Lecture Notes on
The RAISE Development Method

Anne Haxthausen

April 1999

Contents

1 Introduction

RAISE is an acronym for *Rigorous Approach to Industrial Software Engineering* and is a product consisting of

- a software development method
- a formal specification language, RSL
- tools supporting the language as well as the method
- technology transfer material (documents, videos and courses)

This lecture note is a tutorial on the method. It is based on extracts of the RAISE method book [9] and a tutorial by Chris George and myself. The development steps in the harbour example in section 3 differ from the original ones in the book.

The language is described in [8], and the tools and technology transfer material have been commercially available for some time.

In the rest of this introduction we first describe the RAISE background and projects and then the objectives and contents of the tutorial.

1.1 The RAISE background

RAISE is the result of an ESPRIT project carried out during 1985 - 1990 by four companies: DDC/CRI (DK), NBB/ABB/SYPRO (DK), STL/BNR (UK) and ICL (UK).

The starting point for RAISE was the Vienna Development Method, VDM [2], [12], which had had success in industry, but lacked a number of useful features. Hence, the aim was to enhance VDM with structuring facilities, algebraic specification, concurrency, formal semantics and computer-based tools.

Many languages and methods have been sources of inspiration for the enhancements, e.g. Z [1], ML [14], Clear [3], ASL [16], ACT ONE [5], LARCH [10], OBJ [6], CSP [11] and CCS [15].

1.2 The LaCoS continuation

Another ESPRIT project, called LaCoS — Large Scale Correct Systems Using Formal Methods — did follow up on RAISE. The aim of LaCoS was to use RAISE on real industrial applications and, based upon the experience (see [4]) from these applications, to further evolve the RAISE product. The project was carried out in the period 1990 - 1995 by the following companies: CRI (DK), CAP PROGRAMATOR (DK), BNR Europe (UK), Lloyd's Register (UK), Bull (F), MATRA Transport (F), Inisel Espacio (E), Space Software Italia (I) and Technisystems (GR).y

1.3 Contents of lecture notes

First, in section 2, we give a short overview of the method, and then, in section 3 we illustrate the method by an example.

The reader is assumed to be familiar with the RAISE Specification Language.

2 Method overview

The aim of this section is to give a overview of the RAISE method.

First, in subsection 2.1, we state the characteristics of the RAISE development method, and in subsection 2.2, we describe the RAISE implementation relation. Then, in subsection 2.3, we describe different specification styles and their roles in RAISE developments.

2.1 Characteristics of the RAISE method

The RAISE method is based on the *stepwise development* paradigm according to which the software is developed in a number of steps.

Each step starts with a description of the software and produces a new description which is more detailed, see figure 1. The specifications are formulated in the RAISE specification language, RSL. The first specification is typically very abstract. After a number of steps in which design decisions are taken one may obtain a specification which is conveniently concrete to be (perhaps automatically) translated into a programming language.

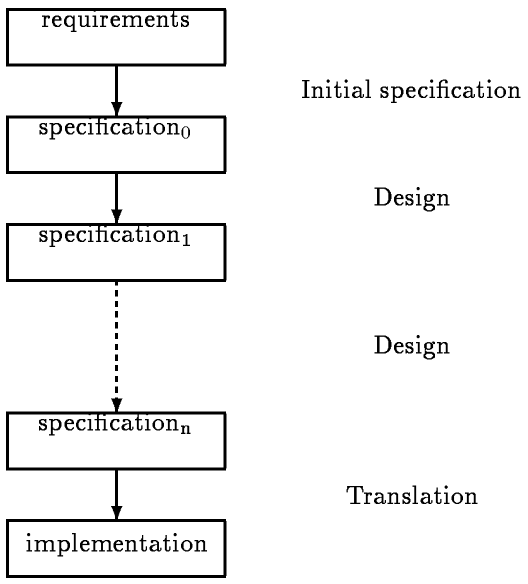


Figure 1: Stepwise development

The stepwise development uses the *invent-and-verify* approach. That is, in each step, first the new specification is invented and then it is verified that it conforms to (is a correct development of) the previous specification. This approach is in contrast to the transformational approach, known from for instance PROSPECTRA [13], where the new specification is obtained from the old one by a transformation and thereby is correct by construction.

The exact relationship (conformance) of the specifications in a development step can be a user-defined relation or the pre-defined *implementation* (*refinement*) relation, which is described in next section.

As a very important feature, the RSL structuring mechanisms, together with the implementation or refinement relation allow for *separate development*. For instance, assume that two modules, A and B, are to be developed by two different teams as shown in figure 2. One team refines A_0 to A_m, and another team refines B_0 to B_n, while still assuming only the properties of A_0 which acts as an contract between the two developments. When the developments by the two teams are complete they integrate by using A_m instead of A_0 in B_n, to form B_{n+1}. Then B_{n+1} refines B_0. Refinement is *compositional*: we can refine components separately and then integrate to get a refinement as a whole.

Verification, or *justification* as it is called in RAISE, is *rigorous* (as the 'R' in RAISE indicates). That is, the method allows the verification to be formal but does

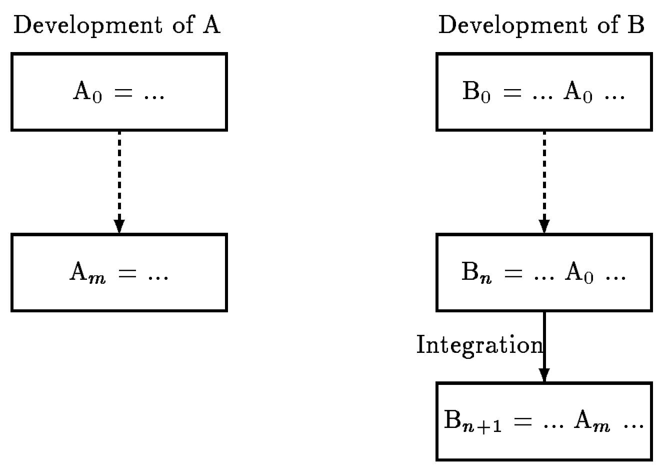


Figure 2: Separate development

not require it.

The postulation of a certain development relation is an example of a *justification condition*. Other examples of conditions that we may want to state and justify are *theorems* about modules and *confidence conditions* for modules. The latter are conditions that ensure that there are no unintended use of language constructs like division by zero or use of a function outside its precondition.

2.2 The implementation Relation

A class expression *ce1 implements* a class expression *ce0* iff the following two conditions are satisfied:

1. *ce1 statically implements ce0*. That is, the visible part of the maximal signature of *ce0* is included in the visible part of the maximal signature of *ce1*.

2. Every theorem of *ce0* is a theorem of *ce1*.

For a further explanation, see the RSL book [8] sections 30.5-30.6.

2.3 Choice of specification style

For each module in a specification we have to decide whether it should be applicative or imperative (i.e. without or with variables), and sequential or concurrent. This gives four combinations:

applicative sequential: a "functional programming" style with no variables or concurrency

imperative sequential: with variables, assignment, sequencing, loops, etc. but with no concurrency

applicative concurrent: functional programming but with concurrency

imperative concurrent: with variables, assignment, sequencing, loops, etc. and concurrency

Applicative concurrent specifications are often inappropriate as the basis for programming language implementations; the main processes are recursive in structure and their continued execution will keep increasing the size of the call stack. So unless we are implementing in an applicative language that can overcome this problem we shall need to use an imperative style; the use of variables enables the recursion to be replaced by a loop. Hence there are only three major kinds of module that we are usually concerned with and that we shall concentrate on in this tutorial: applicative sequential, imperative sequential and imperative concurrent. We will generally abbreviate these to applicative, imperative and concurrent.

Our experience is that of the three, the applicative style is the easiest both to formulate and to reason about in justifications. It also turns out that one can easily start with applicative specifications and develop them into imperative or concurrent ones. For this reason we will adopt this as the basis for the method in the tutorial.

As well as distinguishing between applicative and imperative, sequential and concurrent styles of specification we can also distinguish between abstract and concrete styles.

By abstractness we mean, in general, writing specifications to leave as many alternative development routes open as possible. In other words, the fewer design decisions we have taken in expressing a specification the more abstract it is. By design decisions we mean things like

- deciding how to formulate a module using other modules
- deciding on a particular data structure
- deciding on a particular algorithm
- deciding what variables to use
- deciding what channels and patterns of communication to use

The opposite of "abstract" is "concrete". The distinction between the two is not a black and white one, but we can characterize modules in each of the three categories as tending to be abstract or concrete.

abstract applicative modules will typically be algebraic (using abstract types, i.e. sorts) and will use signatures and axioms rather than explicit definitions for some or even all functions.

concrete applicative modules will typically be model-oriented (using concrete types such as integers, lists, maps, etc.) and will contain more explicit function definitions.

abstract imperative modules will not define variables but will use **any** in their accesses and will use axioms.

concrete imperative modules will define variables and will contain more explicit function definitions.

abstract concurrent modules will not define variables or channels but will use **any** in their accesses and will use axioms.

concrete concurrent modules will define variables and channels and will contain more explicit function definitions.

Again it must be stressed that these are relative rather than absolute distinctions. A module may be abstract in some ways and concrete in others. And certainly a system specification will contain modules in both varying styles and varying degrees of abstractness. We will also use the term *axiomatic* to describe a style of value definition in terms of signature and axiom.

We will adopt a naming convention in this tutorial that applicative modules will be prefixed "A_", imperative ones "I_" and concurrent ones "C_". We will also use the convention that the most abstract modules will be suffixed "0", more concrete ones "1", etc.

3 An example: a harbour system

This section shows the specification and development of a simple information system for controlling entry and exits of ships to a harbour.

3.1 Aims of example

The example is a simple information system, with functions for changing the data, functions for interrogating the data, and invariant properties that the data must satisfy. There is no requirement for concurrent access.

3.2 Requirements

Ships arriving at a harbour have to be allocated berths in the harbour which are vacant and which they will fit, or wait in a "pool" until a suitable berth is available. Develop a system providing the following functions to allow the harbour master to control the movement of ships in and out of the harbour:

arrive: to register the arrival of a ship

dock: to register a ship docking in a berth

leave: to register a ship leaving a berth

The harbour is illustrated in figure 3.

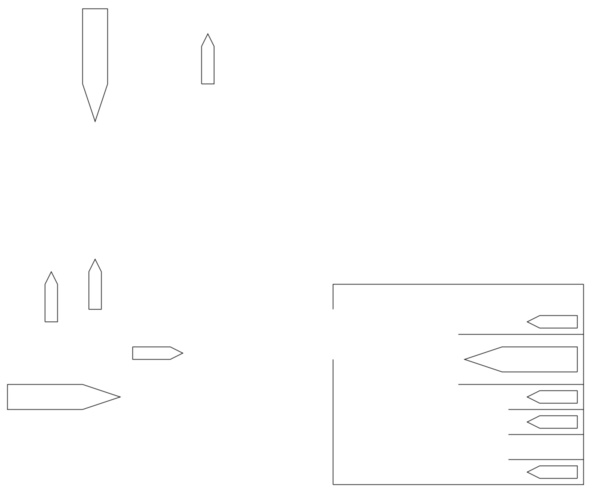


Figure 3: Harbour

We assume all ships will have to arrive and be waiting (perhaps only notionally) in the pool before they can dock. So we can picture the state transitions for ships in figure 4.

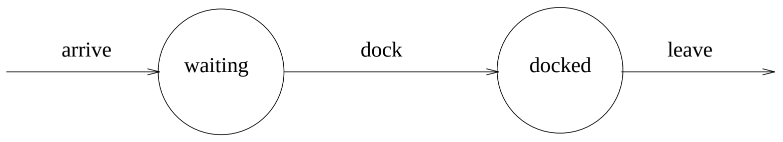
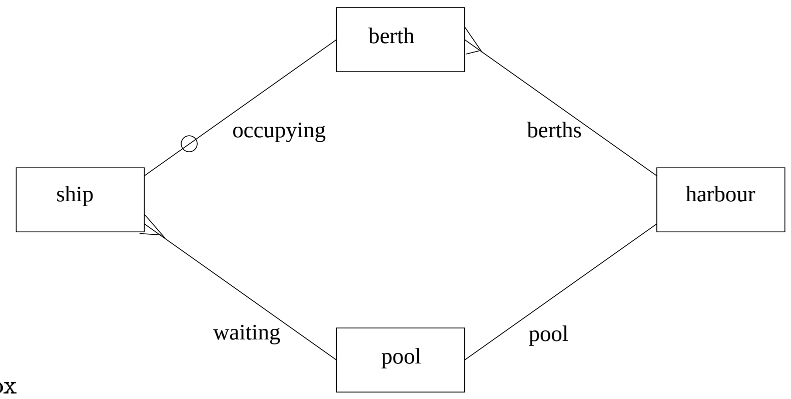


Figure 4: State transitions for ships

3.3 Initial formulation

We first ask what are the objects of the system. Mentioned in the requirements are ships, berths, pool and harbour. It also seems that the harbour is, for our purposes, a fixed collection of berths, while the number of ships in the pool will vary. We can show the entity relationships in figure 5.



mbox

Figure 5: Entity relationships for harbour

Then we try to identify attributes of objects and see which ones may change dynamically. Ships have no attributes given in the requirements, except that they may or may not *fit* a berth. We could invent an attribute like *size* but we don't in fact know if this is what determines fit. So we make a note that we will probably need a function

fits : Ship $\times$ Berth $\rightarrow$ **Bool**

which we will leave underspecified, at least until we have discussed with the customers what they want here.

Berths change in that they may be vacant at one time and contain a ship at another time. Hence what we might term *occupancy* is a dynamic attribute.

The harbour seems to be a collection of berths. The members of this collection are apparently fixed.

The pool of waiting ships will change dynamically as ships arrive and dock.

Note that there is often a choice of what we regard as attributes. We could have a dynamic attribute *location* for a ship, which might be *elsewhere*, *waiting* or *docked in berth k*. We could make ships into RSL imperative objects to model this. Then we would have duplicate information if we also had dynamic berths and pool of waiting ships. This would cause extra overhead in changing both objects consistently. Some systems are designed this way — usually when the amount of information is large, queries are common and need to be fast, and changes are less common. However, it is generally a dangerous practice and for this system it seems more appropriate to structure the system on the basis of the harbour and pool of waiting ships, and to calculate the location of a ship if we need to.

Now we can consider what are the *invariants* (properties that are always true) on the data. Possibilities are

- a ship can't be in two places at once
- at most one ship can be in any one berth
- a ship can only be in a berth it fits

There are two ways to deal with such invariants. Where possible we build them into the model. If the occupancy of a berth is modelled as *vacant* or *occupied_by(s)* (where *s* is a ship), the model avoids any possibility of there being more than one ship in any one berth, and so guarantees the second invariant. (There is also the point that we shouldn't try to dock a ship into a berth that is occupied, but this is dealt with separately.) We have already decided to build in to the model the fact that the collection of berths does not change, which could be considered an invariant.

The first and third invariants suggest the (imperative) predicate

```
∀ s : Ship •
   ∼(waiting(s) ∧ is_docked(s)) ∧
   (∀ b1,b2 : Berth •
      occupancy(b1) = occupied_by(s) ∧ occupancy(b2) = occupied_by(s) ⇒
         b1 = b2) ∧
   (∀ b : T.Berth • occupancy(b) = T.occupied_by(s) ⇒ T.fits(s, b))
```

We expect in the initial specification to use an abstract type for the harbour. Having identified an invariant property captured by a predicate *consistent*, say, then we could use a subtype, as in

```
type
   Harbour_base,
   Harbour = {| h : Harbour_base • consistent(h) |}
```

This possibility can be adopted but it will require us to generate confidence conditions for the concrete applicative specification (when we find some concrete type for *Harbour_base*). Otherwise it is very easy to create a concrete applicative specification that passes the refinement check but does not maintain the invariant (and is thus inconsistent). It is a general rule that subtypes of abstract types should not be used unless confidence conditions of the concrete modules are generated and carefully checked.

Instead, we will express as a collection of axioms the property that the state-changing functions maintain the invariant, which makes the property more visible and will force us to justify it when we justify refinement. This may not seem too important in this example, but safety properties typically look like invariants.

For example, if *arrive* is a state-changing function and *consistent* a predicate expressing the invariant, we can write the axiom

```
axiom
   [ arrive_consistent ]
      ∀ s : Ship •
         arrive(s) post consistent() pre consistent() ∧ can_arrive(s)
```

where *can_arrive* is a predicate expressing the precondition for *arrive*.

We now have some mental picture of the objects in the system.

3.4 Development Plan

We want to proceed from applicative to imperative. So the particular method we will use is as follows:

- Define a scheme TYPES containing types and attributes for the non-dynamic entities we have identified, and make a global object T for this.
- Define an abstract (algebraic) applicative module A_HARBOUR0 containing the top level functions, the axioms relating these and the "invariants".
- Develop A_HARBOUR0 to a concrete (model-oriented) applicative module A_HARBOUR1.
- Develop A_HARBOUR1 to a corresponding imperative module I_HARBOUR1.
- Consider any efficiency improvements we can make to I_HARBOUR1 .
- Translate to the intended target language.

This outline of the method for a particular application we will call a *development plan.* In practice such plans will include a number of other activities for documentation, testing, quality assurance, etc. together with schedules, effort to be used, and so on.

3.5 Type module

From our initial thoughts we formulate the module TYPES:

```
scheme TYPES =
  class
    type
      Ship, Berth,
      Occupancy == vacant | occupied_by(occupant : Ship)
    value
      fits : Ship × Berth → Bool
  end
```

We then make a global object from TYPES:

```
object T : TYPES
```

3.6 Abstract applicative harbour

The method is in summary:

- Define the type of interest as a sort (*Harbour*).
- Define the signatures of the functions we need.
- Categorize these functions as *generators* if the type of interest (or a type dependent on it) appears in their result types and as *observers* otherwise. (We shall see that the imperative counterparts to generators are functions that change (write to) the state. We have previously referred to these as "state-changing".) We find we have three generators: *arrives*, *docks* and *leaves*, and we identify two observers: *waiting* and *occupancy.*

- Formulate preconditions for any partial functions. All three generators are partial: there are situations where they cannot sensibly be applied. We therefore identify three functions (termed "guards") to express their preconditions: *can_arrive*, etc. All these guards are derived from (i.e. given explicit definitions in terms of) the observers.

- Define a function (*consistent*) to express the invariant, making it another derived observer.

- For each possible combination of non-derived observer and non-derived generator, define an axiom expressing the relation between them. We have three non-derived generators and two non-derived observers, so we have six such axioms. These axioms are called *observer-generator axioms.*

- Add axioms expressing the notion that the non-derived generators maintain consistency. We have three such axioms.

This gives the abstract applicative module A_HARBOUR0:

```
scheme
  A_HARBOUR0 =
    class
      type Harbour

      value
        /* generators */
        arrives : T.Ship × Harbour →̃ Harbour,

        docks : T.Ship × T.Berth × Harbour →̃ Harbour,

        leaves : T.Ship × T.Berth × Harbour →̃ Harbour,

        /* observers */
        waiting : T.Ship × Harbour → Bool,

        occupancy : T.Berth × Harbour → T.Occupancy,

        /* derived */
        consistent : Harbour → Bool
        consistent(h) ≡
          (
            ∀ s : T.Ship •
              ∼ (waiting(s, h) ∧ is_docked(s, h)) ∧
              (
                ∀ b1, b2 : T.Berth •
                  occupancy(b1, h) = T.occupied_by(s) ∧
                  occupancy(b2, h) = T.occupied_by(s) ⇒
                  b1 = b2
              ) ∧
              (
                ∀ b : T.Berth •
                  occupancy(b, h) = T.occupied_by(s) ⇒ T.fits(s, b)
              )
          ),
        is_docked : T.Ship × Harbour → Bool
        is_docked(s, h) ≡ (∃ b : T.Berth • occupancy(b, h) = T.occupied_by(s)),

        /* guards */
        can_arrive : T.Ship × Harbour → Bool
        can_arrive(s, h) ≡ ∼ waiting(s, h) ∧ ∼ is_docked(s, h),

        can_dock : T.Ship × T.Berth × Harbour → Bool
        can_dock(s, b, h) ≡
          waiting(s, h) ∧
          ∼ is_docked(s, h) ∧ occupancy(b, h) = T.vacant ∧ T.fits(s, b),

        can_leave : T.Ship × T.Berth × Harbour → Bool
        can_leave(s, b, h) ≡ occupancy(b, h) = T.occupied_by(s)

      axiom
        [ waiting_arrives ]
          ∀ h : Harbour, s1, s2 : T.Ship •
            waiting(s2, arrives(s1, h)) ≡ s1 = s2 ∨ waiting(s2, h)
              pre can_arrive(s1, h),

        [ waiting_docks ]
          ∀ h : Harbour, s1, s2 : T.Ship, b : T.Berth •
            waiting(s2, docks(s1, b, h)) ≡ s1 ≠ s2 ∧ waiting(s2, h)
              pre can_dock(s1, b, h),

        [ waiting_leaves ]
          ∀ h : Harbour, s1, s2 : T.Ship, b : T.Berth •
            waiting(s2, leaves(s1, b, h)) ≡ waiting(s2, h) pre can_leave(s1, b, h),

        [ occupancy_arrives ]
          ∀ h : Harbour, s : T.Ship, b : T.Berth •
            occupancy(b, arrives(s, h)) ≡ occupancy(b, h) pre can_arrive(s, h),

        [ occupancy_docks ]
          ∀ h : Harbour, s : T.Ship, b1, b2 : T.Berth •
            occupancy(b2, docks(s, b1, h)) ≡
              if b1 = b2 then T.occupied_by(s) else occupancy(b2, h) end
              pre can_dock(s, b1, h),

        [ occupancy_leaves ]
          ∀ h : Harbour, s : T.Ship, b1, b2 : T.Berth •
            occupancy(b2, leaves(s, b1, h)) ≡
              if b1 = b2 then T.vacant else occupancy(b2, h) end
              pre can_leave(s, b1, h),

        [ consistent_arrives ]
          ∀ h : Harbour, s : T.Ship •
            arrives(s, h) as h′
              post consistent(h′)
              pre consistent(h) ∧ can_arrive(s, h),

        [ consistent_docks ]
          ∀ h : Harbour, s : T.Ship, b : T.Berth •
```

```
            docks(s, b, h) as h′
              post consistent(h′)
              pre consistent(h) ∧ can_dock(s, b, h),

        [consistent_leaves]
          ∀ h : Harbour, s : T.Ship, b : T.Berth •
            leaves(s, b, h) as h′
              post consistent(h′)
              pre consistent(h) ∧ can_leave(s, b, h)
    end
```

In practice it will typically take several iterations before such a specification can be settled on. In particular, while the generators may be reasonably apparent from the requirements (ships can arrive and dock, etc.) it is often much less clear what good observers will be. Do we, for example, want one for the set of ships waiting? If one tries to use this as an observer it should soon become apparent that it can easily be defined as a derived observer from the simple observer *waiting* that we have used.

We have also omitted a constant of type *Harbour*, like *empty*. This is partly because the requirements were silent about initial conditions. In practice the ability to initialise (and perhaps reset) the system is a likely requirement and an *empty* constant would be required. Adding *empty* (and making the consequent changes in the remainder of the development) is left as an exercise for the reader.

3.6.1 Validation

Validating an initial specification means checking that it meets the requirements. In practice there are usually some requirements that are not expressed in the initial specification. These may be either

- requirements that cannot be expressed in RSL, the "non-functional" requirements, or
- requirements we have decided to defer because they are too detailed to include yet

Both these kinds of requirements will give direction to the development because we will need to deal with them at some point.

So validation means checking, for each requirement we can identify, that it is either correctly reflected in the initial specification or can be dealt with at some stage in the development plan. If we consider some of the requirements for the system, we can record:

1	Ships can arrive and will be registered.	A_HARBOUR0
2	Ships can be docked when a suitable berth is free.	A_HARBOUR0
3	Docked ships can leave.	A_HARBOUR0
4	Ships can only be allocated to berths they fit.	A_HARBOUR0
5	Any ship will eventually get a berth.	outside system
6	Any ship waiting more than 2 days will be flagged.	deferred to ...

We could of course give more precise references to requirements we believe to be met. Thus number 4 could have a reference to *can_dock*.

If we claim to meet a requirement but the claim is not immediate from the specification, we can formulate the requirement as a theorem and justify it.

This process will sometimes raise issues that we have not dealt with properly, causing us to rework the specification. We have assumed, for example, that the actual choice of a ship to fill a vacant berth is outside the system: we just provide the facilities for a user to make the choice. This may not be correct, or it may require a new function, to return, perhaps, a list of ships that can fit a berth, ordered by date of arrival.

Making such a list of requirements will also give us the opportunity to update the list as we do the development so that we can eventually show that all the deferred requirements are met. Showing where and how requirements are met is commonly called requirements tracing.

3.7 Concrete applicative harbour

A natural concrete type to use for *Harbour* is the product of a set of waiting ships and a map from indices of berths to their occupancy.

Extending TYPES

We could then model the type *Berth* as equal to the subtype of integers from *min* to *max*, but it is more general to leave *Berth* as a sort and say there is a function *indx* from *Berth* to this subtype: effectively the index of a berth is an attribute of it. We then leave open the possibility of other attributes. Presumably there will need to be some others (and some attributes of ships) to enable us to eventually compute *fits*.

We therefore add the following definitions to the type module TYPES:

```
type
   Index = {| i : Int • i ≥ min ∧ max ≥ i |}
value
   min, max : Int,
   indx : Berth → Index
axiom
   [index_not_empty] max ≥ min,
   [berths_indexable]
      ∀ b1, b2 : Berth • indx(b1) = indx(b2) ⇒ b1 = b2
```

The axiom *berths_indexable* ensures that indexes identify berths uniquely.

Note that we choose just to add these definitions to the type module directly rather than develop it to a new module TYPES1, say. This is the most convenient way to develop type modules. As here, the extensions to them are typically conservative and making formal developments of them would be more effort than is appropriate.

Developing the system module

From the abstract applicative module *A_HARBOUR0* we develop a concrete applicative module *A_HARBOUR1* by the following method:

- Give a concrete definition for the type of interest.
- Give explicit definitions of constants and functions.
- Remove axioms.

This gives the concrete applicative module A_HARBOUR1:

```
scheme
  A_HARBOUR1 =
    class
      type
        Harbour =
          T.Ship-set ×
            {| bs : T.Index $\underset{m}{\rightarrow}$ T.Occupancy •
            (∀ idx : T.Index • idx ∈ dom bs)
            |}

      value
        /* generators */
        arrives : T.Ship × Harbour $\overset{\sim}{\rightarrow}$ Harbour
        arrives(s, (ws, bs)) ≡ (ws ∪ {s}, bs) pre can_arrive(s, (ws, bs)),

        docks : T.Ship × T.Berth × Harbour $\overset{\sim}{\rightarrow}$ Harbour
        docks(s, b, (ws, bs)) ≡
          (ws \ {s}, bs † [T.indx(b) ↦ T.occupied_by(s)])
          pre can_dock(s, b, (ws, bs)),

        leaves : T.Ship × T.Berth × Harbour $\overset{\sim}{\rightarrow}$ Harbour
        leaves(s, b, (ws, bs)) ≡
          (ws, bs † [T.indx(b) ↦ T.vacant])
          pre can_leave(s, b, (ws, bs)),

        /* observers */
        waiting : T.Ship × Harbour → Bool
        waiting(s, (ws, bs)) ≡ s ∈ ws,

        occupancy : T.Berth × Harbour → T.Occupancy
        occupancy(b, (ws, bs)) ≡ bs(T.indx(b)),

        /* invariant */
        consistent : Harbour → Bool
        consistent((ws, bs)) ≡
          (
            ∀ s : T.Ship •
              ∼ (waiting(s, (ws, bs)) ∧ is_docked(s, (ws, bs))) ∧
              (
                ∀ b1, b2 : T.Berth •
                  occupancy(b1, (ws, bs)) = T.occupied_by(s) ∧
                  occupancy(b2, (ws, bs)) = T.occupied_by(s) ⇒
                  b1 = b2
              ) ∧
              (
                ∀ b : T.Berth •
                  occupancy(b, (ws, bs)) = T.occupied_by(s) ⇒ T.fits(s, b)
              )
          ),

        is_docked : T.Ship × Harbour → Bool
        is_docked(s, (ws, bs)) ≡
          (∃ b : T.Berth • occupancy(b, (ws, bs)) = T.occupied_by(s)),
        /* guards */
        can_arrive : T.Ship × Harbour → Bool
        can_arrive(s, (ws, bs)) ≡
          ∼ waiting(s, (ws, bs)) ∧ ∼ is_docked(s, (ws, bs)),

        can_dock : T.Ship × T.Berth × Harbour → Bool
        can_dock(s, b, (ws, bs)) ≡
          waiting(s, (ws, bs)) ∧
          ∼ is_docked(s, (ws, bs)) ∧
          occupancy(b, (ws, bs)) = T.vacant ∧ T.fits(s, b),

        can_leave : T.Ship × T.Berth × Harbour → Bool
        can_leave(s, b, (ws, bs)) ≡ occupancy(b, (ws, bs)) = T.occupied_by(s)
    end
```

3.7.1 Verification

We formulate the development relation A_HARBOUR0_1, which asserts that A_-HARBOUR1 implements A_HARBOUR0:

development_relation [A_HARBOUR0_1] A_HARBOUR1 $\preceq$ A_HARBOUR0

A development relation is a named statement of a relation between modules. This one takes the most simple form of the statement of an implementation relation ($\preceq$) between two versions of the harbour module.

Justification of this relation shows that the development step is correct. We use a justification editor to first check the static implementation relation and to generate implementation conditions to be proved (cf. section 2.2). In this case (and typically) the implementation conditions are the axioms from the abstract specification, A_HARBOUR0. Each of the implementation conditions has as context the more concrete specification, A_HARBOUR1.

For instance, we get from the axiom *waiting_arrives* the implementation condition

```
∀ h : Harbour, s1, s2 : T.Ship •
   waiting(s2, arrives(s1, h)) ≡ s1 = s2 ∨ waiting(s2, h) pre can_arrive(s1, h)
```

The justification of this condition is done by a series of steps in which RSL proof rules are applied transforming the condition to new conditions whose truth ensure the truth of the original condition. The conditions, also called *goals*, are enclosed by the brackets ⌊ and ⌋, and the names of the applied proof rules are written between the goals.

An example of a proof rule is

```
[ is_annihilation ]
   e ≡ e ≃ true
```

It has the name *is_annihilation* and states that any value expression of the form $e \equiv e$ is equivalent to **true**. This rule can be used in a proof to replace a value expression $e \equiv e$ with **true** or vice versa. In the handbook [7] there is a collection of proof rules for RSL. From the definitions and axioms in the context of a justification additional proof rules can be derived. These rules are called *context rules*. The context of the condition above, A_HARBOUR1, gives, for example, unfold rules for functions with defined bodies. For instance

```
[ waiting_def ]
   waiting(s, (ws, bs)) ≃ s ∈ ws
```

A justification of the condition is

```
   ⌊∀ h : Harbour, s1, s2 : T.Ship •
      waiting(s2, arrives(s1, h)) ≡ s1 = s2 ∨ waiting(s2, h) pre can_arrive(s1, h)⌋
 all_name_change :
   ⌊∀ (ws, bs) : Harbour, s1, s2 : T.Ship •
      waiting(s2, arrives(s1, (ws, bs))) ≡ s1 = s2 ∨ waiting(s2, (ws, bs)) pre can_arrive(s1, (ws, bs))⌋
 all_assumption_inf :
   ⌊waiting(s2, arrives(s1, (ws, bs))) ≡ s1 = s2 ∨ waiting(s2, (ws, bs)) pre can_arrive(s1, (ws, bs))⌋
 pre_deduction_inf :
 [ assump ] can_arrive(s1, (ws, bs)) ⊢
   ⌊waiting(s2, arrives(s1, (ws, bs))) ≡ s1 = s2 ∨ waiting(s2, (ws, bs))⌋
 arrives_def :
   ⌊waiting(s2, (ws ∪ {s1}, bs)) ≡ s1 = s2 ∨ waiting(s2, (ws, bs))⌋
   since
         ⌊can_arrive(s1, (ws, bs))⌋
      assump :
         ⌊true⌋
      qed
   end
 waiting_def :
   ⌊s2 ∈ ws ∪ {s1} ≡ s1 = s2 ∨ waiting(s2, (ws, bs))⌋
 waiting_def :
   ⌊s2 ∈ ws ∪ {s1} ≡ s1 = s2 ∨ s2 ∈ ws⌋
 isin_union :
   ⌊s2 ∈ ws ∨ s2 ∈ {s1} ≡ s1 = s2 ∨ s2 ∈ ws⌋
 isin_singleton :
   ⌊s2 ∈ ws ∨ s2 = s1 ≡ s1 = s2 ∨ s2 ∈ ws⌋
 or_commutativity :
   ⌊s2 = s1 ∨ s2 ∈ ws ≡ s1 = s2 ∨ s2 ∈ ws⌋
 is_annihilation :
   ⌊true⌋
 qed
```

In the first step we rename the binding to match better the context rules like *waiting_def*. In the second step we assume we have a fixed but arbitrary harbour *(ws, bs)* and fixed but arbitrary ships *s1* and *s2*. In the third step we remove the precondition from the equivalence and instead assume that it is true. We give the assumption the name *assump* so that we can refer to it later in the proof. In the fourth step we unfold the application of the function *arrives* (remembering we are in the context of A_HARBOUR1). This gives a new goal with the application replaced with the body of the function definition, with actual parameters replacing the formal ones. As the function has a precondition it also gives a *side condition* (*can_arrive(s1, (ws, bs))*) which expresses that the precondition is satisfied. The proof of the side condition is placed between the keywords **since** and **end**. The side condition is proved using the assumption *assump*. In the fifth and sixth steps we unfold the applications of the function *waiting*. In the next steps we use various proofrules for sets, and finally, in the last step, we apply the rule *is_annihilation* by which the proof is completed.

3.8 Concrete imperative harbour

From the concrete applicative module *A_HARBOUR1* we develop a corresponding concrete imperative module *I_HARBOUR1* by the following method:

- Declare variables which can contain information of the concrete type of interest.
- Define imperative functions which correspond to the applicative ones. They have the same names. Generators have access **write any** and observers have access **read any**. Occurrences of the type of interest are removed from parameter and result types (and replaced by **Unit** if there are no other components in a parameter or result type).
- Define the bodies of the functions by adapting the applicative versions to use the imperative functions corresponding to the applicative ones.
- Remove the declaration of the type of interest (if not used elsewhere).

This gives the concrete imperative module I_HARBOUR1:

```
scheme
 I_HARBOUR1 =
  class
   variable
    ws : T.Ship-set,
    bs :
     {| bs : T.Index $\underset{m}{\to}$ T.Occupancy • (∀ idx : T.Index • idx ∈ dom bs) |}

   value
    /* generators */
    arrives : T.Ship $\overset{\sim}{\to}$ write any Unit
    arrives(s) ≡ ws := ws ∪ {s} pre can_arrive(s),

    docks : T.Ship × T.Berth $\overset{\sim}{\to}$ write any Unit
    docks(s, b) ≡
     ws := ws \ {s} ; bs := bs † [T.indx(b) ↦ T.occupied_by(s)]
     pre can_dock(s, b),

    leaves : T.Ship × T.Berth $\overset{\sim}{\to}$ write any Unit
    leaves(s, b) ≡ bs := bs † [T.indx(b) ↦ T.vacant] pre can_leave(s, b),

    /* observers */
    waiting : T.Ship → read any Bool
    waiting(s) ≡ s ∈ ws,

    occupancy : T.Berth → read any T.Occupancy
    occupancy(b) ≡ bs(T.indx(b)),

    /* invariant */
    consistent : Unit → read any Bool
    consistent() ≡
     (
    ∀ s : T.Ship •
      ∼ (waiting(s) ∧ is_docked(s)) ∧
      (
        ∀ b1, b2 : T.Berth •
          occupancy(b1) = T.occupied_by(s) ∧
          occupancy(b2) = T.occupied_by(s) ⇒
          b1 = b2
      ) ∧
      (∀ b : T.Berth • occupancy(b) = T.occupied_by(s) ⇒ T.fits(s, b))
  ),

  is_docked : T.Ship → read any Bool
  is_docked(s) ≡ (∃ b : T.Berth • occupancy(b) = T.occupied_by(s)),

  /* guards */
  can_arrive : T.Ship → read any Bool
  can_arrive(s) ≡ ∼ waiting(s) ∧ ∼ is_docked(s),

  can_dock : T.Ship × T.Berth → read any Bool
  can_dock(s, b) ≡
    waiting(s) ∧ ∼ is_docked(s) ∧ occupancy(b) = T.vacant ∧ T.fits(s, b),

  can_leave : T.Ship × T.Berth → read any Bool
  can_leave(s, b) ≡ occupancy(b) = T.occupied_by(s)
end
```

3.8.1 Verification

Since this development step was from applicative to imperative, we need to decide what level of assurance we need for correctness. We can either

- check that the method for this transition has been followed correctly, or
- formulate the imperative axioms corresponding to the applicative axioms from A_HARBOUR0 and justify them for I_HARBOUR1

Both of these are verifications since they check on the correctness of the development process. The first is informal and is almost certainly all that is necessary for this fairly straightforward development. The second is formal and can be done if we have any doubts or require the highest level of assurance of correctness.

3.9 Further development

There are a few issues left to be resolved:

- The definition of *is_docked* still involves an existential quantifier and is probably not translatable yet. So we formulate a development of I_HARBOUR1, I_HARBOUR2, in which *is_docked* is defined by

```
value
  is_docked : T.Ship → read any Bool
  is_docked(s) ≡
    local variable found : Bool := false,  indx : T.Index := T.min in
      while ∼ found ∧ indx ≤ T.max do
        found := bs(indx) = T.occupied_by(s) ; indx := indx + 1
      end ;
      found
    end
```

We can then formulate and justify the refinement relation between I_HARBOUR2 and I_HARBOUR1 to check this is correct.

- We have still left unspecified the types *Ship* and *Berth* and the values *min*, *max*, *fits* and *indx*; all defined in TYPES. In practice we should either have been able to define all of these by getting more detailed requirements, or we could regard them as system parameters to be instantiated for particular harbours.

  When we are in a position to make definite choices for these types and values we can define a new module, TYPES1, say. We then justify that TYPES1 refines TYPES. If it does, we can replace **object** T : TYPES with **object** T : TYPES1. This is left as an exercise for the reader.

References

[1] J.R. Abrial. (1) the specification language Z: Basic library, 30 pgs.; (2) the specification language Z: Syntax and "semantics", 29 pgs.; (3) an attempt to use Z for defining the semantics of an elementary programming language, 3 pgs.; (4) a low level file handler design, 18 pgs.; (5) specification of some aspects of a simple batch operating system, 37 pgs. Internal reports, Programming Research Group, Oxford Univ., Computing Laboratory, April-May 1980.

[2] D. Bjørner and C.B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall International, 1982.

[3] R.M. Burstall and J. A. Goguen. The Semantics of Clear, a Specification Language. In *Proceedings of Advanced Course on Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer-Verlag, 1980.

[4] B. Dandanell, J. Gørtz, J. Storbank Pedersen, and E. Zierau. Experiences from Applications of RAISE: Report 2. In *[17]*, 1993.

[5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, vol. 6, Springer-Verlag, 1985.

[6] K. Futasugi, J. Goguen, J. Jouannaud, and J. Meseguer. Principles of OBJ2. In *12th Symposium on POPL*. Association for Computing Machinery, 1985.

[7] C.W. George and S. Prehn. The RAISE Justification Handbook. Technical Report LACOS/CRI/DOC/7, CRI: Computer Resources International, 1991.

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

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

[10] 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.

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

[12] C.B. Jones. *Systematic Software Development — Using VDM, 2nd Edition.* Prentice-Hall International, 1989.

[13] B. Krieg-Brückner. Algebraic specification and functionals for transformational program and meta program development. In J. Diaz and F. Orejas, editors, *TAPSOFT'89. Vol.2: Adv. Seminar on Foundations of Innovative Software Development II*, pages 36–59. Springer-Verlag, Heidelberg, Germany, 1989.

[14] D.B. MacQueen. Modules for Standard ML. *Polymorphism*, II(2), 1985.

[15] R. Milner. *Calculus of Communication Systems*, volume 94 of *Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, 1980.

[16] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. Technical report, Department of Computer Science, University of Edinburgh, 1985.

[17] J.C.P. Woodcock and P.G. Larsen, editors. *FME'93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe*, volume 670 of *Lecture Notes in Computer Science.* Springer-Verlag, Heidelberg, Germany, 1993.