of `factorial` illustrates a *recursive* function, one that is defined in terms of itself. It also illustrates the **if** expression in RSL.

2.3.1 Overloading and Distinguishable Types

Value identifiers in definitions may be *overloaded*, i.e. the same identifier may be used to define different values, provided their types are *distinguishable* by the type checker. Types are distinguishable unless they are subtypes of the same type. For example, **Nat** is not distinguishable from **Int** (or any subtype of **Int**) because they are both subtypes of **Int**. (Any type is a subtype of itself.) Similarly $\to$ is not distinguishable from $\stackrel{\sim}{\to}$, nor $\underset{m}{\to}$ from $\underset{m}{\stackrel{\sim}{\to}}$, nor **-set** from **-infset**, nor * from $^{\omega}$. **Int** and **Real** are distinguishable.

Built-in operators may be overloaded. For example, we might define a new version of "+" as follows:

value
+ : **Real** × **Int** → **Real**
x + y ≡ x + **real** y

This is possible as the type of "+" is distinguishable from both possible types of the built-in infix operator "+", which are

Int × **Int** → **Int**
Real × **Real** → **Real**

2.4 Logic

We have seen several examples of predicates, expressions that (we hope) evaluate to **true** or **false**. But we have to clarify several issues in order to define our *logic*. In particular, we will need to define:

- what happens when expressions do not terminate, and
- what we mean by equality.

We know it is (unfortunately) easy enough to write programs that do not terminate. The problem is present in specification as well, but we need to be very clear about what it means. We could, for example, have written a poor definition of factorial, forgetting the precondition:

value
  poor_factorial : **Int** $\rightarrow$ **Int**
  poor_factorial(i) $\equiv$ **if** i = 1 **then** 1 **else** i $*$ factorial(i$-$1) **end**

and then ask what the expression `poor_factorial(0)` means. The technical answer is **chaos**, a special expression in RSL that represents an expression whose evaluation does not terminate. We need to distinguish in general between *expressions* and *values*. Constants like **true** and 0 are expressions that evaluate to themselves. "1 + 1" is an expression that evaluates to the value 2. **chaos** is an expression that does not evaluate: it does not terminate. So what about an expression like "**chaos** + 1"? The general rule in RSL is "left-to-right" evaluation, which means in this case we evaluate the left argument of +, and if this terminates with a value, we evaluate the right argument. If this also terminates with a value, we add the two values to get the value of the whole expression. If either argument does not terminate, neither does the whole expression. So "**chaos** + 1" is equivalent to **chaos**. So is "0 * **chaos**" that arises when we evaluate `poor_factorial(0)`, that you might have thought should be 0. All infix operators are evaluated the same way.

Equality, =, is an infix operator. So if we try to express the equivalence between "0 * **chaos**" and **chaos** we should not write

(0 $*$ **chaos**) = **chaos**

because this expression would evaluate to **chaos**, not to **true**. We write instead

(0 $*$ **chaos**) $\equiv$ **chaos**

where the symbol $\equiv$ is read as "is equivalent to". Technically, two expressions are equivalent when their semantics, their meanings, are equivalent. For values, and more generally for any expressions that are deterministic, terminating, and read-only (do not write to variables or do input or output on channels) equivalence and equality are the same.

We use the equivalence symbol in explicit function definitions, and we can now explain what a function definition means, namely "when the precondition is true, the function application is equivalent to the defining expression". This definition does not say anything about the situation when the precondition is not true. So, for example, we cannot say what `factorial(0)` is. The definition tells us nothing: it may be **chaos**, or it may be some integer. We say it is *underspecified*. This does not make it a bad specification. Rather, it tells us to be careful only to use `factorial` when we are sure the argument is positive. We will see later in Section 4.8 that there is a tool, called the confidence condition generator, to help us check this.

It seems sensible to be able to assert as true that

n > 1 $\Rightarrow$ (factorial(n) = n $*$ factorial(n$-$1))

for any integer n. This should be true for 0, so we want

$0 > 1 \Rightarrow (\text{factorial}(0) = 0 * \text{factorial}(0-1))$

to be true. That is, we want

false ⇒ **chaos**

to be true. This means that ⇒ should not behave like an infix operator, and in RSL it does not. We call the symbols ⇒, ∧, ∨ and ∼ *connectives* and define them according to the rules, for any expressions e_1, e_2, e:

$e_1 \Rightarrow e_2$ ≡ **if** e_1 **then** e_2 **else true end**
$e_1 \wedge e_2$ ≡ **if** e_1 **then** e_2 **else false end**
$e_1 \vee e_2$ ≡ **if** e_1 **then true else** e_2 **end**
∼e ≡ **if** e **then false else true end**

To understand these, we need the evaluation rule for **if** expressions. This is:

1. Evaluate the expression following **if**.
2. If this does not terminate, the **if** expression does not terminate.
3. If it evaluates to true, evaluate the expression following **then**.
4. If it evaluates to false, evaluate the expression following **else**.

You can check that the definitions of the connectives and the evaluation rules for **if** expressions give the same results as "classical" logic, which is only concerned with the values **true** and **false**. For example:

false ⇒ **false**
≡ **if false then false else true end** — definition of ⇒
≡ **true** — evaluation rule for **if** expression

But now we also know what will happen when some expressions do not terminate. For example, the following all evaluate to true:

false ⇒ **chaos**
∼(**false** ∧ **chaos**)
true ∨ **chaos**

The reason for including **chaos** in RSL is not that it is needed in specifications: you normally do not want your programs to loop forever! It is a useful convenience in expressing the proof theory of RSL, which is what we mean by the logic. (And even if **chaos** were not included, you could write a variety of equivalent expressions, such as "**while true do skip end**".)

The logic in RSL is called a *conditional* logic as it is based on conditionals (if expressions). There are other approaches to the problems of non-terminating expressions, such as the "logic of partial functions" (LPF) [8, 9] which is used by the specification language VDM [10]. Without going into the argument as to which is better, we note two things:

- $\vee$ and $\wedge$ in RSL are only commutative if their arguments terminate. For example:

  (**true** $\vee$ **chaos**) $\equiv$ **true**
  (**chaos** $\vee$ **true**) $\equiv$ **chaos**

- The connectives in RSL are implementable, because they can be translated using if expressions in programming languages, which evaluate just like RSL if expressions.

For LPF the opposite holds: $\vee$ and $\wedge$ are always commutative, but the connectives are in general only implementable when their arguments terminate.

2.4.1 Quantifiers

RSL includes the *quantifiers* $\forall$ (for all), $\exists$ (there exists) and $\exists!$ (there exists exactly one). For example, the following are all true expressions:

$\forall$ i : **Int** • (i ∗ 2) / 2 = i
$\forall$ i : **Nat** • $\exists$ j : **Nat** • j = i + 1
$\exists!$ i : **Int** • i $\leq$ 0 $\wedge$ i $\geq$ 0

The quantification is over *values* in the type. It does not include expressions like 1/0 or **chaos**.

2.4.2 Typings

What follows the quantifier is always a typing, just like the start of every kind of value definition. But we can have more general forms of typing than just an "identifier : type expression": the identifier can be a *binding*.

2.4.3 Bindings

A binding is commonly just an identifier, but it can be parentheses enclosing two or more bindings separated by commas. So the following are all bindings:

x
(x,y)
(x,(y,z))

The identifiers in a binding must all be different.

In a typing, the structure of a binding must match the structure of the type: if the binding is for a product, so must the type be. For example, if `Pair` is defined as an abbreviation for **Int** × **Int**, the possible typings include the following:

x : Pair
(x,y) : Pair
((p,q),(x,y)) : Pair × Pair

but "(x,y) : **Int**", for example, is not possible.

Bindings also occur as the *formal parameters* of implicit and explicit function definitions (like the `n` in factorial(n) ≡ ...). What about a function `f` with type

A × B → ...

Does this have two parameters or one? In RSL you can take either view: the formal application can be written `f(a,b)` or `f((a,b))`, or even `f(p)` (where `p` is a binding for a pair).

2.5 Value Expressions

We have already seen the literals, infix and prefix operators for various types in Section 2.2, the boolean connectives, if expressions and quantified expressions in Section 2.4. There are some other value expressions that we describe in this section.

2.5.1 Set Expressions

Sets may be formed in three ways:

1. *enumerated* sets like {} (the empty set), or {1,3,2}.
2. *ranged* sets (for integers only) like {1..3}, which is equal to the second enumerated set example. If the second number in the range is less than the first, the ranged set is empty.
3. *comprehended* sets like { i/2 | i : **Int** • i ∈ {2..7} }, which is again equal to the second enumerated set example. The predicate following • (called a *restriction*) may be omitted, in which case it is as if it were **true**.

Other expressions may of course also represent sets. For example, a function may return a set and then an application of the function will be a set expression, an expression whose type is `T`-**set** for some type `T`. Similar remarks apply for lists and maps.

2.5.2 List Expressions

Lists may be formed in three ways:

1. *enumerated* lists like ⟨⟩ (the empty list), or ⟨2,1,2,3⟩.
2. *ranged lists* (for integers only) like ⟨1..3⟩, which is equal to the tail of the second enumerated list example. If the second number in the range is less than the first, the ranged list is empty.
3. *comprehended* lists like ⟨ i/2 | i **in** ⟨2..10⟩ • i < 8 ⟩ which is again equal to the tail of the second enumerated list example. As with enumerated sets, the restriction may be omitted. A comprehended list takes its elements from another list expression, rather than a typing as with a set, and, see below, a map.

2.5.3 Map Expressions

Maps may be formed in two ways:

1. *enumerated* maps like [ ] (the empty map), or [1 ↦ **true**, 3 ↦ **true**, 2 ↦ **false**].
2. *comprehended* maps like [i ↦ is_odd(i) | i : **Int** • i > 0 ∧ i < 4], which is again equal to the second enumerated map example (assuming an appropriate definition of `is_odd`). The restriction may be omitted, in which case it is as if it were **true**.

2.5.4 Let Expressions

Let expressions are used in two main ways:

1. to destruct a product. For example:

   let (x,y) = (1,2) **in** x + y **end**

   will evaluate to 3. First we evaluate the expression following the =. Then we bind `x` to the first part, and `y` to the second. Finally we evaluate the expression following the **in**.

2. to organise an evaluation into several steps. For example, a function to sum a list of integers might be defined as:

   ```
   value
     sum : Int* → Int
     sum(s) ≡
       if s = ⟨⟩ then 0
       else
         let h = hd s, t = tl s, x = sum(t) in h + x end
       end
   ```

   This is particularly useful when the sub-expression like **hd** `s` would, without the **let**, occur more than once. But even when this would not occur, **let** expressions often improve readability.

2.5.5 Case Expressions

Case expressions are commonly used to express functions over lists and over variant structures. For example, the `sum` function could be written:

value
  sum : **Int*** → **Int**
  sum(s) ≡
    case s **of**
      ⟨⟩ → 0,
      ⟨h⟩⌢t → h + sum(t)
    end

A **case** expression consists of a series of *patterns* plus associated expressions. The case patterns are tried in order, the first pattern that matches is taken, and the associated expression evaluated. The pattern ⟨⟩ matches the empty list. The pattern `⟨h⟩⌢t` matches a non-empty list, and at the same time binds `h` to the head and `t` to the tail.

An example of a **case** expression for a variant type is the body of a function to calculate the depth-first traversal of a tree (Section 2.2.7), returning a list of the values in the nodes of the tree:

value
  traverse : Tree → Val*
  traverse(t) ≡
    case t **of**
      nil → ⟨⟩,
      node(l, v, r) → traverse(l) ⌢ ⟨v⟩ ⌢ traverse(r)
    end

The bindings in patterns may be replaced by "wildcards", underscores, when their values are not needed. For example, a function to calculate the depth of a tree (assuming `max` is defined somewhere):

value
  depth : Tree → Val*
  depth(t) ≡
    case t **of**
      nil → 0,
      node(l, _, r) → 1 + max(depth(l), depth(r))
    end

The most commonly used case patterns are for lists and variants, but literals are also possible, and there is a "wildcard" pattern _ that matches anything. For example, a strange definition of `is_odd`:

value

```
is_odd : Nat → Bool
is_odd(n) ≡
  case n of
    0 → false,
    1 → true,
    _ → is_odd(n−2)
  end
```

2.6 Axioms

So far we have seen type and value declarations. There are also axiom declarations, introduced by the keyword **axiom** and consisting of axiom definitions separated by commas. Each axiom definition is a predicate, optionally preceded by an identifier in square brackets. For example, instead of defining:

```
value
  floors : Int • floors ≥ 2
```

we could write:

```
value
  floors : Int
axiom
  [ floors_constraint ] floors ≥ 2
```

In fact all value definitions, functions as well as constants, can be written in this style, a typing plus an axiom. There are "axiomatic" or "algebraic" specification languages, like Larch [11] and CASL [12], that use only this style, and are also restricted to abstract types. This style can be used within RAISE, but we choose also to have available the pre-defined sets, lists, maps, and products that are characteristic of the "model-based" specification languages like Z [13], B [14], and VDM [10].

2.7 Test Cases

Test cases have no semantic meaning: they are like comments directed at an interpreter or translator meaning "please provide code to evaluate these expressions and report the results".

The syntax of test cases is much like axioms, except that the test case expressions can be of any type. For example, if we wanted to test the function to sum a list of integers we might define

```
test_case
  [ sum0 ] sum(⟨⟩),
  [ sum1 ] sum(⟨1,2,2,3⟩)
```

and expect to see the results

```
[sum0] 0
[sum1] 8
```

But a perhaps more useful style of test case is to include the expected result in the test case, i.e. to write

test_case
  [sum_0] sum(⟨⟩) = 0,
  [sum_1] sum(⟨1,2,2,3⟩) = 8

so that the output for every test case should be `true`.

Test cases are always evaluated in order of definition. This is useful for imperative specifications, introduced below in Section 2.8, when there are variables storing information. Information stored as a result of one test case is available for the next one, so we can, for example, test use-cases step-by-step as a sequence of test cases, outputting intermediate observations as the result of each.

Test cases were added to RSL after the publication of the two books on RAISE [6, 5].

2.8 Imperative Constructs

Much of the specification in RSL is done with the language described so far in this paper, which we call *applicative*. This comes from the fact that this style of writing specifications (or programs in languages like Lisp, SML, and Haskell) is mainly done in terms of definitions and applications of functions. It is advantageous in that it is close to mathematics, and so supports reasoning in the way mathematics does.

Perhaps the most common style of programming is the *imperative* style of languages like Pascal and C. These depend on the use of *variables*, essentially locations in a store that allow values to be stored and retrieved. RSL allows specifications to use this style, but the RAISE method suggests (Section 4.1) that it only be used in later stages of development, as a step towards an implementation in a programming language.

To support the imperative style, RSL includes **variable** declarations, assignment expressions, and sequences of expressions. For example:

variable
  counter : **Nat** := 0
value
  increment : **Unit** → **write** counter **Nat**
  increment() ≡ counter := counter + 1 ; counter

Here we have a variable declaration with one variable definition, consisting of an identifier, a type, and, optionally, an initial value. There is also a function `increment` which will add 1 to the counter. `increment`

needs no parameters, so we give it none and use **Unit** as its argument type. It returns a **Nat**, so this appears as its result type. But we also include an *access* clause in its type, that says it may write, and so change, the contents of the variable `counter`. (**read** is the other kind of access; **write** includes **read**.) Accesses are normally written using one or more names of variables, but we may also use the *universal access* **any** to allow access to any variable defined in the same module.

The body of `increment` consists of two expressions combined by ";", the *sequential combinator*. The assignment is an expression rather than, as in many languages, a statement: there are no statements in RSL, only expressions. This avoids the need for keywords like "return" that just convert an expression to a statement in languages like C. Since an assignment is an expression it must have a type, and this is **Unit**. There is a rule that, for two expressions combined by ";", the first must have type **Unit** and the type of the whole expression is the type of the second.

For an indication of why imperative features cause problems with reasoning, consider the two predicates:

increment() = increment()
increment() ≡ increment()

Is either of these true? Consider the first, and recall the rule for evaluating infix operators like equality: evaluate the left expression; if it terminates evaluate the right; if that also terminates compare the results. It should be clear that the result will be false. In addition, the evaluation of the equality has had the *effect* (commonly called a "side effect" as if it didn't matter too much) of adding 2 to the counter. Yet the equality looks as if it should be true, certainly to any mathematician!

Now consider the second predicate. Is `increment()` equivalent to itself? It seems natural that we should say yes, and indeed it is so: any expression (even one that does not terminate) is equivalent to itself. We defined equivalence as semantic equivalence, and with imperative constructs we can say that expressions are equivalent if (given the same initial state, i.e. the same initial values in all variables) they are either both non-terminating, or they have the same effect on the state and return the same result. The evaluation of the equivalence, unlike the evaluation of the equality, does not change any variables. It simply returns true if the hypothetical questions "Would these two expressions have the same effect?" and "Would these two expressions return the same result?" are both answered with "Yes".

2.8.1 If Expressions

We have seen **if** expressions before, but there are two features we have not described:

1. When there are several alternatives we can include one or more **elsif** clauses. For example:

   type
     Compare == greater | equal | less
   value
     compare : **Int** × **Int** → Compare
     compare(x, y) ≡
       if x > y **then** greater
       elsif x = y **then** equal
       else less **end**

elsif clauses are equivalent to nested **if** expressions in the obvious way.

2. For **if** expressions of type **Unit** the **else** clause may be omitted, as in:

```
value
  decrement : Unit → write counter Unit
  decrement() ≡ if counter > 0 then counter := counter − 1 end
```

The missing else clause is equivalent to "**else skip**", where **skip** is the "do nothing" expression (of type **Unit**).

2.8.2 Iterative Expressions

There are **while**, **until** and **for** loops in RSL. Consider the following examples:

```
variable
  counter : Nat,
  result : Real
value
  sum1 : Nat →~ write counter, result Real
  sum1(n) ≡
    counter := n;
    result := 0.0;
    while counter > 0 do
      result := result + 1.0/(real counter);
      counter := counter − 1
    end;
    result
  pre n > 0,

  sum2 : Nat →~ write counter, result Real
  sum2(n) ≡
    counter := n;
    result := 0.0;
    do
      result := result + 1.0/(real counter);
      counter := counter − 1
    until counter = 0 end;
    result
  pre n > 0,

  sum3 : Nat →~ write result Real
  sum3(n) ≡
    result := 0.0;
    for counter in ⟨1..n⟩ do
      result := result + 1.0/(real counter)
    end;
    result
  pre n > 0
```

It should be apparent that the three functions $\texttt{sum}_1$, $\texttt{sum}_2$ and $\texttt{sum}_3$ all compute the same value, namely

$$1 + 1/2 + ... + 1/n \tag{1}$$

Are the preconditions necessary? There are two possible reasons for them:

1. We don't know what the expression (1) would mean if `n` is zero.
2. We don't want to divide by zero.

It is clear that only $\texttt{sum}_2$ needs the precondition for the second reason, to avoid dividing by zero. (For the **for** loop, remember that a ranged list is empty when the second number is less than the first. A **for** loop is equivalent to **skip** when the list is empty.)

2.8.3 Local Expressions

It is clear in the examples of loops that it would be better if the variables `counter` and `result` had a smaller scope: they should be inside the functions that use them, when their purpose is clearer. There is a *local expression* in RSL that allows variable declarations, or any other kind of declarations, to be local to an expression. Here is another version of $\texttt{sum}_1$ using a **local** expression:

```
value
  sum1 : Nat ~→ Real
  sum1(n) ≡
    local
      variable
        counter : Nat := n,
        result : Real := 0.0
    in
      while counter > 0 do
        result := result + 1.0/(real counter);
        counter := counter − 1
      end;
      result
    end
  pre n > 0
```

2.9 Concurrency

Concurrency in RSL is based on synchronisation between concurrently executing expressions (commonly called processes) using channels. Channels are declared with the keyword **channel** followed by one or more channel definitions separated by commas. A channel definition is one or more channel identifiers plus a type, the type of data that may be transmitted on the channel(s). For example:

```
type
  Data
channel
  left, right : Data
```

Functions that use channels for input and output need **in** and **out** access to channels, similar to access rights for variables. Consider the following definition:

```
value
  one_place_buffer : Unit → in left out right Unit
  one_place_buffer() ≡
    while true do
      let x = left? in right!x end
    end
```

`left?` is an *input expression* which returns the value made available on the channel `left`. (Hence `left?` has type `Data`.) `right!x` is an *output expression* that puts a value onto the channel `right`. (`right!x` has **Unit** type.) So `one_place_buffer` repeatedly accepts values on the `left` channel and outputs them on the `right` channel.

Channel communication in RSL is point-to-point and synchronised: in order for an input like `left?` or an output like `right!x` to execute it is necessary for another concurrent process to be executing a corresponding output or input, i.e. in the opposite direction on the same channel. So executing on its own, `one_place_buffer` can do nothing: it will wait for ever on the input expression `left?`. This situation, an expression waiting for ever for input or output, is termed *deadlock*, and there is an expression for it in RSL: **stop**. In practice we will want to avoid it occurring.

`one_place_buffer` is a kind of *server*, and is supposed to run forever. But we earlier characterised a non-terminating process as **chaos**. To resolve this, we redefine non-terminating as meaning a process which either runs forever without doing input or output (**chaos**) or deadlocks (**stop**). We might think that stop means terminate, but termination includes the idea of allowing the next expression in sequence to execute, while deadlocking means that no further progress is possible.

If there are two or more other concurrent processes waiting to output on `left` then an arbitrary choice of communication is made between them: one will pass its value to `one_place_buffer`, the other(s) will continue to wait. Then, if there are two or more processes waiting for input on the `right` channel, `one_place_buffer` will synchronise with just one of them, passing its value, and leaving the other(s) waiting. Such behaviour, with arbitrary choices between possible behaviours, is usually undesirable, and we often try to avoid it by making sure that there is only one process inputting and one outputting on each channel. Buffers typically serve just one process putting values in and one process taking them out. But some servers, like databases, for example, are intended to support many concurrent users.

Buffers are common components used to allow processes to run at different speeds. If a process `A` wants to send data to another process `B`, and we use a channel between them, then `A` can only send a data item when `B` is ready. If instead we place a `one_place_buffer` between them, `A` can place a value, `v`, in the buffer by executing `left!v` and continuing. The expression that represents `A` and `B` running in parallel with the `one_place_buffer` is

```
A ∥ B ∥ one_place_buffer()
```

"‖" is the *concurrent composition* combinator. It requires that all its arguments have type **Unit**, and the result is type **Unit**. It is associative and commutative, so its arguments may be written in any order.

With a one place buffer the solution to the problem of processes of different speeds is limited: `A` cannot get more than one item ahead of `B`. If `A` wants to do a second output on `left`, before `B` has done an input on `right`, then `one_place_buffer` will be waiting on the output `right!v` and `A` will have to wait.

A better solution is to have a buffer that can contain many values. Here is a buffer with maximum capacity of `max` items:

```
value
  max : Nat
variable
  buff : Data* := ⟨⟩
channel
  put, get : Data
  empty : Unit
value
  buffer : Unit → write buff in put, empty out get Unit
  buffer() ≡
    while true do
      empty? ; buff := ⟨⟩
      ⎕
      if len buff < max
      then let x = put? in buff := buff ⁀ ⟨x⟩ end
      else stop end
      ⎕
      if buff ≠ ⟨⟩
      then get!hd buff ; buff := tl buff
      else stop end
    end
```

The expression for `buffer` uses *external choice* "⎕" to offer, in general, three choices to its clients:

1. An output on the `empty` channel causes the buffer to be emptied.
2. Provided the buffer is not full, an output of a value `v` on the `put` channel causes `v` to be appended to the buffer.
3. Provided the buffer is not empty, an input on the `get` channel will remove a value from the buffer and return it.

The choice is called "external" because it is the client processes that decide which choice is taken: the server `buffer` will cooperate with whatever is asked of it.

If we have two clients trying to interact with `buffer` at the same time, perhaps one doing a `put` and the other a `get`, an *internal* (arbitrary) choice will be made between them, to choose which interacts first. Such internal choices are common in the execution of concurrent systems. But we always design servers using external choice "⎕", as if we used internal choice "⌈⌉" their behaviour would seem very erratic to

their clients. The reason for this is that if a client wanted to do a `put`, say, (and the buffer was not full), the nondeterministic version of `buffer` could (even in the absence of any other clients) internally choose, say, the first choice of waiting for a communication on `empty`, resulting in deadlock.

What happens when the buffer is full or empty? We would like in these cases for the `put` or `get` choice respectively to not be available, forcing clients wanting to use these channels to wait. One might expect to see **skip** in the relevant **else** clauses of `buffer`, but in fact **stop** is what we need. The reason for this is that **stop** turns out to be the unit for external choice (like 0 is the unit for addition, and 1 for multiplication): it satisfies, for any expression `e`, the equivalence

e $\lbrack\rbrack$ **stop** $\equiv$ e

We now consider another example. Suppose we have a database, initially specified as a map from `Key` to `Data`, and we need a `lookup` function. This would be partial, as the key might not be in the database. Part of our applicative specification might be:

type
  Db = Key $\overrightarrow{m}$ Data
value
  lookup : Key $\times$ Db $\stackrel{\sim}{\rightarrow}$ Data
  lookup(k, db) $\equiv$ db(k)
  pre defined(k, db),

  defined : Key $\times$ Db $\rightarrow$ **Bool**
  defined(k, db) $\equiv$ k $\in$ db

Now suppose we want to make a concurrent database server, to allow multiple concurrent users. The standard approach says that there will be one choice in the server for each function in the applicative version. Functions like `lookup` and `defined` will need a channel to pass the input (in each case of type `Key`) and also a channel to pass back the result (of type `Data` and **Bool** respectively).

But this approach is inadequate with partial functions like `lookup`. In the concurrent case with multiple clients we can get patterns like:

1. `A` asks if key `k` is defined, and gets the result **true**
2. `B` deletes key `k`
3. `A` tries to lookup key `k`

We need to combine `A`'s interactions into a single transaction that is atomic in the sense that, once started, no other interaction with the database is possible until it is completed. The way to do this is to create a total version of lookup, that can return a `Data` value or a “not found” value. We need a new type:

type
  Result == not_found | res(data : Data)

and then the relevant choice in the server process, assuming a variable `db` holding the database, is:

```
let k = lookup? in
  if defined(k, db) then lookup_res!res(lookup(k, db))
  else lookup_res!not_found
  end
end
```

A further improvement is to also define an "interface process" `lookup` to be used by clients:

```
value
  lookup : Key → out lookup in lookup_res Result
  lookup(k) ≡ lookup!k ; lookup_res?
```

This has two advantages over allowing clients access to the channels directly:

1. The channels can all be hidden, allowing tighter control over access to them. We will see how to hide things in Section 2.10.
2. The interface processes enforce the right protocol. In our case we see that the interface process `lookup` does an output on the channel `lookup` and then an input on `lookup_res`. We need to check that there is a choice in the server that offers the dual of this: an input on `lookup` followed by an output on `lookup_res`. Provided this is the case, no deadlocks are possible if clients can only call the interface processes. With direct access to the channels a client could contain an error like an output on `lookup` without the input that should follow it, which could cause the database to deadlock, waiting to do an output on `lookup_res`.

2.9.1 Comprehended Expressions

The combinators ⌷, ⌈⌉, and ‖ may be applied to comprehended sets of expressions of type **Unit**. For example:

```
‖ { A[i].init() | i : Index }
```

is a comprehended expression representing the parallel execution of the `init` functions of all the objects in `A`, which is an object array (see Section 2.12) indexed by the type `Index`. As with comprehended sets, lists, and maps (Section 2.5), the comprehended expression may include a restriction. For example:

```
⌷ { A[i].put(d) | i : Index • ok(i) }
```

is a comprehended expression representing an external choice between invoking the `put` functions from objects in the array `A` whose indexes satisfy the condition `ok`. If there are no such indexes, the comprehended expression will deadlock.

2.10 Modules

As we mentioned earlier, there are two kinds of module in RSL, schemes and objects. Schemes are essentially classes, and objects are instances of classes, so the basic thing is the class expression. These come in six forms: basic, extending, renaming, hiding, with, and instantiation.

2.10.1 Basic Class Expressions

These were introduced in Section 2.1. They consist of the keywords **class** and **end** with any number of declarations between them. The declarations (and their constituent definitions) may come in any order. There is no "define before use" rule in RSL. All the entities defined in the class expression are exported (visible outside it) by default: there is nothing like an "export" clause in RSL.

2.10.2 Extending Class Expressions

If C_1 and C_2 are class expressions:

extend C_1 **with** C_2

is an extending class expression. The declarations of C_2 are added to those of C_1. The declarations of C_2 can refer to entities defined in C_1, but not vice versa. The declarations of C_1 and C_2 must be *compatible*, which simply means that duplicate definitions are not allowed, any more than they would be in a single class expression.

2.10.3 Renaming Class Expressions

If C is a class expression:

use id_1' **for** id_1, ..., id_n' **for** id_n **in** C (n $\geq$ 1)

is a renaming class expression in which the entities id_1, ..., id_n are exported with identifiers id_1', ..., id_n': they are renamed. The entities may be types, values, variables, channels or objects.

2.10.4 Hiding Class Expressions

If C is a class expression:

hide id_1, ..., id_n **in** C (n $\geq$ 1)

is a hiding class expression from which the identifiers id_1, ..., id_n are not exported. Hiding is most commonly used to hide objects, variables, channels and *auxiliary* functions (functions only intended for use within the original class to define other functions). Hiding is used to prevent access from outside the class, and also used to hide auxiliary functions or other entities that we don't expect to use in later developments, because hidden entities do not need to be implemented.

2.10.5 With Class Expressions

If `C` is a class expression:

with O_1, ..., O_n **in** C (n ≥ 1)

is a with class expression. O_1, ..., O_n are object expressions (see Section 2.12). The meaning of **with** `X` **in** `C` is that an applied occurrence of a name `N` in `C` can mean either `N` or `X.N`, so that, in particular, we can write just `N` instead of `X.N`. (It is similar to "using namespace" in C++.)

The with class expression was added to RSL after the publication of the two books on RAISE [6, 5].

2.10.6 Scheme Instantiations

If we define a scheme called `S`, say:

scheme S = C

then we can use `S` to mean the class expression `C`, for example in "**extend** S **with** ...": the occurrence of `S` here just means the same as `C`. The occurrence of `S` is called an *instantiation* of `S`.

But it is also possible to *parameterise* a scheme, and we discuss this in the following section.

2.11 Parameterised Schemes

The most common use of parameterised schemes is to make *generic* schemes. For example, we considered earlier the type of binary trees. We may want more than one kind of binary tree: one to hold integers, another to hold names, etc. But we would like to define the type `Tree` and its associated functions only once. We can proceed as follows:

- We define a class to act as the scheme parameter. Commonly we use a scheme to define this class:

  scheme ELEM = **class type** Elem **end**

  This is a very simple, as well as a very common scheme to define a parameter. But there are no restrictions on what we can put into a parameter's class expression. This makes the parameterisation mechanism in RSL much more powerful than, for example, templates in C++.

- We define a generic scheme `TREE` using `ELEM` as a parameter:

  scheme TREE(E : ELEM) =
  class
    type
      Val = E.Elem,
      Tree == nil | node(left : Tree, val : Val, right : Tree)
    ...
  end

  The abbreviation definition of `Val` is just a commonly used convenience. We could omit it, replacing all other occurrences of `Val` with `E.Elem`.

  Technically the parameter "`E : ELEM`" is like an object definition (see Section 2.12). `E` is the identifier of an object, so `E.Elem` means the type `Elem` defined in the object `E`.

So how do we make trees of integers, say? We need to make an instantiation of `TREE`, and the actual parameter we need is an object, just as the formal parameter is an object. So we define an object `I`, say:

object I : **class type** Elem = **Int end**

and now the scheme instantiation `TREE(I)` is what we want. The formal definition of `TREE(I)` says that it is the class expression of `TREE` with every occurrence of the object identifier `E` replaced by `I`. So, in particular, the defining type expression of the type `Val` will be `I.Elem`, which we can see from the definition of `I` is just an abbreviation for **Int**.

For type checking, there is a condition between the class of the formal parameter `E` and the class of the actual parameter `I`. This is that the latter must be a *static implementation* of the former. This means that for every entity in the formal parameter there must be an entity in the actual parameter of the same kind (type, object, value, variable or channel) with the same identifier and:

- for types, if the formal type definition is an abbreviation, the actual type definition must be an abbreviation for a type that is maximally the same
- for objects, the defining class in the actual parameter must statically implement the defining class in the formal parameter
- for values, variables and channels, the types in the actual and formal parameters must be maximally the same.

Here "maximally the same" means the types must not be distinguishable (see Section 2.3.1).

The actual class expression may contain more entities than the formal.

Schemes can have several parameters. For example, we might define a generic database:

scheme DATABASE(D : ELEM, R : ELEM) =

```
class
  type
    Domain = D.Elem,
    Range = R.Elem,
    Database = Domain →m Range
  ...
end
```

and we can instantiate `DATABASE` with two different objects, or the same object twice.

Sometimes we find we have an object that defines the things we need for the actual parameters, but with the wrong identifiers. For example, the RAISE method (Section 3) suggests defining a number of simple types that will be used throughout the specification in a scheme `TYPES`, and making an object `T` from this. Now suppose `TYPES` defines types `Id` and `Name`, and we want to instantiate the `DATABASE` with `Id` as the domain type and `Name` as the range type.

We can instantiate `DATABASE` as

```
DATABASE(T{Id for Elem}, T{Name for Elem})
```

The construct $\{\mathtt{id}_1{}'$ **for** $\mathtt{id}_1$, ..., $\mathtt{id}_n{}'$ **for** $\mathtt{id}_n\}$ is called a *fitting*. It acts as if the fitting had been applied to the formal parameter class as a renaming.

It is possible to have parameters which depend on each other. For example we could define:

```
scheme S(E : ELEM, T : TREE(E)) = ...
```

Then if we define objects by, say:

```
object
  I : class type Elem = Int end,
  TR : TREE(I)
```

then `S` could be instantiated as `S(I, TR)`.

2.12 Object Declarations

Technically class expressions denote, or mean, classes (collections) of possible implementations of them. We get different possible implementations with abstract types (since any type can be used as an implementation) and with underspecified values. The possible implementations are called *objects*. Object declarations consist of the keyword **object** followed by one or more object definitions separated by commas.

If `C` is a class expression, we can define an object `O` by:

object
  O : C

and `O` denotes some object in the class `C`.

If `x` is an entity in `C` (and not hidden or renamed in `C`), then, in the scope of this object definition, `x` can be referred to by the *name* `O.x`. This is sometimes called a *qualified name*, and the prefix `O` the *qualifier*.

The universal access **any** can also be qualified. For example, the access clause **read** `O`.**any** in a function signature allows the function to read any variable defined in the object `O` (including variables defined in any objects defined in `C`). This is often needed to write the signatures of functions that invoke functions in imperative modules, since variable and channel names are commonly hidden.

It is also possible to define *object arrays* in RSL. The object name is given a formal parameter in the form of a (list of) typings. For example, a collection of buffers indexed by a type `Index` could be defined by

object
  B[i : Index] : BUFFER

and the expression `B[e].put(d)`, where `e` is an expression of type `Index`, and `put` a function defined in `BUFFER`, would be used to put data value `d` in the buffer indexed by the value of `e`.

2.13 Comments

There are two kinds of comment supported in RSL. *Block comments* are opened by `/*` and closed by `*/`. They may be nested. *Line comments* are opened by `--` and closed by the end of a line (or file). Both kinds of comment are allowed anywhere where white space would be allowed.

Line comments were introduced, and the original restriction on the use of block comments to only certain syntactic constructs was removed, after the publication of the two books on RAISE [6, 5].

3 The RAISE Method: Writing Initial Specifications

As long as you conform to the syntax and type rules of RSL, you can describe and develop software in any way that you choose. But there are a number of ideas for using RSL that have been found useful in practice, and that collectively we describe as “the” RAISE method.

There are two main activities involved in the method: writing an initial specification, and developing it towards something that can be implemented in a programming language, and we describe these separately in this Section 3 and the following Section 4.

Writing the initial specification is the most critical task in software development. If it is wrong, i.e. it fails to meet the requirements, then following work will be largely wasted. It is well known that mistakes made

early in the life-cycle are considerably more expensive to fix than those made later, precisely because they cause so much time and effort to be expended going in the wrong direction. But we should clarify this to say that it is mistakes made *and not quickly found* that are expensive. We can't guarantee that we won't make mistakes, but if we can discover them quickly then not too much harm is done.

What kind of errors are made at the start? The main problem is that we may not understand the requirements. They are set in some domain in which we are usually not experts, while the people who wrote them, to whom the domain is familiar, tend to forget to explain what to them is obvious.

In addition, requirements are written in a natural language, like English or Chinese, and as a result are likely to be ambiguous. They are often large documents developed by several people over a period of time. As a result they are often contradictory: what they say on one page may differ from what they say on another.

The aim of the initial specification is to capture the requirements in a formal, precise manner. Formality means that our specification has just one meaning, it is unambiguous. By *capturing* the requirements we mean rewriting them in our terms, creating our model of what the system will do. So how can we check that the model we create accurately models what the writer of the requirements has in mind?

3.0.1 Be Abstract

The specification should be *abstract*, it should leave out as much detail as possible. The requirements may demand that identifiers have a certain format, or that dates should be presented in a particular style, or that calculations should be done to a certain degree of accuracy, or that a user screen should have a certain appearance, but we try to extract the essential information: that there are identifiers, presumably different for each different entity they identify, that we need dates, that certain calculations need to be done, that users may be requested for certain information and as a consequence they may be presented with other information, or the system's state may be changed in certain ways. We know that we can fill in the details later: we can design screens provided the information to be presented is available or can be calculated, and provided we know what input to demand.

3.0.2 Use Users' Concepts

The concepts in the specification should be the same as the user's concepts. If the requirements say that each customer has an account, and an account is a record of all the customer's transactions, then that is what the specification should say. It should not refer to concepts like databases, tables, and records: these are computer concepts that describe ways of solving the problem, while what we want to do first is *describe the problem, not its solution.*

3.0.3 Make it Readable

Specifications are intended to be read by others: by those who are to check that they correspond to requirements, by those who are to implement them, by those who are to write test plans, by those who later want to maintain the system, etc. So we want to make them as readable as possible. The guidelines here are very much like those for programming languages: meaningful identifiers, comments, simple functions, modules that are coherent and loosely coupled, etc.

3.0.4 Look for Problems

We recall that what we want to do is avoid mistakes, or find them quickly. So we concentrate on the things that appear difficult, strange, or novel, and we ignore or defer things that are straightforward. We might be mistaken as to what is hard, of course, but we hope that with some experience we have a feeling for such things. In capturing requirements we are also trying to find out if the system we intend to develop is feasible, at least within our budget constraints, and so we want to be assured as early as possible that we have appropriate solutions to all the problems. If we don't, we may need to do some experimentation or research before we commit ourselves further.

3.0.5 Minimise the State

State information should be *minimal*. This means in particular that we try hard not to include in the state *dependent* information: information that can be calculated from other information in the state. If C can be calculated from A and B, then we should not model C as part of the state. If C is stored as part of the state, together with A and B, then we will need a *consistency* condition that what is stored for C is the same as would be calculated from the stored A and B. There is a general notion that the simpler the set of consistency conditions needed, the better the state is designed. It may be that later we decide we need to store C, to achieve sufficient speed, but this should be done as a later stage of development.

When we refer to the *state* of a system we mean the information that is stored, that persists between interactions with it. We also speak of the state of a module, where we mean the part of the state associated conceptually with that module, which will typically provide functions to change it and report on it. We use the term *global state* where necessary to refer to the state of the whole system, as opposed to that of a module, or of a group of modules that we see as a subsystem.

3.0.6 Identify Consistency Conditions

While we try to make the state minimal, it is still usually the case that we need *consistency* conditions and *policy* conditions. Consistency conditions are needed if some possible state values cannot correspond to reality: two users of a library borrowing the same copy of a book simultaneously, perhaps. Policy conditions are ones that might perhaps arise in reality, but we intend that they should not happen: a user borrowing too many books at one time, perhaps.

If our system's state cannot correspond to reality then it becomes essentially useless: it cannot tell us who really has the book, and we probably cannot trust any information it might give us. Preserving consistency conditions is more critical for the healthiness of our system than keeping within policy.

We identify the consistency requirements first because sometimes we can think of a state design that will reduce the need for consistency conditions. For example, if we record a borrower against a copy of a book, only one such borrower can be recorded and the inconsistency of two simultaneous borrowers cannot occur. We need to bear the consistency conditions in mind during development, as we will want our functions to maintain consistency, and our initial state to establish it.

Sometimes consistency is dealt with by a subtype: we can record the number of books someone can borrow as a **Nat**, for example, to prevent it being negative. But often consistency requirements will involve more than one module, and then it is generally better to define a function expressing it, but

not try to impose it as a subtype. When there are several modules involved it may not always be true during processing: we will merely want to establish that, starting from a consistent state, every top-level function will generate another one.

There are several common sources of possible inconsistency that arise in many domains, because they relate to common data structures:

- Much data is modelled as maps, allowing us to use identifiers as references. These identifiers may then be used elsewhere, and we need to ensure that every reference is to data that exists. For example, the borrower of a copy of a book should be a registered user.
- Sometimes we have relations that relate values of some type to itself, like "child" or "part of" relations. Then we typically need to ensure that there are no cycles in the relation, or else functions using the relation are likely not to terminate.
- It may be possible to access information in two ways (which is an indication that our state is not minimal, but may be done for efficiency reasons, especially in refinements of the initial specification). Then we need to check that the two ways to access information give the same result. If we can find out borrowers from information about copies of books, and find out copies borrowed from information about borrowers, then we can state as a consistency conditions (a) that the recorded borrower of a book (if any) has a borrow record for that copy for that book, and (b) that each copy in the set of copies borrowed by a borrower has the borrower recorded.

Consistency conditions help us write functions, or at least they help us avoid mistakes in functions that would occur if we overlooked consistency. They also have a relation to preconditions. Preconditions serve two main purposes:

1. They allow us to avoid unsafe or unpredictable situations, like dividing by zero, or in general applying a function or operator when its result would be undefined or non-terminating.
2. They allow us to avoid situations where we would otherwise break consistency. So a function `borrow`, for example, might include in its precondition that the user involved is registered.

It is not usually a good idea to include consistency as part of preconditions. The reason for this is that functions at the top level, accessible by our users (people or other software), will generally need to have preconditions checked when they are invoked. Checking consistency typically involves searches through all the state and this would be too inefficient. (At the same time, including a simple check even though it is implied by consistency is sensible as part of "safety-first" style.) We instead, as we mentioned above, take steps during development to ensure that our functions all preserve consistency, and that our initial state establishes it, so we can then assume it to be true.

Policy conditions are generally separated from consistency. States that violate policy requirements are possible in the real world, and if our system is to be a faithful model of the real world it must also allow them. Such states are often used to generate warning messages, raise alarms, or instigate corrective actions, so we still need to define precisely what the policy conditions are so that we can specify how to check them.

3.1 Kinds of Module

We identify two kinds of module that we find most commonly used: *global objects* and *state components*.

3.1.1 Global Objects

Global objects are objects declared at the top level, in a separate file. In general, they are not advised, because they have too wide a scope. But there are typically a collection of, in particular, types that we need in many places, such as identifiers for various kinds of entity, and it is convenient to collect these in one global object. Dates and a few functions or operators like $\leq$ to compare them, and perhaps also periods modelled as pairs of dates, or a date and a duration, are other common candidates. Global objects should not include any part of the state.

Another guide to when types should be in a global object is that types visible to users, i.e. types that occur as parameters to user functions or in the results of user functions, should generally be defined in one.

3.1.2 State Components

Most modules will contain a type modelling (a part of) the state, together with functions to *observe* it and *generate* values of it, and we term these state components. Generators usually include functions to change state values, and perhaps also to create them. The type is often called the *type of interest* of the module. Such modules are usually defined as schemes, and typically instantiated within others, as we will see in Section 3.3. Modules should have only one type of interest.

We write separate modules for each state component because we can then enforce a discipline that the part of the state within the module is only accessed through the functions defined for it. This enables us, for example, to change the way that part is modelled without affecting anything else, so long as we maintain the original properties. Such a technique is known as *encapsulation* through *information hiding*.

Object oriented approaches to program design follow the same ideas: they typically call the observers and generators *methods*.

The discussion in the following sections is almost entirely concerned with state component modules, and we will in particular talk of the type of interest, the observers and the generators of a module.

3.2 Abstract and Concrete Modules

There are various ways of writing modules, according to way in which the type of interest is defined. We will illustrate these with the idea of a bank account, seen as a record of transactions. We will keep the example very simple, as our purpose here is not to describe banking in detail but to show approaches to modelling. We assume that transactions have a date (the type `Date` defined in a global object `T`), an `Amount` (similarly defined globally, with + also defined for it), and some other information as yet

undetermined. We will need functions `open`, to create a new account, `add` to add a transaction, and `balance` to calculate the current balance.

We start with the easiest specification to write, which is concrete as it uses a concrete model of the type of interest `Account`: an account is an ordered list of transactions:

```
scheme CONC_ACCOUNT =
with T in class
  type
    Account = {| trl : Transaction* • is_ordered(trl) |},
    Transaction ::
      date : Date
      amount : Amount
      info : Info,
    Info
  value
    open : Account = ⟨⟩,

    add : Transaction × Account ~→ Account
    add(tr, ac) ≡ ⟨tr⟩^ac
    pre can_add(tr, ac),

    can_add : Transaction × Account → Bool
    can_add(tr, ac) ≡
      case ac of
        ⟨⟩ → true,
        ⟨h⟩^t → date(h) ≤ date(tr)
      end,

    balance : Account → Amount
    balance(ac) ≡
      case ac of
        ⟨⟩ → 0,
        ⟨h⟩^t → amount(h) + balance(t)
      end,

    is_ordered : Transaction* → Bool
    is_ordered(ac) ≡
      case ac of
        ⟨⟩ → true,
        ⟨_⟩ → true,
        ⟨h1, h2⟩^t → date(h2) ≤ date(h1) ∧ is_ordered(⟨h2⟩^t)
      end
end
```

We realised that `add` needs to be partial, since we should not be able to add a transaction older than the latest one in the account. We adopt a useful convention to define a function `can_f` to express the precondition of a function `f`. This allows writers of functions elsewhere to check the precondition using `can_f`, and to be somewhat insulated from changes to `can_f` that might be made later.

This very simple specification raises some immediate questions, such as whether `open` should be a function with some parameters, such as the date of opening, or information about the account holder? This is at least in part a question about the requirements, and is typical of the questions that naturally arise from the specification that can indicate missing requirements (either missing from the requirements or missed by us in reading them). We might similarly wonder, since there is a function to add a transaction, if there should also be one to delete or change an existing transaction. And if so, should transactions have identifiers? Similarly, if an account can be opened, can it also be closed, and if so, do its records disappear?

There is also a question about the type `Transaction`: Should it be in a module of its own? We might consider this not necessary, because it is so simple: the functions needed to generate and observe it are already embodied in its definition and the module would have nothing more to define. But in fact the appropriate conclusion here is that `Transaction` should probably be defined in the global object `T`, because it is visible to users, who will have available some function to display or print their account. In the following versions we will assume that `Transaction` is defined in `T`. We also realise another missing or missed requirement: that we need another function to report on transactions:

value
  transactions_since : Date × Account → Transaction*
  transactions_since(d, ac) ≡ ⟨ tr | tr **in** ac • d ≤ date(tr) ⟩

The concrete module `CONC_ACCOUNT` says that an account is precisely its sequence of transactions. But this may not be an adequate model. It might in the final implementation be an archive of previous years' transactions plus this year's stored separately and more immediately available. And balances are in practice not calculated from the beginning each time, but perhaps stored and updated with each transaction. Such a balance would be an example of dependent information not included in the initial specification.

Here is an abstract version of the account module:

scheme ABS_ACCOUNT =
with T **in class**
  type
    Account
  value
    /* generators */
    open : Account,
    add : Transaction × Account $\xrightarrow{\sim}$ Account,
    /* observers */
    can_add : Transaction × Account → **Bool**,
    balance : Account → Amount,
    transactions_since : Date × Account → Transaction*
  axiom
    [can_add_open] ∀ tr : Transaction • can_add(tr, open) ≡ **true**,

    [can_add_add]
      ∀ tr_0, tr_1 : Transaction, ac : Account •
        can_add(tr_1, add(tr_0, ac)) ≡ date(tr_0) ≤ date(tr_1)
        pre can_add(tr_0, ac),

```
  [balance_open] balance(open) ≡ 0,

  [balance_add]
    ∀ tr : Transaction, ac : Account •
      balance(add(tr, ac)) ≡ amount(tr) + balance(ac)
      pre can_add(tr, ac),

  [transactions_since_open] ∀ d : Date • transactions_since(d, open) ≡ ⟨⟩,

  [transactions_since_add]
    ∀ d : Date, tr : Transaction, ac : Account •
      transactions_since(d, add(tr, ac)) ≡
        if d ≤ date(tr) then ⟨tr⟩^transactions_since(d, ac)
        else ⟨⟩ end
      pre can_add(tr, ac)
end
```

Abstract specifications in this style define the type of interest `Account` as an abstract type, and the constants and functions as signatures only. Axioms then relate observers to generators.

Most people do not find such specifications easy to write. There is an alternative style that gives a specification that is also abstract, in that it allows the same room for the type `Account` to be developed further, but is easier to write – which means less likely to contain errors!

We define the type `Account` abstractly, and then define one or more *main observers* that return precisely the type we used in the concrete version `CONC_ACCOUNT`. (We would use more than one observer if the concrete type had been a product or record.)

```
scheme ACCOUNT =
with T in hide transactions, balance1 in class
  type
    Account,
    Transactions = {| trl : Transaction* • is_ordered(trl) |}
  value
    transactions : Account → Transactions,

    open : Account • transactions(open) = ⟨⟩,

    add : Transaction × Account ~→ Account
    add(tr, ac) as ac′ post transactions(ac′) = ⟨tr⟩^transactions(ac)
    pre can_add(tr, ac),

    can_add : Transaction × Account → Bool
    can_add(tr, ac)
      as b post
        b ⇒
          case transactions(ac) of
            ⟨⟩ → true,
            ⟨h⟩^t → date(h) ≤ date(tr)
```

```
        end,

    balance : Account → Amount
    balance(ac) ≡ balance1(transactions(ac)),

    balance1 : Transaction* → Amount
    balance1(trl) ≡
      case trl of
        ⟨⟩ → 0,
        ⟨h⟩^t → amount(h) + balance1(t)
      end,

    transactions_since : Date × Account → Transaction*
    transactions_since(d, ac) ≡ ⟨ tr | tr in transactions(ac) • d ≤ date(tr) ⟩,

    is_ordered : Transaction* → Bool
    is_ordered(ac) ≡
      case ac of
        ⟨⟩ → true,
        ⟨_⟩ → true,
        ⟨h1, h2⟩^t → date(h2) ≤ date(h1) ∧ is_ordered(⟨h2⟩^t)
      end
end
```

Here the type `Transactions` is just a convenience, and the main observer is `transactions`. Constants are defined by implicit value definitions, and functions by post conditions. The specification of `can_add` says that the new transaction date being no earlier than the last transaction in the account is a necessary, but not a sufficient condition for `add`: we envisage the possibility of adding some means of closing an account, perhaps, or insisting that the balance must always be non-negative. But we are sufficiently sure to define `balance` and `transactions_since` in terms of `transactions`. The main observer `transactions` is not defined: we can only do so when we decide exactly what the type Account will be. It is usually hidden, and we have also hidden the extra function $\texttt{balance}_1$ we used to define `balance`.

The concrete version `CONC_ACCOUNT` can easily be obtained as a refinement of `ACCOUNT` by implementing the main observer `transactions` as the identity function. But there are many other possible refinements with a richer state with more information, such as adding the possibility for accounts to be opened and closed, or including the current balance.

3.3 Module Hierarchies

There are several suggested principles in creating a collection of modules to model a system:

- Each module should have only one type of interest, defining functions to create, modify and observe values of the type.
- The modules should as far as possible form a *hierarchy*: each module below the top one should be instantiated in only one other, its *parent*, as an embedded object, and its functions should only be called from its parent.

This leads naturally to a top-down style of specification and development. As we decide on the concrete type for a module, perhaps involving several components, then as long as these component types are non-trivial we define new modules for them as children of the original.

The restriction to a hierarchy sometimes seems more complicated than, say, a collection of global objects each defining one part of the state, with objects able to call functions in any others. But such designs have definite disadvantages:

- The many interdependencies mean that changes to a module may affect many others, so maintenance is more difficult.
- They are harder to test individually. With a hierarchy there is natural testing order that tests children before parents.
- In a concurrent system it is hard to ensure that the system will not deadlock. Following the guidelines for developing concurrent systems from sequential ones in Section 4.2 means that freedom from deadlock is guaranteed by a simple syntactic check.

It may not be clear why we suggested using embedded objects to link child modules to their parents. There are three possibilities to use one module (the child) in another (the parent), which we consider in turn:

1. Merging the specifications textually into a single module. This is clearly not very sensible. Apart from breaking the suggestion that there only be one type of interest per module, the resulting large module is hard to read, the child cannot be reused elsewhere, it is tedious to hide the child components (as they must be hidden individually), and there may be name clashes between the two parts.
2. Writing the parent as an extension (**extend** `S` **with** ... where `S` is the scheme defining the child). This gives two separate modules, and so is readable, and the child module `S` can be reused, but it still suffers from the disadvantages that it is hard to hide the child components, and there may be name clashes between the two parts. (We typically use **extend** to add definitions to an existing type of interest, or perhaps to make a subtype of it, such as defining an interest-bearing deposit account by extending a basic account specification.)
3. Instantiating the child as an object within the parent. The separate modules are small and readable, the child is reusable, the child can be hidden merely by hiding its object identifier, and name clashes cannot occur because within the parent specification all the entities from the child have an object identifier qualifier. Hence this is normally the best solution.

3.4 Sharing Child Modules

Consider the proposed module structure in Figure 1.

If we take the advice about instantiating children as objects in parents, then in `SYS` we get two objects, called S_1 and S_2 perhaps, and in each of SUB_SYS_1 and SUB_SYS_2 we get an object `B`, say, instantiating `BUFFER`. How many buffers are there? There are two. We can see this because in `SYS` they have names $S_1.B$ and $S_2.B$, and RSL is constructed so that different names imply different objects: there is no possibility of "aliasing", of having different names for the same variable, channel or object. Different

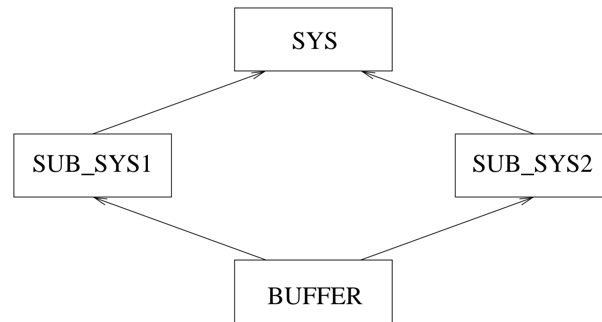

Figure 1: Sharing a child module.

objects will have different variables, different channels, and different embedded objects, even if they are instantiations of the same scheme.

If the buffers are intended to be different, this is fine. But what if the two sub-systems want to share one buffer, perhaps for passing information between them? This will break the normal idea of hierarchical design that child modules are independent, since a call in SYS of a function in S_1, say, can result in a change in state of both B and S_2. But sometimes it is necessary. We then have to be more careful than usual how we call child functions from SYS.

If we need such a design, there are two ways to achieve it. The first is to make a global object B, say, from BUFFER, and use this in both SUB_SYS_1 and SUB_SYS_2. Now there is one buffer (because there is only one name for it) and so the two sub-systems must be sharing it. But other modules can also access it. What we probably want is for the buffer to be shared between the sub-systems, but be hidden within SYS.

The second solution us to use parameterisation. We make BUFFER a parameter of both SUB_SYS_1 and SUB_SYS_2:

scheme SUB_SYS_1(B : BUFFER) = ...
scheme SUB_SYS_2(B : BUFFER) = ...

and in SYS we define the following objects:

object
  B : BUFFER,
  S_1 : SUB_SYS_1(B),
  S_2 : SUB_SYS_2(B)

Now we can see that there is only one buffer object B, which is defined in SYS and can be hidden there. The objects of the two sub-systems now share this buffer because any mention of a name prefixed by B in their specifications is now bound to that name defined in the object B in SYS.

4 The RAISE Method: Developing Specifications

In this section we consider developing the initial, applicative specification into a program. We will see that there is a standard "development route" that we often use, that develops from an applicative to an imperative style, and possibly on to a concurrent style. We will also discuss what it means for a development to be "correct", and how we can ensure correctness.

4.1 Imperative Modules

Our experience is that applicative modules, ones without any variables, following a "functional programming" style, are the easiest to write. This may seem surprising to people used to writing in imperative languages like C++ or Java, but dependence on variables also seems to encourage a style that is programming rather than specification. The idea of specification is, ideally, to write as little as possible, to model the data structures in a minimal way, trying to avoid making premature design decisions about data structures and algorithms. Above all, the aim is to meet the requirements, and the less you write the less there is to validate against the requirements.

If you want to do proof, then proofs are certainly much simpler based on applicative specifications.

But the intended implementation style is likely to be imperative. The comparative run-time efficiency of imperative programs over applicative ones is often exaggerated, but there can be definite advantages in, for example, using iteration instead of recursion if the depth of recursion is deep.

Fortunately, it is very simple to transform an applicative specification into an imperative one. This is normally only done when the design has reached the point where the type of interest is concrete, so we take the previous concrete specification `CONC_ACCOUNT` as an example. Here is the imperative version:

```
scheme IMP_ACCOUNT =
with T in hide Account, ac in class
  type
    Account = {| trl : Transaction* • is_ordered(trl) |}
  variable
    ac : Account := ⟨⟩
  value
    open : Unit → write ac Unit
    open() ≡ ac := ⟨⟩,

    add : Transaction ~→ write ac Unit
    add(tr) ≡ ac := ⟨tr⟩^ac
    pre can_add(tr),

    can_add : Transaction → read ac Bool
    can_add(tr) ≡
      case ac of
        ⟨⟩ → true,
        ⟨h⟩^t → date(h) ≤ date(tr)
      end,
```

```
    balance : Unit → read ac Amount
    balance() ≡ balance(ac),

    balance : Account → Amount
    balance(ac) ≡
      case ac of
        ⟨⟩ → 0,
        ⟨h⟩^t → amount(h) + balance(t)
      end,

    transactions_since : Date → read ac Transaction*
    transactions_since(d) ≡ ⟨ tr | tr in ac • d ≤ date(tr) ⟩,

    is_ordered : Transaction* → Bool
    is_ordered(ac) ≡
      case ac of
        ⟨⟩ → true,
        ⟨_⟩ → true,
        ⟨h1, h2⟩^t → date(h2) ≤ date(h1) ∧ is_ordered(⟨h2⟩^t)
      end
end
```

(The type `Transaction` is omitted because, following the discussion earlier, it is now assumed to be defined in the global object `T`.)

The changes from `CONC_ACCOUNT` are quite straightforward:

- A variable is defined to hold values of the type of interest. This is usually initialised if there is an obvious value to use. If the type of interest is a record or product, several variables may be used, one for each component.
- The type of interest is removed from the parameter and result types of functions, and if this leaves nothing as a parameter or result type, **Unit** is inserted.
- Generators are given write access to the variable(s); observers are given read access.
- Formal parameters of the type of interest are removed. References to them in the bodies of functions are replaced with reference to the variable(s).
- Generators include assignment(s) to the variable(s) of the new value generated.
- Constants like `open` are defined as functions in the way illustrated.
- The type of interest and the variable(s) are hidden.

Sometimes, especially with recursive functions, we define the imperative function in terms of the corresponding applicative one, as with `balance`.

The transformation is very straightforward and easily checked. There is a theorem (explained in detail in the RAISE method book [5]) that the resulting imperative module has the same properties as the applicative one. To be more precise, there is a way of rewriting the properties of the applicative one

into corresponding imperative ones that are guaranteed to hold for the imperative version. Intuitively, the applicative and imperative versions "behave in the same way". For example, starting with empty accounts and adding the same transactions to both will give the same observed values for `balance` and `transactions_since`.

This transformation is often applied to leaf modules in the hierarchy, i.e. to modules with no children, but can, as we shall see, be applied at any level in the hierarchy, with the children left applicative.

Parent modules normally have no variables: their state is the state of their children. For parent modules with imperative children the type of interest disappears altogether, and the changes to signatures also involve the removal of this type. Bodies change by calling the new imperative functions of the children, which is just a matter of removing the formal parameters corresponding to the type of interest. Since there are no such parameters needed for the children's functions this is simple.

To illustrate, suppose for simplicity that there is only one account per account-holder, and there are separate modules for storing information about account-holders and their accounts. Then the applicative parent module of `CONC_ACCOUNT`, `ACCOUNT_AND_HOLDER`, might have contained:

```
object
  A : CONC_ACCOUNT,
  H : HOLDER
type
  Account_and_holder ::
    account : A.Account
    holder : H.Holder        -- type of interest of HOLDER
value
  open : Holder_info → Account_and_holder
  open(i) ≡ mk_Account_and_holder(A.open, H.new(i))
```

The imperative version will contain:

```
object
  A : IMP_ACCOUNT,
  H : IMP_HOLDER
value
  open : Holder_info → write A.any, H.any Unit
  open(i) ≡ A.open() ; H.new(i)
```

We see that in the imperative version applicative modules are replaced by their imperative versions, and the type of interest disappears entirely.

We know that `open` is a generator, so it may change variables in the child modules. But we don't know what the variables are: they are hidden, the state of these modules is encapsulated. So we use universal accesses to indicate that the generator `open` may change any variables in the two child modules.

Since we know from the rules of hierarchical design that child modules cannot call each others' functions, their states are *independent*: a change in one cannot in itself affect the other. So it doesn't matter in which order we call the functions `A.open` and `H.new`: we could even invoke them in parallel.

This method of making leaf modules imperative encounters a difficulty when there is a collection of data at lower levels. If we replace each applicative object with one imperative one, how can we hold data about many accounts, for example? We illustrate by considering the same example at the level above. An applicative module `ACCOUNTS`, say, might have contained:

```
object
  AH : ACCOUNT_AND_HOLDER
type
  Accounts = Ac_no -m-> AH.Account_and_holder
value
  open : Ac_no × Holder_info × Accounts -~-> Accounts
  open(no, i, acs) ≡ acs † [ no ↦ AH.open(i) ]
  pre ∼ exist(no, acs),

  exist : Ac_no × Accounts → Bool
  exist(no, acs) ≡ no ∈ acs
```

The simplest solution is to make this module imperative, and keep its children applicative. Their functions are simply those used to create, modify, and observe components of the values held in the variable(s) of the parent. So instead of making `ACCOUNT_AND_HOLDER` and its children imperative, we would change `ACCOUNTS` to:

```
object
  AH : ACCOUNT_AND_HOLDER          -- still applicative
type
  Accounts = Ac_no -m-> AH.Account_and_holder
variable
  acs : Accounts := [ ]
value
  open : Ac_no × Holder_info -~-> write acs Unit
  open(no, i) ≡ acs := acs † [ no ↦ AH.open(i) ]
  pre ∼ exist(no),

  exist : Ac_no → read acs Bool
  exist(no) ≡ no ∈ acs
```

The variable `acs` holds all the accounts in a map. This is a natural style to use if the variable `acs` is to be implemented as a database. The applicative child modules are a specification of the database design.

It is also possible to specify `ACCOUNTS` with imperative child modules, when the object declaration in `ACCOUNTS` would be replaced by an object array. This is the way to specify `ACCOUNTS` if we want to program each account as a separate object, rather than, as above, using a single database for all the accounts. It is complicated by the fact that object arrays in RSL are static, so that all accounts potentially exist from the start, and we have to include information about which account numbers are currently "live". Then our imperative version of ACCOUNTS would look something like:

```
object
```

```
  AH[no : Ac_no] : IMP_ACCOUNT_AND_HOLDER        -- imperative
variable
  live : Ac_no-set := {}
value
  open : Ac_no × Holder_info ~→ write AH.any, live Unit
  open(no, i) ≡ AH[no].open(i) ; live := live ∪ {no}
  pre ∼ exist(no),

  exist : Ac_no → read live Bool
  exist(no) ≡ no ∈ live
```

4.2 Concurrent Modules

If a concurrent system is required, for example if the system is to have multiple users at the same time, or is to be distributed, concurrent modules can be developed from the imperative ones. (It is also possible to go directly from applicative to concurrent if the specification is simple.) Each imperative module with variables is transformed into a *server*. A server has a main process that is normally a **while true** loop. The purpose of the server is to ensure that the interactions with the imperative module, which is embedded in the concurrent one, are atomic. When one user starts an interaction others are forced to wait until the interaction is complete. The concurrent module is like a wrapper of the imperative one, to mediate interactions with it. To illustrate, we show a concurrent version of `ACCOUNT`:

```
scheme C_ACCOUNT =
with T in hide I, C, server in class
  object
    I : IMP_ACCOUNT,
    C :
      class
        channel
          open : Unit, add : Transaction, add_res : Add_result,
          balance : Amount, since : Date, since_res : Transaction*
      end
  value
    init : Unit → in C.any out C.any write I.any Unit
    init() ≡ I.open() ; server(),

    server : Unit → in C.any out C.any write I.any Unit
    server() ≡
      while true do
        C.open? ; I.open()
        ⌷
        let tr = C.add? in
          if I.can_add(tr) then I.add(tr) ; C.add_res!ok
          else C.add_res!fail end
        end
        ⌷
        C.balance!I.balance()
        ⌷
```

```
      let d = C.since? in C.since_res!I.transactions_since(d) end
    end,

  open : Unit → out C.open Unit
  open() ≡ C.open!(),

  add : Transaction → out C.add in C.add_res Add_result
  add(tr) ≡ C.add!tr ; C.add_res?,

  balance : Unit → in C.balance Amount
  balance() ≡ C.balance?,

  transactions_since : Date → out C.since in C.since_res Transaction*
  transactions_since(d) ≡ C.since!d ; C.since_res?
end
```

We see that the imperative module `IMP_ACCOUNT` is used to define an embedded object `I` that is hidden. A number of channels are defined for communicating the parameters and results of the functions. These can be defined in an object, here `C`, which is just a convenience for hiding them collectively. An `init` process is defined to initialise the object `I` and then start the `server`. The `server` is a loop containing an external choice, one choice for each function. It simply inputs parameters (if any) on a channel, calls the appropriate function from `I`, and outputs results (if any) on another channel. Finally there is a set of *interface processes* which are the functions that may be called to access the module. There must be one for each choice in `server`, and it must do the converse of the inputs and outputs in that choice.

If this design approach is followed and checked then it is impossible for the module to deadlock, and concurrent calls of the interface processes will be atomic, i.e. they will be executed sequentially in some arbitrary order.

As we discussed earlier in Section 2.9, we need to change partial functions like `add` into total ones. So we added an `Add_result` type, defined as a variant with two values `ok` and `fail` to the global object `T` and return an appropriate value of this type. Since `can_add` was previously only included for the parent module to check the precondition of `add`, it may now be redundant and is not included in the concurrent version: it could easily be included if still required.

It should be apparent that the concurrent version of `ACCOUNT` behaves just like the imperative one. All we have done is protect the imperative module against concurrent accesses to it interfering with each other. This theorem is formalised in the RAISE method book [5].

In the transformation to a concurrent system, children of the imperative one with variables remain applicative. Parent modules are changed in minor ways as follows:

- Instantiations of imperative modules are replaced by instantiations of the concurrent ones.
- An `init` process is added that calls the `init` processes of all its children in parallel. This means there will be an `init` process in the top level module that will, when invoked, initialise and start the servers of all its children running in parallel.
- Calls of previously partial functions need to be changed because they will have new result types.

- Sequential invocations of two or more functions in different children are normally replaced by concurrent invocations. This is safe, because we know that hierarchical design ensures that there can be no interference between the state changes of different children. So different evaluation orders will make the same changes and return the same results.

4.3 Development Route

The overall suggested development route is illustrated in Figure 2. The initial vertical line indicates development of data structures and algorithms from an initial, abstract specification. The development is done using an "invent and verify" strategy. That is, design decisions are made and reflected in a new specification. This can then be verified, shown to be correct, perhaps by proof, according to the rules discussed later in Section 4.6. The transformations to imperative and then concurrent versions may be done if needed, and are applied to concrete modules (modules with concrete type of interest). Some further refinement may be done to particular functions in the imperative and concurrent versions if needed. For example, expressions involving quantifiers might be developed into loops. Techniques for doing this are described in the RAISE method book [5].

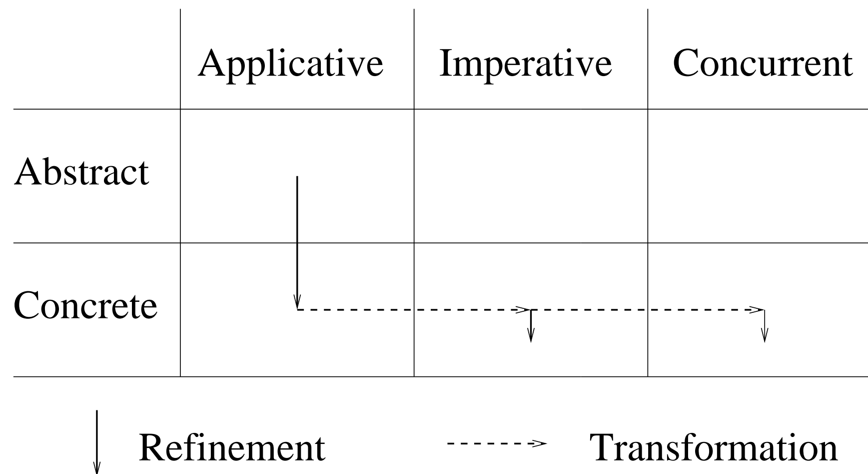

Figure 2: The RAISE development route.

4.4 Asynchronous Systems

Some systems, particularly distributed ones, will need further development into asynchronous systems, typically by introducing buffers between components instead of the "remote procedure call" design implied by the concurrent specification described in the previous Section 4.2. In outline, to make a system asynchronous, we:

- Define a collection of buffers to carry request and result messages.
- Replace calls of functions in child modules with functions that send request messages and then wait for replies to appear in the appropriate buffers.

- In modules with servers, "wrap" the interface processes with ones that wait for messages in appropriate buffers, call the interface processes to obtain results, and then send replies containing those results. These wrappers are defined as **while true** loops, so that they continually wait for request messages.

Apart from delays, and provided the buffers are fault-free, the asynchronous system will behave just like the synchronous one. Fault tolerance can be introduced by, for example, having a policy that every request is answered by an acknowledgement, and by, in the final implementation, including time-outs on the waits for replies. [15] describes such a distributed development.

4.5 Validation and Verification

Validation is the check that we have written the right specification, i.e. that we have met the requirements. It has nothing to do with internal properties: one can have a perfectly satisfactory description of a tunnel when what is wanted is a bridge, and no detailed inspection of the tunnel's description can uncover the fact that it is not what is required. Such a gross disparity between requirements and specification is unlikely, of course, but the basic fact remains: to validate a specification we must look outside it, at the requirements

Validation therefore cannot be formalised because, usually, requirements are written in natural language. But it is a very important step: if we make mistakes in the initial specification then the following effort may be wasted! Many software projects have failed because requirements were incomplete, inconsistent, infeasible given the effort available, or misunderstood. Note that we are concerned with errors in the requirements themselves as well as with errors we make in modelling them. So we try in writing specifications to actively consider whether what we read seems sensible, complete and consistent. In creating a formal model we tend to come up with many questions, and generating these questions to ask of the people responsible for the requirements (the *customers*) has proved to be extremely beneficial in detecting problems at the start of the project. We try to be abstract, but that is not the same thing as being vague!

The main technique in validation is to check that each requirement is met. When we have written the initial specification we go back to the requirements and for each issue that we can find, we should conclude one of the following:

- It is met.
- It is not met, and we need to change the specification.
- It is not met because we think it is not a good idea (because of infeasibility, or for consistency with other parts, perhaps) and we need to discuss with the customers.
- It will be deferred to later in the development. This applies to "non-functional requirements" like the intended programming language or operating system, or performance requirements, but also to things that we have not yet designed, like aspects of the user interface or particular algorithms to be used. In this case we add it to a list of the requirements against which later development steps will be validated. We need, of course, to have in mind a development strategy that will allow such requirements to be met eventually.

There are also other validation techniques we can use:

- With experience, we can read the specification to look for properties that it will have that are not mentioned in the requirements. To take a trivial example, when we specify data storage, we naturally ask if it may become full, and if so what should happen. It may be that the user has not considered the possibility. Another example is whether a data structure should be initialised, and if so to what? This is typical of the kind of issue that may seem so obvious to the customers, who know the domain well, that they omitted to mention it. Scenarios, or use-cases, often lack essential but, to the customer, "obvious" steps. We should set up a formal procedure of queries to customers and their answers being documented.
- We should develop system tests (test cases and expected results) along with the specification. Doing this often helps to clarify the requirements, and these can also be shown to the customers, who will usually find them easier to read than the formal specification [16, 17].
- It is possible to rewrite the requirements from the specification. This is an expensive task, but generally produces requirements documents that are clearer, better structured, more concise, and more complete than the originals.
- We can prototype all or part of the system, perhaps by doing a quick and simplified refinement of the abstract types in it, and using the translators to SML or C++ (see Section 4.9) in the RAISE tools to run some test cases. We can also let the customers use it to get more feedback from them.

Providing early feedback to the customers in the form of queries, test cases, rewritten requirements, or prototypes has the added advantage of committing them to what has been done so far, and helps demonstrate to them the added cost and danger of later requirement changes, the bane of every software project manager's life! We try to make the initial specification a *contract* between us and the customers.

Verification is the check that we are developing the system correctly, so that the final implementation conforms to the initial specification. It must come after validation, since it assumes the correctness of the initial specification. We discuss it as part of the next section on refinement.

4.6 Refinement

We mentioned earlier that we develop by "invent and verify": we invent a more concrete version of a module and then verify that it is correct with respect to previous one. The formal relation that must exist between the two is the *refinement relation*, sometimes also called the *implementation relation*.

The refinement relation needs to be transitive: we want to develop, say, from A_0 to A_1 and then from A_1 to A_2, checking refinement at each step, and be assured that A_2 must refine A_0. Additionally, refinement needs to be monotonic with respect to building modules from other modules. Suppose module A is developed through version A_0 to the final A_2 as above, and module B has first version B_0 that instantiates A_0 and is developed (perhaps by other people) to B_1, say, that still instantiates A_0. Now we want to integrate the final versions. We write module B_2 that differs from B_1 only in substituting the identifier A_2 for the identifier A_0: see Figure 3. We want, provided A_2 refines A_0 and B_1 refines B_0, that B_2 should be guaranteed to refine B_1 and hence B_0. Monotonicity is what gives this guarantee. If this were not true we could not conveniently develop modules separately. Effectively A_0 is a *contract* between the developers of B and the developers of A: it says to the developers of B what A will provide, and to the developers of A what they must provide. Just how the latter group does this should be of no concern to the former.

The refinement relation should also hold in instantiations of parameterised of schemes: the class of each actual parameter should be a refinement of the class of the corresponding formal parameter.

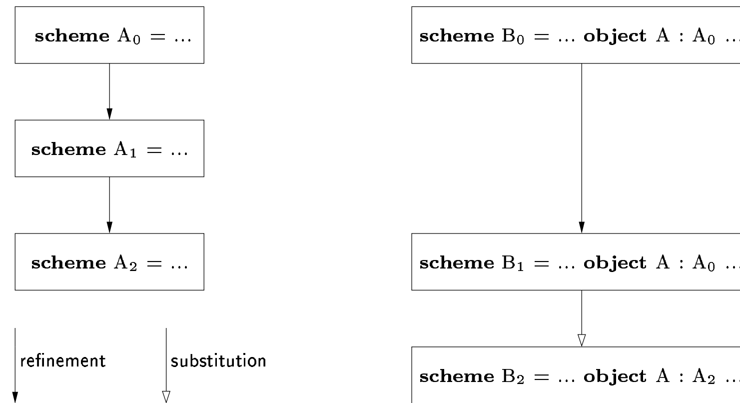

Figure 3: Separate development.

The formal definition of refinement can be found in the RAISE method book [5]. Here we give an intuition. It has two components. For A_1 to refine A_0 we require:

- The signature of A_1 must include the signature of A_0. That is, A_1 must contain all the entities (types, values, variables, channels, and objects) with the same names and the same maximal types or, for objects, with classes that are in the same relation. This relation, termed *static implementation*, was introduced earlier in Section 2.11. The relation is necessary for the monotonicity property: we need to be able to replace references to A_0 with references to A_1 in other modules without causing type or scope errors. The signature we are concerned with does not include hidden entities: these do not need to be included in refinements. The relation is also one of inclusion: A_1 may have more entities than A_0.
- All the *properties* of A_0 must hold in A_1. Properties may be expressed as axioms, but also include definitions of constants and functions, initial values of variables, and the restrictions in subtypes. Property preservation is clearly transitive.

The first of these conditions can be checked statically, and the RAISE tools do this as part of type checking. The second is not statically checkable, and in general requires proof for full verification. But the "R" in RAISE stands for "rigorous": the method allows for the conditions to be checked informally, by hand. The amount of proof we do will depend on how critical the system is, and how much budget we have. Proof is expensive because it involves considerable time and also skilled, experienced people to do it. It is unfortunately the case that the kinds of proofs that arise in software development are generally beyond the capabilities of automated proof tools.

4.7 Lightweight Formal Methods

It is possible to use formal methods without proof, and even without refinement: the initial specification is sufficient to explore the problem and provide a basis for implementation. Such use of a formal method is sometimes called "lightweight". It is found that most of the benefit of a formal method is in analysing and capturing requirements, in identifying and resolving requirements issues at the start of development, and in providing a sound basis for implementation. If the specification is not too complicated, implementation may be done directly from it.

There are some formal techniques that we can employ, that may or may not employ proof, that we can adopt to increase confidence in specifications: confidence conditions and theorems. We consider these in turn.

4.8 Confidence Conditions

Confidence conditions are conditions that should probably be true if the module is not to be inconsistent, but that cannot in general be determined as true or false by an automatic tool. The following conditions are generated by the RAISE tools:

1. Arguments of invocations of functions and operators are in subtypes, and, for partial functions and operators, preconditions are satisfied.
2. Values supposed to be in subtypes are in the subtypes. These are generated for
   - values in explicit value definitions;
   - values of explicit function definitions (for parameters in appropriate subtypes and satisfying any given preconditions);
   - initial values of variables;
   - values assigned to variables;
   - values output on channels.
3. Subtypes are not empty.
4. Values satisfying the restrictions exist for implicit value and function definitions.
5. The classes of actual scheme parameters implement the classes of the formal parameters.
6. For an implementation relation, the implementing class implements the implemented class. This gives a means of expanding such a relation or expression, by asserting the relation in a theory and then generating the confidence conditions for the theory.
7. A definition of a partial function without a precondition (which generates the confidence condition **false**).
8. A definition of a total function with a precondition (which generates the confidence condition **false**).

Examples of all the first 4 kinds of confidence conditions listed above are generated from the following intentionally peculiar scheme (in which line numbers have been inserted so that readers can relate the following confidence conditions to their source):

```
1 scheme CC =
2 class
3   value
4     x1 : Int = hd <..>,
5     x2 : Int = f1(-1),
6     x3 : Nat = -1,
7     f1 : Nat -~-> Nat
```

```
 8     f1(x) is -x
 9     pre x > 0
10   variable
11     v : Nat := -1
12   channel
13     c : Nat
14   value
15     g : Unit -> write v out c Unit
16     g() is v := -1 ; c!-1
17   type
18     None = {| i : Nat :- i < 0 |}
19   value
20     x4 : Nat :- x4 < 0,
21     f2 : Nat -> Nat
22     f2(n) as r post n + r = 0
23 end
```

This produces the following confidence conditions (which are all provably false). The first part of each condition is a reference to its source in the form `file:line:column`:

```
CC.rsl:4:19: CC:
-- application arguments and/or precondition
let x = <..> in x ~= <..> end

CC.rsl:5:18: CC:
-- application arguments and/or precondition
-1 >= 0 /\ let x = -1 in x > 0 end

CC.rsl:6:14: CC:
-- value in subtype
-1 >= 0

CC.rsl:8:5: CC:
-- function result in subtype
all x : Nat :- (x > 0 is true) => -x >= 0

CC.rsl:11:13: CC:
-- initial value in subtype
-1 >= 0

CC.rsl:16:17: CC:
-- assigned value in subtype
-1 >= 0

CC.rsl:16:24: CC:
-- output value in subtype
-1 >= 0

CC.rsl:18:26: CC:
-- subtype not empty
```

```
exists i : Nat :- i < 0

CC.rsl:20:8: CC:
-- possible value in subtype
exists x4 : Nat :- x4 < 0

CC.rsl:22:5: CC:
-- possible function result in subtype
all n : Nat :- exists r : Nat :- n + r = 0
```

It is usually sufficient to carefully *inspect* confidence conditions rather than trying to prove them. Most of the time it is easy to see that the conditions are OK, but they are a good way to find errors, particularly in the first category where we apply a function forgetting its precondition.

There is a danger when proving confidence conditions, since they can indicate an inconsistency in the module. For example, scheme `CC` above asserts through the definition of `x3` that -1 is in the type **Nat**. This is false, and so this definition implies the property **false**. `CC` is therefore inconsistent and anything can be proved about it. In particular, all the provably false confidence conditions above can also be proved true! So if we try to prove confidence conditions we must proceed with care.

4.8.1 Theorems

Theorems are formal statements about specifications that we state separately: they are intended to be consequences of the specifications, not part of their definitions. They can be proved, formally or by hand, or just examined carefully. Even when not proved they can be useful as part of the documentation.

Theorems can be stated in RAISE by means of a **theory** module. A theory takes the form:

theory name :
axiom
...
end

where ... is one or more axiom definitions. To support theories there are two extensions to the RAISE syntax that are useful:

- The *implementation relation* $\vdash C_1 \preceq C_0$, where `C1` and `C0` are class expressions. The implementation (or refinement) relation was described in Section 4.6.
- The *class scope expression* **in** C $\vdash$ expr, where `C` is a class expression and `expr` is a boolean expression which may reference entities defined in `C`.

Generating confidence conditions for an implementation relation will expand it into its constituent properties, allowing us to examine them without necessarily proving them, or perhaps only proving some.

Typically if we want to do proof we will concentrate on critical properties. For example, if there are system consistency properties that should always be maintained, we can formulate as theorems the property that

they are maintained by our generators (provided any preconditions hold). For example, suppose in scheme `A`, `gen` is a generator with a parameter of type `U`, `T` is the type of interest, and `can_gen` is a function expressing the precondition of `gen`, so the definition of `gen` looks like:

value
  gen : U × T $\stackrel{\sim}{\to}$ T
  gen(u, t) ≡ ...
  pre can_gen(u, t)

Then the theorem we would write is:

in A ⊢ ∀ u : U, t : T •
  consistent(t) ∧ can_gen(u, t) ⇒ consistent(gen(u, t))

Consistency conditions are a good choice for doing proofs. Generating the wrong result values of functions often shows up in testing, but creating inconsistencies in the system may not show up until some time after the inconsistency was created, and so it may be be hard to find them in testing and hard to identify when and how an inconsistency was originally generated.

Inclusion of checking consistency is also a good thing to include in test cases. But only with a proof can one be sure that a generator will never cause an inconsistency.

Finally, one should not forget the value of code reading by peers. This is a comparatively cheap and very effective means of discovering errors, and can be applied to specifications as much as to code. In fact it is generally easier to read specifications than programming language code. They are more abstract, and are intended to be read by people rather than machines.

4.9 Generating the Executable Program

The traditional development route is from RAISE to a programming language like C++ or Java. The RAISE tools available from UNU/IIST's web site `www.iist.unu.edu` include a translators from a subset of RSL to C++ and SML (though the latter is intended mainly for prototyping and testing). Parts of specifications may need to be translated to SQL, say, if part of the specification is intended to specify a database. The original RAISE tools [1] also include a translator to Ada. There is advice on translation by hand, including translation of concurrency, in the RAISE method book [5].

But there are many other possibilities. [18], for example, uses AWK as the implementation language.

5 When Not to Use RAISE

We do not mean to give the impression that RAISE or a similar software specification language should be used to define all software systems. There are exceptions, and we give some examples in this section.

5.1 There is a Special-Purpose Formalism

There are many special-purpose formalisms (sometimes with associated tools) that can sometimes be used in preference to a general purpose language like RSL. For example, BNF is a standard notation for defining grammars, and has associated tools like flex and bison for generating parsers and building abstract syntax trees. BNF is well defined, and provides a well-known, convenient, and compact notation. Copying this in RSL could be done, but the result would be less concise and still require the equivalent of flex and bison to be developed.

Real-time systems, ones which depend heavily on precise timing, such as real-time schedulers and process control systems, are often better analysed using a special-purpose formalism like Duration Calculus (DC) [19]. (There is some ongoing work to add real-time features to RSL [20, 21, 22].)

Another example is defining semantics of languages. There are notations like Structured Operational Semantics [23] that have their own compact notations that would be much less readable in RSL.

5.2 The Effort is Not Worth the Gain

Sometimes there is a language adapted to a particular kind of application that allows the implementation to be written at a level that is very close to how one would specify it. An example is the RAISE tools. These were written in a language Gentle [24] that is a high level language intended for use by compiler constructors. The RAISE tools were written in this language without writing a specification of them. The reason is that for the type checker, for example (the first tool written and a basis for all the others) it was felt that the scope and type rules could not have been written at a much more abstract level: the actual error messages, and some details about input and output to files, which were largely copied from another system, would have been almost the only things left more abstract. So in this case the executable program (Gentle is executable in that it translates to C) is also the specification. There is, of course, a definition of the semantics of RSL (using a special-purpose formalism) that includes the static semantics (the scope and type rules) but the tools were not developed with close reference to this (and RSL has also been extended): the tool developer had a very good working knowledge of RSL having worked on its original design.

Another, rather different, example is graphical user interfaces. The top level RAISE specification of a system defines the functions that may be accessed by users, which may be people or other software. In the case of people, graphical user interfaces are common, and there are many languages and tools to aid their construction. A main feature of such interfaces is that they are functionally simple. They help users select the function they want to invoke (often with menus or buttons), they ask for the necessary inputs to be provided (by selection or on forms) and they display or output results. The top level specification describes what functions are available, shows through their signatures what inputs are needed and what results will be returned, and defines what preconditions need to be checked. All that needs to be done is the design of the graphical part, the definition of helpful messages when preconditions are violated, and perhaps the design of convenient output formats for extensive result values. There seems in practice little point in trying to specify these aspects, especially the graphical, visual ones.

References

[1] Chris George and S. Prehn. The RAISE Tools Users Guide. LaCoS Report DOC/7, Computer Resources International A/S, 1992.

[2] D.L. Chalmers, B. Dandanell, J. Gørtz, J. Storbank Pedersen, and E. Zierau. Using RAISE — First Impressions From a LaCoS User Trial. In *Proceedings of VDM '91*, volume 551 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[3] R.E. Milne. The Formal Basis for the RAISE Specification Language. In *Semantics of Specification Languages*, Workshops in Computing. Springer-Verlag, 1993.

[4] D. Bolignano and M. Debabi. On the Semantic Foundations of RSL: a Concurrent, Functional and Imperative Specification Language. In *Proceedings of FORTE '93*. Boston University, 1993.

[5] The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995. Available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book.

[6] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992. Available from Terma A/S. Contact jnp@terma.com.

[7] Chris George. RAISE Tools User Guide. Technical Report 227, UNU/IIST, P.O. Box 3058, Macau, February 2001.

[8] Cliff B. Jones and Kees Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[9] J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. Woodcock, editors, *Proceedings of the Third Refinement Workshop*. Springer-Verlag, 1990.

[10] C.B. Jones and R.C.F. Shaw. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[11] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[12] Peter D. Mosses. CASL: A Guided Tour of its Design. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, Proc. 13th International Workshop, WADT '98, Lisbon, 1998, Selected Papers*, volume 1589 of *LNCS*, pages 216–240. Springer-Verlag, 1999.

[13] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[14] Jean-Raymond Abrial. *The B-Book*. Cambridge University Press, 1996.

[15] Do Tien Dung, Chris George, Hoang Xuan Huan, and Phung Phuong Nam. A Financial Information System. Technical Report 115, UNU/IIST, P.O.Box 3058, Macau, July 1997. Partly published in *Requirements Targeting Software and Systems Engineering*, LNCS 1526, Springer-Verlag, 1998.

[16] Bernhard K. Aichernig. Test-design through abstraction, a systematic approach based on the refinement calculus. *Journal of Universal Computer Science (J.UCS)*, 7(8), 2001.

[17] Johann Hörl and Bernhard K. Aichernig. Validating voice communication requirements using lightweight formal methods. *IEEE Software*, pages 21–27, May/June 2000.

[18] Balkhis Abu Bakar and Tomasz Janowski. Automated Result Verification with AWK. Technical Report 205, UNU/IIST, P.O. Box 3058, Macau, June 2000. Presented at and published in the proceedings of the 6th IEEE International Conference on Engineering of Complex Computer Systems, Tokyo, Japan, September 2000, IEEE Computer Society Press.

[19] Zhou ChaoChen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

[20] Chris George and Xia Yong. An Operational Semantics for Timed RAISE. Technical Report 149, UNU/IIST, P.O.Box 3058, Macau, November 1998. Presented at and published in the proceedings of FM'99, Toulouse, France, 20–24 September 1999, LNCS 1709, Springer-Verlag, 1999, pp. 1008–1027.

[21] Li Li and He Jifeng. Towards a Denotational Semantics of Timed RSL using Duration Calculus. Technical Report 161, UNU/IIST, P.O.Box 3058, Macau, April 1999. Publication by Chinese Journal of Advanced Software Research in 2000.

[22] Li Li and He Jifeng. A Denotational Semantics of Timed RSL using Duration Calculus. Technical Report 168, UNU/IIST, P.O.Box 3058, Macau, July 1999. Presented at and published in the proceedings of The Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99), part of the federated 1999 International Computer Congress, December 13 - 15, Hong Kong, IEEE Computer Society Press, 1999, pp. 492–503.

[23] G.D. Plotkin. A structured approach to operational semantics. Technical report, University of Edinburgh, 1981.

[24] Friedrich Wilhelm Schroër. *The GENTLE Compiler Construction System.* R. Oldenbourg, 1997. Available from http://www.first.gmd.de/gentle/.

A RSL Syntax

A.1 Conventions

X Y	X followed by Y
X	Y	X or Y
[X]	X or empty
{X}	a possibly empty sequence of X's

A.2 Changes from RSL Book

with_class_expr added
forall removed
“ $\overset{\sim}{\underset{m}{\rightarrow}}$ ” added
prefix “−” and “+” added; infix “==” added
comments allowed wherever white space allowed
schemes cannot be embedded
test_case_decl added

A.3 Modules and Declarations

module ::=
  [file_id {“,” file_id}] **scheme** scheme_def
| [file_id {“,” file_id}] **object** object_def
| [file_id {“,” file_id}] **theory** theory_def
| [file_id {“,” file_id}] **devt_relation** devt_relation_def

decl ::=
<table>
<tr><td> object object_def {“,” object_def}</td><td align="right">object_decl</td></tr>
<tr><td>| type type_def {“,” type_def}</td><td align="right">type_decl</td></tr>
<tr><td>| value value_def {“,” value_def}</td><td align="right">value_decl</td></tr>
<tr><td>| variable variable_def {“,” variable_def}</td><td align="right">variable_decl</td></tr>
<tr><td>| channel channel_def {“,” channel_def}</td><td align="right">channel_decl</td></tr>
<tr><td>| axiom axiom_def {“,” axiom_def}</td><td align="right">axiom_decl</td></tr>
<tr><td>| test_case test_case_def {“,” test_case_def}</td><td align="right">test_case_decl</td></tr>
</table>

scheme_def ::= id [“(” object_def {“,” object_def} “)”] “=” class_expr

object_def ::= id [“[” typing {“,” typing} “]”] “:” class_expr

type_def ::=
<table>
<tr><td> id</td><td align="right">sort_def</td></tr>
<tr><td>| id “==” variant {“|” variant}</td><td align="right">variant_def</td></tr>
<tr><td>| id “=” (type-name | “_”) “|” (type-name | “_”) {“|” (type-name | “_”)}</td><td align="right">union_def</td></tr>
<tr><td>| id “::” component_kind {component_kind}</td><td align="right">short_record_def</td></tr>
<tr><td>| id “=” type_expr</td><td align="right">abbreviation_def</td></tr>
</table>

variant ::= (id_or_op | "_") ["(" component_kind {"," component_kind} ")"]

component_kind ::= [id_or_op ":"] type_expr ["↔" id_or_op]

value_def ::=
typing
| single_typing "=" *pure*-value_expr — explicit_value_def
| single_typing *pure*-restriction — implicit_value_def
| single_typing formal_function_application ≡ value_expr [pre_condition] — explicit_function_def
| single_typing formal_function_application post_condition [pre_condition] — implicit_function_def

formal_function_application ::=
value-id "(" [binding {"," binding}] ")" {"(" [binding {"," binding}] ")"} — id_application
| prefix_op id — prefix_application
| id infix_op id — infix_application

variable_def ::=
id ":" type_expr [":=" *pure*-value_expr] — single_variable_def
| id "," id {"," id} ":" type_expr — multiple_variable_def

channel_def ::= id {"," id} ":" type_expr

axiom_def ::= ["[" id "]"] *readonly_logical*-value_expr

test_case_def ::= ["[" id "]"] value_expr

A.4 Class Expressions

class_expr ::=
class {decl} **end** — basic_class_expr
| **extend** class_expr **with** class_expr — extending_class_expr
| **hide** defined_item {"," defined_item} **in** class_expr — hiding_class_expr
| **use** rename_pair {"," rename_pair} **in** class_expr — renaming_class_expr
| **with** *element*-object_expr {"," *element*-object_expr} **in** class_expr — with_class_expr
| *scheme*-name ["(" object_expr {"," object_expr} ")"] — scheme_instantiation

rename_pair ::= defined_item **for** defined_item

defined_item ::= id_or_op [":" type_expr]

A.5 Object Expressions

object_expr ::=
object-name
| *array*-object_expr "[" *pure*-value_expr {"," *pure*-value_expr} "]" — element_object_expr

| "[|" typing {"," typing} "•" *element*-object_expr "|]" — array_object_expr
| object_expr "{" rename_pair {"," rename_pair} "}" — fitting_object_expr

A.6 Type Expressions

type_expr ::=
  Unit | **Bool** | **Int** | **Nat** | **Real** | **Text** | **Char** — type_literal
| *type*-name
| type_expr "×" type_expr {"×" type_expr} — product_type_expr
| type_expr"**-set**" | type_expr"**-infset**" — set_type_expr
| type_expr"*" | type_expr"ω" — list_type_expr
| type_expr (" $\overset{\sim}{\underset{m}{\to}}$ " | " $\underset{m}{\to}$ ") type_expr — map_type_expr
| type_expr ("$\overset{\sim}{\to}$" | "$\to$") {access_desc} type_expr — function_type_expr
| "{|" single_typing *pure*-restriction "|}" — subtype_expr
| "(" type_expr ")" — bracketed_type_expr

access_desc ::= access_mode access {"," access}

access_mode ::= **read** | **write** | **in** | **out**

access ::=
  variable_or_channel-name
| "{" [access {"," access}] "}" — enumerated_access
| [qualification] **any** — completed_access
| "{" access | *pure*-set_limitation "}" — comprehended_access

A.7 Value Expressions

value_expr ::=
  value_literal
| *value_or_variable*-name
| *variable*-name "`" — pre_name
| **chaos** | **skip** | **stop** | **swap** — basic_expr
| "(" value_expr "," value_expr {"," value_expr} ")" — product_expr
| set_expr
| list_expr
| map_expr
| "λ" lambda_parameter "•" value_expr — function_expr
| *list_or_map_or_function*-value_expr
  "(" [value_expr {"," value_expr}] ")" {"(" [value_expr {"," value_expr}] ")"} — application_expr
| ("∀" | "∃" | "∃!") typing {"," typing} restriction — quantified_expr
| value_expr ≡ value_expr [pre_condition] — equivalence_expr
| value_expr post_condition [pre_condition] — post_expr
| value_expr ":" type_expr — disambiguation_expr
| "(" value_expr ")" — bracketed_expr
| infix_expr

| prefix_expr	
| (“□” | “⊓” | “‖”) “{” value_expr “|” set_limitation “}”	comprehended_expr
| [qualification] **initialise**	initialise_expr
| *variable*-name “:=” value_expr	assignment_expr
| *channel*-name “?”	input_expr
| *channel*-name “!” value_expr	output_expr
| structured_expr	

structured_expr ::=

local {decl} **in** value_expr **end**	local_expr
| **let** let_def {“,” let_def} **in** value_expr **end**	let_expr
| **if** *logical*-value_expr **then** value_expr {**elsif** *logical*-value_expr **then** value_expr} [**else** value_expr] **end**	if_expr
| **case** value_expr **of** pattern “→” value_expr {“,” pattern “→” value_expr} **end**	case_expr
| **while** *logical*-value_expr **do** *unit*-value_expr **end**	while_expr
| **do** *unit*-value_expr **until** *logical*-value_expr **end**	until_expr
| **for** list_limitation **do** *unit*-value_expr **end**	for_expr

value_literal ::=

“(” “)”	unit_literal
| **true** | **false**	bool_literal
| int_literal	
| real_literal	
| text_literal	
| char_literal	

set_expr ::=

“{” *readonly_integer*-value_expr “..” *readonly_integer*-value_expr “}”	ranged_set_expr
| “{” [*readonly*-value_expr {“,” *readonly*-value_expr}] “}”	enumerated_set_expr
| “{” *readonly*-value_expr “|” set_limitation “}”	comprehended_set_expr

list_expr ::=

“⟨” *integer*-value_expr “..” *integer*-value_expr “⟩”	ranged_list_expr
| “⟨” [value_expr {“,” value_expr}] “⟩”	enumerated_list_expr
| “⟨” value_expr “|” list_limitation “⟩”	comprehended_list_expr

set_limitation ::= typing {“,” typing} [restriction]

list_limitation ::= binding **in** *readonly_list*-value_expr [restriction]

restriction ::= “•” *readonly_logical*-value_expr

map_expr ::=

“[” [value_expr_pair {“,” value_expr_pair}] “]”	enumerated_map_expr
| “[” value_expr_pair “|” set_limitation “]”	comprehended_map_expr

value_expr_pair ::= *readonly*-value_expr “↦” *readonly*-value_expr

lambda_parameter ::= “(” [typing {“,” typing}] “)” | single_typing

pre_condition ::= **pre** *readonly_logical*-value_expr

post_condition ::= [**as** binding] **post** *readonly_logical*-value_expr

infix_expr ::=

Production	Name
value_expr (“▯” | “⊓” | “‖” | “╫” | “;”) value_expr	stmt_infix_expr
| *logical*-value_expr (“⇒” | “∨” | “∧”) *logical*-value_expr	axiom_infix_expr
| value_expr infix_op value_expr	value_infix_expr

prefix_expr ::=

Production	Name
“∼” *logical*-value_expr	axiom_prefix_expr
| “□” *readonly_logical*-value_expr	universal_prefix_expr
| prefix_op value_expr	value_prefix_expr

let_def ::=

Production	Name
typing	
| let_binding “=” value_expr	explicit_let
| single_typing restriction	implicit_let

let_binding ::=

Production	Name
binding	
| *pure_value*-name “(” inner_pattern {“,” inner_pattern} “)”	record_pattern
| “⟨” [inner_pattern {“,” inner_pattern}] “⟩” [“⌢” inner_pattern]	list_pattern

A.8 Bindings

binding ::= id_or_op | “(” binding “,” binding {“,” binding} “)”

A.9 Typings

typing ::= binding {“,” binding} “:” type_expr

single_typing ::= binding “:” type_expr

A.10 Patterns

pattern ::=

Production	Name
value_literal	
| *pure_value*-name	
| “_”	wildcard_pattern
| “(” inner_pattern “,” inner_pattern {“,” inner_pattern} “)”	product_pattern
| *pure_value*-name “(” inner_pattern {“,” inner_pattern} “)”	record_pattern
| “⟨” [inner_pattern {“,” inner_pattern}] “⟩” [“⌢” inner_pattern]	list_pattern

inner_pattern ::=
  value_literal
| id_or_op
| "_"   wildcard_pattern
| "(" inner_pattern "," inner_pattern {"," inner_pattern} ")"   product_pattern
| *pure_value*-name "(" inner_pattern {"," inner_pattern} ")"   record_pattern
| "⟨" [inner_pattern {"," inner_pattern}] "⟩" ["⌢" inner_pattern]   list_pattern
| "=" *pure_value*-name   equality_pattern

A.11 Names

name ::= [qualification] (id | "(" op ")")

qualification ::= *element*-object_expr "."

A.12 Identifiers and Operators

id_or_op ::= id | op

op ::= infix_op | prefix_op

infix_op ::=
  "=" | "≠" | "==" | ">" | "<" | "≥" | "≤" | "⊃" | "⊂" | "⊇" | "⊆" | "∈" | "∉"
| "+" | "−" | "\" | "⌢" | "∪" | "†" | "∗" | "/" | "°" | "∩" | "↑"

prefix_op ::= "−" | "+" | **abs** | **int** | **real** | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**

A.13 Theories and Development Relations

theory_def ::= id ":" **axiom** [theory_axiom {"," theory_axiom}] **end**

devt_relation_def ::= id "(" id **for** id ")" ":" theory_expr

theory_axiom ::= ["[" id "]"] theory_expr

theory_expr ::=
  in class_expr "⊢" theory_expr   class_scope_expr
| "⊢" class_expr "⪯" class_expr   implementation_relation
| "⊢" *element*-object_expr "⊑" *element*-object_expr   implementation_expr
| *readonly_logical*-value_expr
| "(" theory_expr ")"

B Symbols and Keywords

B.1 ASCII Versions of RSL Symbols

<table>
<tr><th>Sym</th><th>ASCII</th><th>Sym</th><th>ASCII</th><th>Sym</th><th>ASCII</th></tr>
<tr><td>$\times$</td><td>><</td><td>*</td><td>-list</td><td>$^\omega$</td><td>-inflist</td></tr>
<tr><td>$\rightarrow$</td><td>-></td><td>$\stackrel{\sim}{\rightarrow}$</td><td>-~-></td><td>$\xrightarrow[m]{}$</td><td>-m-></td></tr>
<tr><td>$\xrightarrow[m]{\sim}$</td><td>-~m-></td><td>$\leftrightarrow$</td><td><-></td><td></td><td></td></tr>
<tr><td>$\wedge$</td><td>/\</td><td>$\vee$</td><td>\/</td><td>$\Rightarrow$</td><td>=></td></tr>
<tr><td>$\forall$</td><td>all</td><td>$\exists$</td><td>exists</td><td>$\bullet$</td><td>:-</td></tr>
<tr><td>$\Box$</td><td>always</td><td>$\equiv$</td><td>is</td><td>$\neq$</td><td>~=</td></tr>
<tr><td>$\leq$</td><td><=</td><td>$\geq$</td><td>>=</td><td>$\uparrow$</td><td>**</td></tr>
<tr><td>$\in$</td><td>isin</td><td>$\notin$</td><td>~isin</td><td>$\subset$</td><td><<</td></tr>
<tr><td>$\subseteq$</td><td><<=</td><td>$\supset$</td><td>>></td><td>$\supseteq$</td><td>>>=</td></tr>
<tr><td>$\cup$</td><td>union</td><td>$\cap$</td><td>inter</td><td>$\dagger$</td><td>!!</td></tr>
<tr><td>$\langle$</td><td><.</td><td>$\rangle$</td><td>.></td><td>$\mapsto$</td><td>+></td></tr>
<tr><td>$\|$</td><td>||</td><td>╫</td><td>++</td><td>⌷</td><td>|=|</td></tr>
<tr><td>$\sqcap$</td><td>|^|</td><td>λ</td><td>-\</td><td>$\circ$</td><td>#</td></tr>
<tr><td>$\vdash$</td><td>|-</td><td>$\preceq$</td><td>{=</td><td>$\sqsubseteq$</td><td>[=</td></tr>
</table>

B.2 RSL Keywords

Bool	**class**	**initialise**	**stop**
Char	**do**	**int**	**swap**
Int	**dom**	**len**	**test_case**
Nat	**elems**	**let**	**then**
Real	**else**	**local**	**tl**
Text	**elsif**	**object**	**true**
Unit	**end**	**of**	**type**
abs	**extend**	**out**	**until**
any	**false**	**post**	**use**
as	**for**	**pre**	**value**
axiom	**hd**	**read**	**variable**
card	**hide**	**real**	**while**
case	**if**	**rng**	**with**
channel	**in**	**scheme**	**write**
chaos	**inds**	**skip**	

theory and **devt_relation** are also keywords for theories and development relations respectively.