The United Nations University

UNU/IIST

International Institute for Software Technology

A RAISE tutorial

Chris George

December 1998

UNU/IIST Report No. 153

UNU/IIST and UNU/IIST Reports

UNU/IIST is a Research and Training Center of the United Nations University. It was founded in 1992, and is located in Macau. UNU/IIST is jointly funded by the Governor of Macau and the Governments of China and Portugal through contribution to the UNU Endowment Fund.

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. courses which typically teach advanced software development techniques,
5. events in which conferences and workshops are organised or supported by UNU/IIST, and
6. 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 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 – 31.7.2001

The United Nations
University

UNU/IIST

**International Institute for
Software Technology**

P.O. Box 3058
Macau

A RAISE tutorial

Chris George

Abstract

RAISE includes a wide-spectrum specification language (RSL), a method for the rigorous development of software systems, and a set of accompanying tools. This tutorial presents an introduction to the language and method. It also introduces a recently proposed extension to RAISE to allow the specification and development of real-time systems.

Chris George is a Senior Research Fellow at UNU/IIST, 1 September 1994 — 31 August 1999. 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 © 1999 by UNU/IIST, Chris George

Contents

1 The RAISE background

RAISE — Rigorous Approach to Industrial Software Engineering — was an ESPRIT project running from 1985 to 1990 and consuming about 120 person years of effort. Dansk Datamatik Center (later Computer Resources International) was the main contractor. STC Technology (now BNR Europe) was the second partner producing the RAISE technology. Nordisk Brown Boveri (now SYPRO) and part of ICL (now owned by Fujitsu) were involved mainly as industrial trialists. The aim of RAISE was to produce a method for the rigorous development of software, based on a wide spectrum specification language, with accompanying tools and technology transfer material.

The RAISE project was succeeded by another ESPRIT project — LaCoS — Large scale Correct Systems using formal methods — which ran from 1990 to 1995. The project was similar in size to the RAISE project but involved nine industrial partners in seven European countries. Most of the partners were *users* trying RAISE on a number of different projects. The others were *producers* developing further the method and tools. There was also a possibility of revising RSL in the light of user experience, but in practice almost no change was found to be necessary. A series of experience reports [1, 2] describe the results of the industrial applications of this project. Most of the experiences were positive, and were reflected in the tool and method development.

The main inspiration at the start of the RAISE project was VDM [3, 4], which was seen as having two major deficiencies. It lacked modularity and it could not deal with concurrency.

There was also what was then seen as a completely different approach, the algebraic. This differed from the model-based approach of VDM and Z [5] in terms of how things were specified (implicitly in terms of signatures and axioms) but also in the way in which the specification language was given a semantics. It was not at all clear how the two could be combined. Doing so was a major achievement of the RAISE Specification Language — RSL.

The modularity in RSL is largely inspired by the algebraic languages (CLEAR [6], OBJ [7, 8], Larch [9], etc.). Concurrency is based on process algebras — close to CSP [10] and CCS [11], but with value passing [12]. RSL added a new operator, "interlock", that enabled the axiomatic specification of concurrency.

And, of course, it was high time there were some decent tools!

RSL is a "wide spectrum" language. This means that it has features allowing its use for very abstract, initial specifications and also for more concrete developments of initial specifications that can be easily (or even automatically) translated into a programming language.

We originally wanted a wide spectrum language so that we could stay within one language, and hence within one semantic framework, at all development levels. In fact, it turns out that the ability to mix styles at the same development level, and even within the same module, is very useful.

This tutorial does not attempt to be comprehensive. It aims to introduce the important aspects of RSL (fully described in the book *The RAISE Specification Language* [13]) and the main ideas of the RAISE method (described in the later book *The RAISE Development Method* [14]). Finally, it introduces briefly an extension to RAISE currently being defined to allow it to deal with real-time systems. The tutorial is structured round an example chosen to illustrate the main notions.

2 The example

We describe a system intended to model the transport of messages. The requirements are:

- Messages can be inserted and extracted.
- There may be some delay between a message being inserted and it being available for extraction.
- The extraction order should be the same as the insertion order, except that there should be some possibility of higher priority messages "overtaking" lower priority ones.
- It is not necessary to guarantee that the next message extracted is the highest priority one in the system. This is ideal, but may not always be possible.

We will keep the "messages" completely abstract, and only assume that a "priority", for simplicity a natural number, is part of a message. Higher numbers give higher priorities.

We will model the system in terms of two buffers, one holding messages "in transit" and one holding messages that have "arrived" and are waiting to be extracted. Transfer of messages between the two will be essentially non-deterministic.

The "transit" buffer will be modelled as a FIFO buffer, a queue, and the "arrived" buffer as a priority queue.

3 The RAISE method

Before we return to the example we give an overview of the method. This suggests writing the initial specification in an applicative sequential (functional) style, at as abstract a level as one is comfortable with, and doing the initial development within this style. The initial development typically includes developing (refining) the abstract types into concrete ones, which may involve defining more modules for new (perhaps abstract) types used to construct the concrete types. As the types are made more concrete, so the functions over them can be given more concrete definitions, and axioms gradually disappear.

It is typical that implementations are to be written in imperative programming languages, involving programming variables, assignment, sequences and loops. But we can move from applicative sequential RSL to imperative sequential RSL by means of a standard transformation. This introduces variables, assignments and sequencing, and allows the introduction of loops where appropriate or necessary (such as for translating quantifiers).

A further transformation is possible from imperative sequential to imperative concurrent specifications, if a concurrent or distributed implementation is required.

Since the applicative concurrent style is rarely used, we use the terms *applicative* for applicative sequential, *imperative* for imperative sequential and *concurrent* for imperative concurrent.

These transformations are illustrated in figure 1.

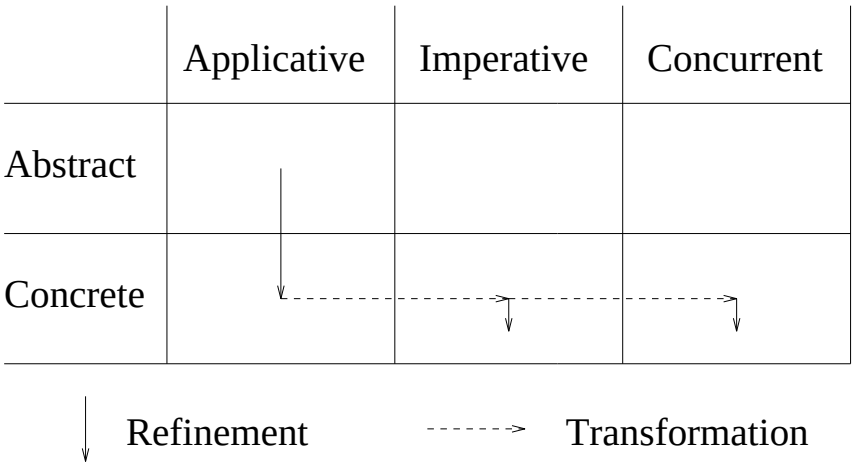

Figure 1: The standard RAISE development route

There are several points to note about this method:

- The transformations are standard (and could be largely automated). Even if done by hand they are easily checked as part of quality control.
- The blank areas in figure 1 represent styles of RSL that are possible but, in practice, more difficult to write. The method avoids them.
- The transformed specifications are correct *by construction*; we only need to check that the transformations are properly applied. There is a theorem, described fully in the method book [14], that shows that a transformed specification is a correct development of the original applicative abstract one (provided the concrete applicative one is a correct refinement

of the abstract applicative one).

There are two reasons for this approach:

1. The abstract imperative and concurrent specification styles were found hard to use in practice.
2. Refinement proofs are mainly limited to the applicative style. It is our experience that proving effectively the same property about specifications in the three styles is easiest in the applicative style. It is something like twice as difficult in the imperative one, and several times more difficult in the concurrent one. This disparity might be alleviated with more automated proof strategies, but we believe it will always hold to some extent.

4 The initial specification

The initial specification should try to capture the important properties of the system. If the system is small, like this one, it may be possible to do this in a single module. Otherwise, we may have to initially model the system in terms of its major components, and specify the properties of these components.

In a typical development there are a number of types and perhaps some functions over them that will be used in many components. Typical examples are identifiers and simple data structures like dates. Such types are not intended to become the state types of the imperative objects of the final system. It is convenient to place these in one or more modules that are made generally available to the other modules in the system. Such global sharing is in general not a good idea, and it is possible to define such types in modules which are then shared through parameterisation. But the number of parameters of modules can as a result grow quite alarmingly. So the method suggests that such types and their associated functions are made globally available and so can be freely referenced. Obviously, in a large development, careful control needs to be applied to changing such modules, but the advantages seem to outweigh the disadvantages.

For our system, we see the type of messages, and the notion of message priority, being candidates for such a global module and we define a scheme TYPES (figure 2).

TYPES illustrates the basic components of modules. They are built from *class expressions* of which the simplest is the *basic* class expression **class** ... **end** with *declarations* within. Here we have a **type** declaration and a **value** declaration. The type declaration defines a type *Message*. This is an abstract type, or *sort*: it is not defined in terms of any other types.

Values are introduced by giving their names and types. *priority* is a function value: it takes *Message* values as parameters and produces **Nat** results. Such a function obviously represents an attribute of its parameter type. **Nat**, as indicated by its bold face, is built-in to RSL. It is

```
scheme
  TYPES =
    class
      type Message

      value
        priority : Message → Nat,

        leq : Message × Message → Bool
        leq(m1, m2) ≡ priority(m1) ≤ priority(m2)
    end
```

Figure 2: The scheme TYPES

the type of natural numbers (the non-negative integers). Built-in types like **Nat** have associated operators like + and ≤.

The value *leq* is another function representing a relation on messages: it takes a pair of messages and returns a **Bool**ean. **Bool** is another built-in type, containing the values **true** and **false**. It has associated connectives ∧ (conjunction), ∨ (disjunction), ⇒ (implication), and ∼ (negation). *leq* is given a concrete definition: the ordering on messages is the corresponding ordering on their priorities.

There are two kinds of modules in RSL, **scheme**s and **object**s. A scheme is a named class expression. A class expression denotes its class of models, essentially all the things that could be used as implementations of it. A possible model of a message is a pair of a text string and a priority. Another possible model is a triple of a destination, text, and priority. And so on. An object denotes a particular model.

We want to share the notion of message and priority between other modules. So they cannot just share the scheme TYPES as it has many models. We need to share the same model. We do this by creating the object T (figure 3) from the scheme TYPES.

```
object
  T : TYPES
```

Figure 3: The object T

We define an object by giving a name and a class: TYPES is the name for its class.

There is an analogy between objects/classes and values/types, and we write "T : TYPES" just as we might write "i : Int". But objects in RSL are not the same as values, and classes are not the same as types.

Since the type *Message* in TYPES is abstract, we do not know which model T represents. But

this does not matter: we can refine the type *Message* later. All we need to know for now is that all modules referring to T will refer to the same model.

Now we can write the initial specification.

Initial specifications are typically abstract: the properties are described in terms of axioms rather than constructively in terms of a concrete model. The initial specification for our system is A_MESSAGE0 (figure 4). We use a conventional prefix "A_" in the name A_MESSAGE0 to indicate an applicative module, since there will be imperative and perhaps concurrent versions developed later. We also use a conventional suffix "0" since we expect to make a more concrete version "1".

In A_MESSAGE0, we refer to the type *Message* defined in the object T as *T.Message*. The type of the second parameter of *buffered*, *T.Message**, is the type of finite lists, or sequences, of values of type *T.Message*. "*" is one of the type constructors of RSL. Others are products, or tuples (as used in the result type of *get*); sets; and maps. Each comes with syntax for creating values, like "(x,y)" for a pair (2-tuple), $\langle\rangle$ for the empty list, $\langle$x,y$\rangle$ for a list of two values. There are also associated operators like $\cup$ (set union), $\frown$ (list concatenation), **hd** and **tl** for the head and tail of a (non-empty) list.

The type of *get* uses $\xrightarrow{\sim}$, which indicates that it is a *partial* function. That is, it might not be defined for all parameter values. We expect *get* not to be defined when *can_get* is false.

In making *put* a total function, we are assuming our system has unlimited capacity. A more realistic version would include a *can_put* function, initially underspecified, that we would later define in terms of maximum capacities of the queues involved. A capacity of a queue could be defined either in terms of the numbers of messages or in terms of storage consumption. For the latter we would need a size attribute of messages.

We model the state of the system by defining an abstract type *Buffer*. Values of type *Buffer* will depend on the sequence of messages input and not yet extracted. But we expect this dependency to be nondeterministic. Messages may be in transit and so not yet ready for extraction, so it may not be the case that the highest priority message input will be the next one extracted. Conversely, we do not necessarily expect to be able to discover from the state of the buffer what the actual input order of messages was. Suppose, for example, that messages *m1* and *m2* have both arrived and are ready for extraction, with *m1* of higher priority than *m2*. We cannot tell which of these messages was input first unless we add extra information not in the requirements, like including the time of sending of messages. We therefore model the connection between the input sequence of messages and the buffer state not as a function from one to the other, but as a relation, expressed by the function *buffered*. This is introduced, like the other three functions in the first value declaration of A_MESSAGE0, just by giving the name and the type.

There are a number of properties that our system must have in order to meet its requirements. These are expressed in the **axiom** declaration.

```
scheme
  A_MESSAGE0 =
    hide buffered, permutation, count in
      class
        type Buffer

        value
          put : T.Message × Buffer → Buffer,
          get : Buffer ~→ T.Message × Buffer,
          can_get : Buffer → Bool,
          buffered : Buffer × T.Message* → Bool

        axiom
          [ can_get_ax ] ∀ buff : Buffer • buffered(buff, ⟨⟩) ⇒ ∼ can_get(buff),

          [ buffered_put ]
            ∀ buff, buff′ : Buffer, l : T.Message*, m : T.Message •
              buffered(buff, l) ∧ put(m, buff) = buff′ ⇒ buffered(buff′, l ⌢ ⟨m⟩),

          [ buffered_get ]
            ∀ buff, buff′ : Buffer, l : T.Message*, m1, m2 : T.Message •
              buffered(buff, l) ∧ can_get(buff) ∧ get(buff) = (m1, buff′) ⇒
              (
                ∃ l1, l2 : T.Message* •
                  l = l1 ⌢ ⟨m1⟩ ⌢ l2 ∧
                  buffered(buff′, l1 ⌢ l2) ∧ (m2 ∈ elems l1 ⇒ ∼ T.leq(m1, m2))
              ),

          [ no_loss_or_gain ]
            ∀ buff : Buffer, l1, l2 : T.Message* •
              buffered(buff, l1) ∧ buffered(buff, l2) ⇒ permutation(l1, l2)

        value
          permutation : T.Message* × T.Message* → Bool
          permutation(l1, l2) ≡ (∀ m : T.Message • count(m, l1) = count(m, l2)),

          count : T.Message × T.Message* → Nat
          count(m, l) ≡ card { i | i : Nat • i ∈ inds l ∧ l(i) = m }
      end
```

Figure 4: The scheme A_MESSAGE0

Each axiom consists of an (optional) name in square brackets, followed by a Boolean expression, a predicate.

can_get_ax asserts that if the input list of messages is empty, the buffer must be empty (i.e.

nothing can be extracted from it). The converse is not necessarily true, as input messages may be in transit and not available for extraction.

buffered_put expresses the properties of a *put*. The new state is the buffering of the message appended to the end of an input list of the previous state.

buffered_get expresses the properties of a *get*. If a message is extracted it must have been input, the rest of the messages are retained, and any messages overtaken must have lower priority. **elems** is a built-in operator returning the set of elements in a list. $\in$ is set membership.

no_loss_or_gain expresses the property that messages are not lost or invented: for any given buffer state the collection of messages is fixed.

The specification is very loose in that it allows a range of implementations. At one extreme, the internal state might be just a FIFO queue, and there would be no overtaking by higher priority messages. At the other extreme, the internal state might be just a priority queue, with the highest priority message always the next one to be extracted. Our implementation will be between these two extremes. This reflects the requirements.

Since messages might be duplicates (as the requirements did not prohibit this) we have to be careful about defining the "collection" of messages. We use two extra functions to express this notion, *permutation* and *count*. **inds** is the set of index values of a list. For example, the list $\langle$x, y$\rangle$ has index values 1 and 2. Applying the list (as if it were a function) to the index value 1 returns x, the first element. And so on. **card**inality is the number of elements in a set. The argument of **card** is a *comprehended* set, in this case the set of index values i of the list l for which applying l to i gives the message m.

All types, even sorts, have equality and inequality defined automatically. This enables us to write equalities between, for example, pairs of *Message* and *Buffer* as in the *buffered_get* axiom. Equalities between tuples are defined pointwise.

There are in fact two kinds of equality in RSL: $=$ and $\equiv$. $=$ just compares values, and is essentially the equality found in programming languages. $\equiv$, as used in function definitions like that of *leq* in TYPES, is the semantic equivalence between expressions. As long as we avoid problems of undefinedness, the two are the same for applicative specifications. When we come to imperative and concurrent specifications we shall see that expressions may have *effects*, such as assigning to variables, and equivalent expressions must have the same effects as well as returning the same values, while expressions are equal if they return the same values.

Unlike many languages, RSL has no "define-before-use" restriction. *permutation*, for example, is used before it is defined. This gives useful flexibility, and modules are commonly written "top-down" as here, with details coming later.

The functions *buffered*, *permutation*, and *count* are just used to express abstract properties. We do not intend to implement them in the final system and so we **hide** them.

But before proceeding from the initial, abstract specification to a more concrete one we must *validate* the specification against the requirements. We go back to the requirements and check carefully that each requirement is either satisfied, or we have a development route in mind from the initial specification that can make it so. There are usually many requirements that should be deferred because they introduce detail that we are still leaving abstract. In a more realistic system there might be requirements about the maximum size of messages, the maximum rate at which the system should be able to deal with them, the programming language to be used, the machine architecture, the operating system, and so on.

It is almost always the case that writing the initial specification generates lots of questions about the requirements. Can there by duplicated messages? Are priorities linearly ordered? Are there limits on the buffer size? Writing specifications tends to find the inconsistencies and omissions which requirements documents in natural languages are typically full of.

This specification A_MESSAGE0 may well look rather complicated. It is not in general very easy to find such specifications. A common alternative approach is to start with the more concrete specification like the one we will present in section 5. More concrete specifications are generally easier to write, and allow us to explore the problem in more concrete terms. Then we can write the abstract specification later, having obtained a better grasp of the problem. Formally, since we will then show that the abstract specification is correctly implemented by the concrete one, the result is the same.

5 The concrete applicative specification

We recall our intention to model the system in terms of a queue of messages in transit and a priority queue of messages that have arrived but are yet to be extracted. We need to create specifications of these two queues.

Modules like queues should be made generic so that they can be reused. To define such a generic module we first use a standard parameter module that gives the parameter requirements.

5.1 Parameter classes

For the queue the parameter class ELEM (figure 5) is simple.

```
scheme
  ELEM = class type Elem end
```

Figure 5: The scheme ELEM

For the priority queue we need a total ordering on elements. We could add the necessary features

to ELEM by extension, but we here choose to use parameterisation to first define the scheme PARTIAL_ORDER (figure 6), and then to define TOTAL_ORDER (figure 7) by extension.

```
scheme
  PARTIAL_ORDER(E : ELEM) =
    class
      value
        leq : E.Elem × E.Elem → Bool

      axiom
        [ reflexive ] ∀ a : E.Elem • leq(a, a),

        [ transitive ] ∀ a, b, c : E.Elem • leq(a, b) ∧ leq(b, c) ⇒ leq(a, c)
    end
```

Figure 6: The scheme PARTIAL_ORDER

The formal parameter of PARTIAL_ORDER essentially defines an object E of class ELEM.

```
scheme
  TOTAL_ORDER(E : ELEM) =
    extend PARTIAL_ORDER(E) with
      class axiom [ linear ] ∀ a, b : E.Elem • leq(a, b) ∨ leq(b, a) end
```

Figure 7: The scheme TOTAL_ORDER

TOTAL_ORDER illustrates another way of making a class: by **extend**ing another class. Extension is very much like inheritance in object-oriented languages: all the declarations of the first class are inherited by the second.

There is no restriction in RSL on the classes that may be used to make parameter classes.

5.2 Queues and priority queues

Queues are easily specified in terms of lists, and we have seen there is a built-in RSL list type. We present first the simpler FIFO queue A_QUEUE (figure 8).

A_QUEUE is completely concrete: the type *Queue*, the constant *empty*, and the functions *put*, *get* and *is_empty* are all defined explicitly. It is not clear that the constant *empty* is required, but we will need it later for initialising the corresponding imperative queue.

We follow Guttag [15] in using the term "type of interest" for the type *Queue* in A_QUEUE.

```
scheme
  A_QUEUE(E : ELEM) =
    class
      type Queue = E.Elem*

      value
        empty : Queue = ⟨⟩,

        put : E.Elem × Queue → Queue
        put(e, s) ≡ s ⌢ ⟨e⟩,

        get : Queue ~→ E.Elem × Queue
        get(s) ≡ (hd s, tl s) pre ~ is_empty(s),

        is_empty : Queue → Bool
        is_empty(s) ≡ s = ⟨⟩
    end
```

Figure 8: The scheme A_QUEUE

Buffer is the type of interest of A_MESSAGE0. It is the type that an applicative module is trying to define, together with associated functions for generating and observing values of the type. For a module like A_QUEUE, with no subsidiary modules, the type of interest will become its state type when we make an imperative object of it.

The priority queue A_PRI_QUEUE is presented in figure 9. For its type of interest we use an ordered list of elements.

The ordering of the type *Pri_queue* is expressed using a *subtype* expression. The type *Queue* in A_QUEUE includes all finite lists of elements. The type *Pri_queue* only includes those finite lists of elements that are ordered according to the function *is_ordered*.

A_PRI_QUEUE requires TOTAL_ORDER as a parameter, which in turn requires ELEM, so we need two parameters. Such a use of parameters might be considered "higher order" but causes no problems in RSL.

A_PRI_QUEUE also illustrates the use of **case** and **if** expressions.

From the type of *put* we can assume that the second parameter is an ordered list. But this type also claims that the result value will be an ordered list. Since we have an explicit definition, there is the possibility of a contradiction here, and we should check that the defining expression will indeed be an ordered list if the second parameter is. That is, we should prove the theorem

```
∀ l : E.Elem*, e : E.Elem •
  is_ordered(l) ⇒ is_ordered(put(e, l))
```

```
scheme
  A_PRI_QUEUE(E : ELEM, T : TOTAL_ORDER(E)) =
    hide is_ordered in
      class
        type Pri_queue = {| l : E.Elem* • is_ordered(l) |}

        value
          empty : Pri_queue = ⟨⟩,

          put : E.Elem × Pri_queue → Pri_queue
          put(e, s) ≡
            case s of
              ⟨⟩ → ⟨e⟩, ⟨h⟩ ⌢ t → if T.leq(e, h) then ⟨h⟩ ⌢ put(e, t) else ⟨e, h⟩ ⌢ t end
            end,

          get : Pri_queue ~→ E.Elem × Pri_queue
          get(s) ≡ (hd s, tl s) pre ~ is_empty(s),

          is_empty : Pri_queue → Bool
          is_empty(s) ≡ s = ⟨⟩,

          is_ordered : E.Elem* → Bool
          is_ordered(l) ≡ (∀ i, j : Nat • {i, j} ⊆ inds l ∧ i < j ⇒ T.leq(l(j), l(i)))
      end
```

Figure 9: The scheme A_PRI_QUEUE

This is an example of what is called in RAISE a *confidence condition*. There is a tool in the RAISE toolset that generates such conditions. Others arising from this specification are that the empty list is ordered (from the definition of *empty*) and that the definition of *get* produces an ordered list.

The other common type of confidence condition arises from applications of functions or operators that have preconditions and/or have subtype parameters. The applications of **hd** and **tl** in the definition of *get* will generate the confidence conditions that their arguments are not empty, and perhaps remind us that a precondition is needed for *get*. In the definition of *is_ordered*, the applications of the list *l* to the arguments *j* and *i* will generate the conditions that these arguments are in **inds** *l*. Any call of *get* in another module using A_PRI_QUEUE will generate the confidence condition that its argument is ordered and not empty.

Confidence conditions can be proved formally or checked informally. The latter is often sufficient; the kinds of errors they point to are usually oversights and soon corrected once identified. We also have to beware of the danger, though it seems slight in practice, that a confidence condition can be proved precisely because there is a contradiction, from which anything can be proved.

5.3 The concrete applicative message system

Having defined the two types of queue, we can use them to form the new top-level specification A_MESSAGE1 (figure 10).

```
scheme
  A_MESSAGE1 =
    hide PQ, Q in
      class
        object
          PQ : A_PRI_QUEUE(T{Message for Elem}, T),
          Q : A_QUEUE(T{Message for Elem})

        type Buffer = PQ.Pri_queue × Q.Queue

        value
          put : T.Message × Buffer → Buffer
          put(m, (pq, q)) ≡ (pq, Q.put(m, q)),

          get : Buffer ~→ T.Message × Buffer
          get(pq, q) ≡ let (e, pq′) = PQ.get(pq) in (e, (pq′, q)) end pre can_get(pq, q),

          can_get : Buffer → Bool
          can_get(pq, q) ≡ ~ PQ.is_empty(pq),

          shift : Nat × Buffer → Buffer
          shift(n, (pq, q)) ≡
            if n = 0 ∨ Q.is_empty(q) then
              (pq, q)
            else
              let (m, q′) = Q.get(q), pq′ = PQ.put(m, pq) in shift(n − 1, (pq′, q′)) end
            end
      end
```

Figure 10: The scheme A_MESSAGE1

We instantiate the two component queues as objects which are hidden in A_MESSAGE1. This is the most common way of using component modules. Hiding them ensures that only the upper module has access to them and so gives control over how they are used. No other part of the overall system can access them.

We can use the object T for both parameters of A_PRI_QUEUE, as the class TYPES of T meets the implementation requirements for both. We want to use *Message* for the type *Elem*, and we can achieve this with a *fitting* applied to the first actual parameter.

If we had not included an appropriate *leq* function in TYPES, or if it had been defined differently, such as by a function *higher*, say, then we could have defined an extra object in A_MESSAGE1 to use as the second parameter of A_PRI_QUEUE:

```
object
T1 : class
        value
          leq : T,Message × T.Message → Bool
          leq(m1, m2) ≡ ~ T.higher(m1, m2)
      end
```

We need to check that the class of the actual parameters of A_PRI_QUEUE and A_QUEUE implement the classes of the formal parameters (see section 6). Most of this is checked statically by tools, but we must also check that the definition of *leq* in TYPES satisfies the properties expressed in the axiom of TOTAL_ORDER, plus the two axioms it inherited from PARTIAL_ORDER.

We can use the types of the component objects PQ and Q to provide the concrete type *Buffer* for the upper module, here using a product. RSL also provides a *record* type constructor that we could have used. Records are isomorphic to tuples, but provide a richer syntax for extracting and changing components, and are particularly useful when there are more components.

The definition of *shift* illustrates the use of the **let** expression. The form used is a shorthand for the nested **let**

```
let (e, q′) = Q.get(q) in
   let pq′ = PQ.put(e, pq) in
      shift(n − 1, (pq′, q′))
   end
end
```

When a message is *put* into the combined queue it is initially added to the "transit" queue Q. Messages are extracted from the "arrived" queue PQ with *get*. We have included a function *shift* to transfer a number of messages (if available) from one queue to the other. *shift* has effectively been added to the user interface of the system. Without it the system would meet its "safety" requirements, but not its "liveness" requirements. A safety requirement is that "nothing bad happens": messages do not get lost; lower priority ones do not overtake higher priority ones. A liveness requirement is that "something good happens": input messages can eventually be extracted. We will see later how to put *shift* inside the system.

We intend A_MESSAGE1 to be correct with respect to A_MESSAGE0. What this means and how we check it is presented in the next section.

6 The implementation relation

The *implementation relation* between classes (also termed the *refinement relation*) is the relation used to define the correctness of a development step from a more abstract module to a more concrete one.

Parameterised schemes have objects as formal parameters, and objects must be used as actual parameters. There needs to be a relation between the classes of the formal and actual objects. This is also the implementation relation, as we remarked earlier.

Class B *implements* a class A (written B $\preceq$ A) if and only if

1. the signature of B includes the signature of A
2. all the properties of A hold in B

The first condition is called *static implementation*. It means that

- for every type in A there is a type of the same name in B, and with the same defining type if the type in A is not a sort
- for every value, variable, or channel in A there is a value, variable, or channel in B with the same name and same maximal type (i.e. ignoring subtypes)
- for every object in A there is an object in B of the same name with a class that statically implements the class of the object in A.[1]

The second condition involves the *properties* of a class, which arise from

- axioms
- value definitions
- subtype conditions on values, variables and channels
- initialisations of variables
- properties of objects defined in the class

[1] The original version of RSL [13] allowed schemes to be defined in classes, with the obvious static implementation requirement. These are disallowed in the later version [14] to simplify the logic. There are also good methodological reasons for this restriction. There is no danger in making all schemes global (not defined within a class) — unlike objects which need to be protected against global access, particularly when they are imperative. The problem with a scheme defined inside a class is that it may refer to entities defined in the rest of the class, and this is bad practice since we believe that such sharing should generally be made explicit through parameterisation.

A formal definition of the properties of a class is given in the method book [14].

We can see that if we can show that A_MESSAGE1 implements A_MESSAGE0 we will have shown that all the functions we were supposed to provide (*put* and *get*) are still supplied, and that their now concrete definitions satisfy the axioms we used to express the required properties. So, if A_MESSAGE0 met the requirements, A_MESSAGE1 will.

6.1 Checking implementation

Static implementation can be checked by tools. For the properties part we generally need to do proof. We can choose to do it formally or (partly or wholly) informally.

It is a good idea to first document an informal argument. Then we (or, better, someone else) can decide if it is sufficiently convincing, or whether some parts, or even all of it, should be done using a proof tool.

A very informal argument is that the two queues never lose or create elements. For each, *put* adds an element and *get* may remove one, while the others remain. *put* and *get* at the top level call *put* and *get* respectively at the lower level, and therefore have similar properties to the lower level functions. Finally, *shift* transfers the same number of elements from one queue to the other. Thus we can see there is no loss or gain of messages.

A_QUEUE clearly maintains its order of elements. A_PRI_QUEUE only puts a new element in front of an existing one if the new element has a higher priority. We have to check that this means that the elements in the priority queue behind this existing one must also have elements of a lower priority than the new one, i.e. that the queue is ordered.

This very informal argument might suffice in this case. But for critical systems a formal argument is necessary. For such an argument we must first define in terms of A_MESSAGE1, i.e. in terms of a buffer consisting of a queue and a priority queue, the hidden and undefined function *buffered* of A_MESSAGE0.

The definition is

```
value
  buffered : Buffer × T.Message* → Bool
  buffered((pq, q), l) ≡
    (∃ l1 : T.Message* • l = l1 ^ q ∧ permutation(l1, pq))
```

With this definition it is possible to prove the axioms of A_MESSAGE1. The RAISE tools include a *justification editor* which can be used either to create a completely formal proof of this, or else a proof in which some steps or lemmas are done only informally. Such a justification

is termed *rigorous*. Informal parts can be done formally later if they are questioned during quality control.

Justifications about modules that use other modules typically involve theorems about the used modules. For example, we will need the theorem that states that when we *put* an element in a priority queue we get a permutation of the old queue elements plus the new one:

$$\forall\ q : \text{Pri\_queue}, e : \text{E.Elem} \bullet \\ \quad \text{permutation}(\text{put}(e, q), q \frown \langle e \rangle)$$

RAISE supports the use of *theories* about modules that can be used in justifications about other modules using these modules.

The introduction of the function *shift* introduces an extra condition that we ought to check. The implementation relation allows the introduction of new functions, and there is nothing to prove about them, since the previous description said nothing about them and so placed no requirements on them. But *shift* is special in that we intend eventually to make it internal, like a demon. Hence, at the most abstract level (where we could not define *shift* because there was no notion of two queues), it seems that *shift* should be invisible. In other words, if two concrete states are related by *shift*, they should represent equivalent abstract states. The normal definition of equivalence over abstract types is observational equivalence. But *shift* will make changes that can be observed by *can_get*, for example, since it can change the "arrived" queue from empty to non-empty, so *shift* is not invisible. A more general notion of equivalence in A_MESSAGE0 is to use the relation *buffered*: two *Buffer* values will be equivalent if they are bufferings of the same input. This leads to the following property which should be satisfied by *shift*:

$$\forall\ \text{buff}, \text{buff}' : \text{Buffer}, n : \mathbf{Nat} \bullet \\ \quad \text{buff}' = \text{shift}(n, \text{buff}) \Rightarrow \\ \quad (\exists\ l : \text{T.Message}^* \bullet \text{buffered}(\text{buff}, l) \wedge \text{buffered}(\text{buff}', l))$$

This corresponds precisely to our informal idea that *shift* is safe in not losing or creating messages, and it can be proved.

7 From applicative to imperative

The applicative specification is generally complete when all the types of interest are concrete. If A_QUEUE or A_PRI_QUEUE had abstract types of interest we would develop these into concrete applicative modules first. Or we might well have concrete versions for such standard modules already done from other projects.

The transformation from applicative to imperative is straightforward. It proceeds module by module, but depends on whether the module is a "leaf" module in the hierarchy, like A_QUEUE or A_PRI_QUEUE, or is an upper module instantiating others, like A_MESSAGE1.

For a leaf module we create an imperative module like I_QUEUE (figure 11).

```
scheme
  I_QUEUE(E : ELEM) =
    hide v, Q in
      class
        object Q : A_QUEUE(E)

        variable v : Q.Queue := Q.empty

        value
          empty : Unit → write any Unit
          empty() ≡ v := Q.empty,

          put : E.Elem → write any Unit
          put(e) ≡ v := Q.put(e, v),

          get : Unit ~→ write any E.Elem
          get() ≡ let (e, v') = Q.get(v) in v := v' ; e end pre ~ is_empty(),

          is_empty : Unit → read any Bool
          is_empty() ≡ Q.is_empty(v)
      end
```

Figure 11: The scheme I_QUEUE

The method is:

- Use the same scheme parameters, if any, as the corresponding applicative module.
- Define an object instantiating the corresponding applicative module. This object is hidden.
- Define a variable of the type of interest of the applicative module. This is usually initialised, and is always hidden. (If the concrete type of interest has several components, several corresponding variables may be defined instead of the single one.)
- For each non-hidden constant and function from the applicative module, define a function of the same name. The type of an imperative function corresponding to a constant of the type of interest (like *empty*) is "**Unit → write any Unit**". **Unit** is the simplest type in RSL, consisting of just one value, written "()". The type of an imperative function corresponding to an applicative function is obtained by removing any occurrences of the

type of interest. If this leaves nothing on either side of a function arrow, **Unit** is used to fill the hole.

- For a generator function (one whose applicative result type depends on the type of interest) include the access "**write any**" in its type.
- For an observer function (one whose applicative parameter type(s) depend on the type of interest) include the access "**read any**" in its type.
- Define the imperative functions using the corresponding applicative constants or functions in the obvious manner. The definitions of generators will include assignments to the variable(s).

It is possible to simplify the result by replacing types, constants and function calls from the applicative module with their applicative definitions. For example, the defining expression of *put* in *I_QUEUE* could be replaced by "v := v ^ ⟨e⟩". In the case of I_QUEUE this would eventually allow the object Q to be deleted. With I_PRI_QUEUE this is not immediately possible because in A_PRI_QUEUE the function *put* is recursive. But this simplification has the danger of introducing transcription errors.

A **variable** declaration is straightforward: we need to give it a type and, optionally but usually (as here), an initial value.

The inclusion of *accesses* in the types of imperative functions indicates that they can **read** or **write** variables. **write** includes **read**. **any** in such an access means all variables defined in the same class or in the classes of subsidiary objects, and allows us to write the accesses without knowing what variables there are in subsidiary objects. **any** is necessary because the names of such variables will generally be hidden and so cannot be referred to directly. In I_QUEUE, there are no subsidiary objects and "**any**" could be replaced throughout by "v". The accesses are necessary in the logic because the proof rules for imperative functions are not the same as for applicative ones: many proof rules only hold for *read-only* (no write accesses) or even *pure* (no accesses) expressions. For example, the addition operator for integers is only commutative if both arguments are read-only (and terminating).

There are no statements in RSL, only expressions. What are regarded as statements in some languages are in RSL expressions with accesses. Sequencing (";") is a *combinator* for expressions.

I_PRI_QUEUE is constructed in the same fashion.

For an upper module like A_MESSAGE1 we construct a corresponding imperative module like I_MESSAGE1 (figure 12):

- An instantiation of an applicative module is replaced by an instantiation of its imperative counterpart.

- The type of interest is removed. For each component of it not defined in subsidiary modules, we define a variable of that type, and hide it in the module.
- Function types are derived from applicative counterparts in the same way as for the leaf modules.
- The imperative definitions are derived from the applicative ones: formal parameters of the type of interest are removed, as are the type of interest parameters of applications of functions defined in subsidiary modules.
- The bodies of generators will include assignments of the new type of interest value being generated to the module's variable(s).

The rules for transformations of expressions are too long to be described here, but are generally obvious. Full details are included in the method book [14].

```
scheme
  I_MESSAGE1 =
    hide PQ, Q in
      class
        object
          PQ : I_PRI_QUEUE(T{Message for Elem}, T),
          Q : I_QUEUE(T{Message for Elem})

        value
          put : T.Message → write any Unit
          put(m) ≡ Q.put(m),

          get : Unit ~→ write any T.Message
          get() ≡ PQ.get() pre can_get(),

          can_get : Unit → read any Bool
          can_get() ≡ ~ PQ.is_empty(),

          shift : Nat → write any Unit
          shift(n) ≡
            if n = 0 ∨ Q.is_empty() then
              skip
            else
              let m = Q.get() in PQ.put(m) ; shift(n − 1) end
            end
      end
```

Figure 12: The scheme I_MESSAGE1

8 From imperative to concurrent

Our system is likely to have several human users or other software components wanting to put messages into it concurrently. It may also have several concurrent readers (perhaps extracting messages for different destinations). In general there is a need to ensure some degree of locking for such transactions, to prevent interference — the changing of the state by one transaction while others are accessing it. Locking in databases is a much studied subject and not our topic here. It is often sufficient just to make sure only one transaction can run at once; to make them atomic viewed from outside. The standard transformation in the RAISE method from imperative to concurrent ensures this. The transformation involves encapsulating the imperative objects with *server* processes that do the actual interaction with the imperative objects, plus small *interface* processes that can be called to achieve the transactions. We call writing the encapsulating modules a transformation to emphasise its generally mechanical nature, even though the imperative modules are not changed.

We refer to server and interface "processes", but they are just functions in RSL. The term "process" is used informally for a function intended to be run concurrently with others.

We illustrate the method by showing C_QUEUE. C_PRI_QUEUE is written in the same way.

Before presenting C_QUEUE, we show how to deal with partial interface processes. *get* is partial since the queue may be empty. But with a concurrent system, calling one interface process (like calling *is_empty* to check the queue is not empty), and then acting on the response (like calling *get* if the answer to the first call is **false**) is not adequate. Some other user might in the meantime have made the queue empty. So we need to make functions like *get* total, and the remedy here is obvious. We define a result type for *get* that can give either a message or an indication that none is available. We can use an RSL *variant* type for this purpose:

```
type
  Result == nil | result(elem : Elem)
```

Here the type *Result* is defined to consist of the value *nil* plus a type consisting essentially of all values of type *Elem*. *result* is a function injecting *Elem* values into *Result* values, and *elem* the corresponding projection function. *nil* and *result(e)* are by definition different for all *e*.

We need to include this type in the parameter of C_QUEUE and C_PRI_QUEUE, so we define a new scheme ELEM_RES (figure 13):

```
scheme
  ELEM_RES = extend ELEM with class type Result == nil | result(elem : Elem) end
```

Figure 13: The scheme ELEM_RES

ELEM_RES implements ELEM (by construction) and so can be used in C_QUEUE (figure 14) to make an actual parameter for TOTAL_ORDER.

```
scheme
  C_QUEUE(E : ELEM_RES) =
    hide I, CH in
      class
        object
          I : I_QUEUE(E),
          CH :
            class
              channel empty : Unit,  put : E.Elem,  get : E.Result,  is_empty : Bool
            end

        value
          init : Unit → write any in any out any Unit
          init() ≡ I.empty() ; main(),

          main : Unit → write any in any out any Unit
          main() ≡
            while true do
              CH.empty? ; I.empty()
              □
              CH.get ! if ∼ I.is_empty() then E.result(I.get()) else E.nil end
              □
              let e = CH.put? in I.put(e) end
              □
              CH.is_empty ! I.is_empty()
            end,

          empty : Unit → out any Unit
          empty() ≡ CH.empty ! (),

          get : Unit → in any E.Result
          get() ≡ CH.get?,

          put : E.Elem → out any Unit
          put(e) ≡ CH.put ! e,

          is_empty : Unit → in any Bool
          is_empty() ≡ CH.is_empty?
        end
```

Figure 14: The scheme C_QUEUE

Concurrency in RSL is based on *channels* which can pass values [12] between concurrently executing expressions. RSL uses a syntax close to that of CSP [10] but its semantics of concurrency are closer to CCS [11].

Each generator and observer will need at least one channel to be defined, and will need two if the parameter and result types are both different from **Unit**. Channels are defined rather like variables (but without any initialisation) by giving their identifiers and types. A channel c can transfer the value of an expression e through an output expression $c!e$. The value is input by the expression $c?$. Communication is point-to-point (not broadcast) and may only happen if the input and output expressions are combined in parallel and may be executed simultaneously. If more than one output on a channel, say, is ready to execute, and there is a corresponding input, then only one of the outputs will occur and the other will have to wait for another input to be available. Such choices as to which output will occur are nondeterministic and we try to design systems so that either they do not occur, or the choice does not matter. So, for example, if two users call the *put* interface process simultaneously then *main* may accept the messages in either order.

Functions which include input or output need **in** and **out** accesses to channels in their signatures in much the same way that functions accessing variables need **read** and **write** accesses.

The server process *main* is a loop which on each iteration offers four choices. The choice operator [] is an "external" choice whose behaviour is determined by the "environment", i.e. any processes running concurrently with it. Each choice is "guarded" by an input or output, so on each iteration *main* will wait for one of the interface processes to be called and hence be ready to do the complementary part of the communication.

RSL cannot express fairness, so it is theoretically possible for an interface process invocation to wait indefinitely while *main* communicates with others. Dealing with fairness has to be done by other means, if it is a possible problem. If our ultimate users are human and therefore comparatively slow then, unless there is a very large number of them, fairness will not be an issue for our system.

Each choice clause in a server process accepts any parameters, completes the appropriate transaction with the imperative object, and then returns any result. There is one clause for each interface function. We have to check that the inputs and outputs are appropriately complementary: if an interface process outputs a parameter and then inputs a result and we forget to include the corresponding output of the result in the server then the interface process will wait forever — it will deadlock. Checking for this kind of deadlock is easy, and, provided we also ensure that all the channels are hidden and all the server process are invoked initially, it is the only way that deadlock can occur in this style of design. Checking against deadlock in hierarchic designs like this one is essentially syntactic.

We also include an *init* process that initialises the imperative object and invokes *main*.

For the concurrent top level module C_MESSAGE1 we have to add a definition of *Result* to

TYPES, so that TYPES can implement ELEM_RES. This would normally be done just by editing TYPES — such addition cannot harm previous specifications. In our case we created a new TYPES1 and a corresponding global object T1 so that both versions of TYPES could be pretty-printed from one directory for this paper.

For C_MESSAGE1 (figure 15) there are server processes in subsidiary modules and so it does not need one of its own. We implement its functions in the obvious way based on its imperative version.

```
scheme
  C_MESSAGE1 =
    hide PQ, Q, shift in
      class
        object
          PQ : C_PRI_QUEUE(T1{Message for Elem}, T1),
          Q : C_QUEUE(T1{Message for Elem})

        value
          init : Unit → write any in any out any Unit
          init() ≡ PQ.init() ‖ Q.init() ‖ shift(),

          put : T1.Message → in any out any Unit
          put(m) ≡ Q.put(m),

          get : Unit → in any out any T1.Result
          get() ≡ PQ.get(),

          can_get : Unit → in any out any Bool
          can_get() ≡ ∼ PQ.is_empty(),

          shift : Unit → in any out any Unit
          shift() ≡
            while true do
              case Q.get() of T1.nil → skip, T1.result(m) → PQ.put(m) end
            end
      end
```

Figure 15: The scheme C_MESSAGE1

We also include an *init* process that calls the *init* processes of its component modules in parallel (so we can be sure that all servers are started when the top level *init* is invoked).

There is an extra process in this module that is not part of the standard method: *shift*, also invoked by *init*. This is a way of implementing *shift* as a separate process that shifts messages

from the transit queue to the arrived queue. Previously we had provided a *shift* function but not specified when it would be called. Now it becomes an internal demon, and so can be hidden.

shift has become iterative instead of recursive. It has also lost the first parameter, which would have been used to say how many messages to attempt to shift, in favour of a loop that continuously shifts messages as soon as each is available.

The process *shift* is not necessarily intended to be implemented in this manner. Rather it reflects the expected behaviour of other, perhaps already existing, and perhaps hardware, components of our final system. It is quite common for parts of specifications to be used to record assumptions about the system's eventual operating environment, rather than to specify what we intend to create. Recording such assumptions as part of the specification is useful in itself: formalising them in the specification allows us to prove properties of our system based on these assumptions.

shift may appear too greedy. It will continually call *Q.get* and we might be worried about fairness. We can deal with this when we introduce time in section 9.

The structure of the final specification may seem strange. C_MESSAGE contains objects instantiating C_QUEUE and C_PRI_QUEUE. Yet if we expect a distributed implementation, these objects are likely to be on different machines. So the structure of the specification seems to be different from the expected structure of the implementation. There are two aspects to consider:

1. The specification structure is an abstraction of the complete system. The structuring is designed in particular to ensure information hiding and privacy of connection. This enables us to reason about the communication: we need to be able to justify that if a message is input it can eventually become available for output. How these aspects are achieved is another matter. We may, again, be specifying our assumptions about how the system is built, e.g. what mechanisms may be used to connect it, rather than specifying components we expect to implement. It is important to note that only by specifying the complete system can we reason about system properties like no loss or unintended reordering of messages.

2. In terms of the expected program code, we intend to implement the server processes (and their associated imperative objects) in our intended programming language(s) quite directly. But the communication primitives in RSL will require modelling in different ways according to how the system is built and what language (or languages) are used to implement its processes. Channels might eventually be modelled using Ada tasking, or Unix sockets, or internet connections, or e-mail messages, or even some physical transfer of data.

9 Adding time to RSL

We have recently started a project to add time to RSL so that it can deal with real-time problems. We aim to be able to interpret timed RSL descriptions in terms of Duration Calculus (DC) [16] formulae. This will allow us to use the power of RAISE in describing and developing large systems and also the power of DC for describing and reasoning about timing properties. This work is currently at an early stage, though we have defined a proposed extension to RSL, defined the changes to the proof system, and given an operational semantics [17].

The extension is very small: essentially just a **wait** construct. For convenience, a type name **Time** is also included, but it is only an abbreviation for the non-negative real numbers. As an illustration, we can use it in C_MESSAGE1. We add the definition

value δ : **Time** • $\delta > 0.0$

so that δ is an underspecified but strictly positive amount of time. We also change the definition of *shift* to

```
shift : Unit → in any out any Unit
shift() ≡
    while true do
        wait δ ;
        case Q.get() of T1.nil → skip, E.result(m) → PQ.put(m) end
    end
```

Thus, remembering that *shift* records our assumptions about the mechanism for message transfer, we are now assuming a periodic process that transfers available messages.

A common use of **wait** is to add choices to server processes. For example, a process of the form

```
while true do
    let x = c? in ... end
    □
    wait δ ; ...
end
```

will repeatedly either accept an input on channel c, or, if this is not available within time δ (again defined to be strictly positive) take some other action. We can use "timer" variables that are started (set to zero) by some events, reset (set to some negative value) by others and incremented

by δ whenever a **wait** δ occurs and they are "running" (not negative). Such variables can then be used to tell us the durations of intervals between events and hence, if events are associated with state changes, to tell us the durations of states. Durations of states are the basis of DC, so we can obtain a DC interpretation of the RSL description.

10 The RAISE tools

The tools for RAISE produced during the RAISE and LaCoS projects are available (free for research and education). They include:

- syntax directed editors that also do static type checking for modules, theories and development relations
- a justification editor for creating and checking proofs
- a confidence condition generator
- pretty printers (into LaTeX) for modules, theories, development relations and justifications
- translators of subsets of RSL into Ada and C++
- "make" utilities for maintaining modules, theories, development relations, translations, and documents containing pretty printed entities

This document and the specifications described in it were produced using these tools. See the RAISE home page http://dream.dai.ed.ac.uk/raise/.

The tools listed above are currently only available for Sun workstations. A more portable set of tools that will run on PCs as well as Unix or Linux is being developed by UNU/IIST and partners in universities in developing countries and can be downloaded from ftp://ftp.iist.unu.edu/pub/RAISE/dist. It currently covers syntax and static type checking, pretty-printing, and confidence condition generation.

11 Further reading

The two books on RAISE [13, 14] are the main sources for detailed information on the language and method.

The original semantics of RAISE is written in a denotational style [18]. The method book [14] provides an axiomatic semantics, a proof theory. An operational semantics of a kernel of RSL was also produced [19, 20].

The experiences from the LaCoS project were summarised in experience reports [1, 2]. Other published papers on projects using RAISE include [21, 22, 23, 24, 25, 26, 27].

Acknowledgements

Richard Moore made very useful comments on an early draft of this paper.

References

[1] 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.

[2] B. Dandanell, J. Gørtz, J. Storbank Pedersen, and E. Zierau. Experiences from Applications of RAISE. In *FME'93: Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[3] Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall International, 1980.

[4] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

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

[6] R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. In *Proc. of (IJCAI) Int'l. Joint Conf. on AI.* Boston, Aug. 1977.

[7] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of obj-2. In *12th Ann. Symp. on Principles of Programming*, pages 52–66. ACM, 1985.

[8] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 1988.

[9] J. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical Report 5, DEC SRC, Dig. Equipm. Corp. Syst. Res. Ctr., Palo Alto, California, USA, 1985.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[11] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[12] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value passing. In *Proceedings: 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *LNCS*. Springer Verlag, 1990.

[13] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.

[14] The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995.

[15] J.V. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6), June 1977.

[16] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991. Revised June 3, 1992.

[17] Chris George and Xia Yong. An Operational Semantics for Timed RAISE. Technical Report 149, UNU/IIST, P.O.Box 3058, Macau, November 1998.

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

[19] 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.

[20] D. Bolignano and M. Debabi. A Denotational Model for the Integration of Concurrent, Functional, and Imperative Programming. In *Proceedings of the NAPAW'93 Workshop*. Cornell University, August 1993.

[21] F. Erasmy and E. Sekerinsky. Stepwise Refinement of Control Software — A Case Study Using RAISE. In *FME'94: Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[22] F. Erasmy and E. Sekerinsky. RAISE. In *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[23] C.W. George. The NDB Database Specified in the RAISE Specification Language. *Formal Aspects of Computer Science*, 4(1), 1992.

[24] C.W. George. A Theory of Distributed Train Rescheduling. In *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[25] J. Gørtz. Specifying Safety and Progress Properties with RSL. In *FME'94: Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[26] C.W. George and R.E. Milne. Specifying and Refining Concurrent Systems — an Example from the RAISE Project. In *Proceedings of 3rd Refinement Workshop*. Springer-Verlag, 1990.

[27] A.E. Haxthausen and C.W. George. A Concurrency Case Study Using RAISE. In *FME'93: Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.