Formal Aspects of Computing (1989) 1: 85–114
© 1989 BCS

Formal Aspects of Computing

The RAISE Language, Method and Tools

Mogens Nielsen[a], Klaus Havelund[a], Kim Ritter Wagner[a] and Chris George[b]

[a] Corporate Technology Computer Resources International A/S, Vesterbrogade 1A, DK-1620 Copenhagen, Denmark and [b] STC Technology Limited, London Road, Harlow, Essex CM17 9NA, UK

Key words: The RAISE method; Formal software development; Wide-spectrum specification language; Computer based tools for SE

Abstract. This paper presents the RAISE[1] software development method, its associated specification language, and the tools supporting it. The RAISE method enables the stepwise development of both sequential and concurrent software from abstract specification through design to implementation. All stages of RAISE software development are expressed in the wide-spectrum RAISE specification language. The RAISE tools form an integrated tool environment supporting both language and method.

The paper surveys RAISE and furthermore, more detailed presentations of major RAISE results are provided. The subjects of these are (a) an example of the use of the RAISE method and language, and (b) a presentation of the mathematical semantics of the RAISE specification language.

1. Introduction

As described in Prehn [Pre87], the starting point for RAISE[1] is the Vienna development method (VDM) - [BjJ82, Jon86] probably the most widely used "formal" method for software development. Experience from various applications has revealed a number of problems which seem to complicate the use of VDM in full scale industrial software development projects:

1. VDM has until now been a largely paper-and-pencil approach. Real life

[1] RAISE is an acronym for "Rigorous Approach to Industrial Software Engineering".

Correspondence and offprint requests to: M. Nielsen, Dansk Datamatik Center, Lundtoftevej 1C, DK-2800 Lyngby, Denmark.

software development requires a number of powerful, computerised tools supporting the development process.

2. The VDM specification language does not have a satisfactory facility for the specification of concurrency. Many applications need to deal with concurrency, in specification, development and implementation.
3. VDM does not have facilities for modularisation of specification and development in such a way that the development of large software systems can be divided into blocks of a reasonable size, which can then be combined in a well-defined way.
4. The VDM specification language has never been given a satisfactory mathematical semantics. Such a semantics is a prerequisite for a thorough understanding of the language, and for a proof theory allowing reasoning about specifications written in the language.
5. The VDM specification language lacks abstraction facilities. The developments of formalisms for property oriented specification of abstract data types [FGJ85, CIP85, GHW85] have shown the feasibility and usefulness of employing more abstraction than the domain equations in the VDM specification language provide.

These problems have been the motivation for designing a "second generation" formal method for software development. The aim of the RAISE project is to construct a mathematically well-founded software development method, supported by comprehensive computer based tools, which forms an environment for the method and language. RAISE extends and improves VDM in the areas mentioned above. The outcome of the RAISE project is termed the RAISE product and it consists of the following components:

1. The RAISE method for software development
2. The RAISE specification language in which the stages of software development can be expressed
3. The RAISE tools consisting of the tools supporting method and language
4. The RAISE documentation, including manuals and educational material for language, method and tools

RAISE is intended to be used for industrial development of large and complex software systems, an area in which the need for the extensions and improvements of VDM, removing the above mentioned problems, are commonly acknowledged. RAISE is designed to be applicable to the development of a wide variety of software systems. Examples are: embedded real-time systems, network software, database management systems, application generators, expert system generators, operating systems, compilers, and control and robotic systems.

As is the case for VDM, the RAISE focus is on supporting the specification, design, and implementation stages of the software development process. There are, however, important implications on most of the remaining development stages from using RAISE, e.g. the maintenance stage will be improved considerably by the presence of a formal recording of the development of the software system to be maintained.

The development of software using RAISE is a stepwise process in which all stages are expressed in the RAISE specification language (RSL). Each stage in the process is called a specification and represents the knowledge of the problem and its solution at that stage. The number of steps may vary according to the nature of the problem to be solved and the project organisation.

A RAISE specification is often derived from the preceding specification by constraining the description (commitment), reflecting the fact that a degree of freedom or indeterminacy has been removed. A specification can also be constructed from the preceding specification by taking further requirements into account. In the last step, the specification will be transformed into a program written in the programming language chosen for the project.

Within this framework, the use of RAISE can be varied according to the nature of the software project in question and the people involved in the development. RAISE allows the user several styles of expression through a wide-spectrum language with the possibility of implementation in a range of programming languages, and allows the application of a user-defined degree of formalism ranging from systematic, via rigorous, to formal.

In this paper, the main characteristics of the RAISE product are surveyed. It should be noted that RAISE is an ongoing project, and that the method, language and tools might undergo changes during the remaining year and a half of the project. In particular the results so far are being evaluated in "industrial trials" where project partners are applying them to real projects. Their evaluations will be an important input to the final product. The paper is organised as follows: Section 2 surveys the RAISE method and Section 3 the RAISE specification language. These two sections are conceptual, as the method and language is exemplified in Section 5. The mathematical semantics of RSL is discussed in Section 6. Section 4 presents the RAISE tools and finally Section 7 contains information on the RAISE project organisation and discusses the current state of the project.

2. The RAISE Method

In this section we survey the RAISE method. An example of its use is given in Section 5, and for a more complete documentation of the method, see the report by George [Geo88].

2.1. RAISE – A Rigorous Method

The aim of the RAISE method is to enable the construction of reliable software by formalising the software development process. By a formal method we mean a method in which properties of specifications can be mathematically proved and in which the development steps can be mathematically proved to maintain desired, recorded properties. Insisting on complete formality would render the method unsuitable for industrial usage (at present at least). But many of the benefits of formal development can be obtained without actually carrying out the proofs completely, and a good notation for development steps is useful in itself.

Software development can be characterised by a sequence of increasingly strict development styles: (a) ad hoc, (b) systematic, (c) rigorous, (d) formal.

Rigorous methods and formal methods differ from ad hoc and systematic ones, in having a specification language and a notion of development with a mathematical description (semantics), such that specification and development of software can be subject to mathematical reasoning. The process of software development then produces the obligation to prove mathematically the well-definedness and correctness of the specifications written and the development steps carried out. Whereas a formal method would insist on the formal proof of all such obligations, a rigorous method allows a level of formality which fits the

actual situation. This notion of rigour is central to the RAISE method, and allows users to select the level of formality that is appropriate to particular circumstances, project standards, etc. Since there is an underlying mathematical semantics, a correctness argument that is challenged can always be proved in more detail.

2.2. The Development Process

As it is the case for VDM, the RAISE method is based on the notion of stepwise refinement ([Dij76], [Wir71], Jon86]). The basis of stepwise refinement can be summarised as follows:

Software is constructed by a series of steps - it is an iterative procedure.

Each step starts with a description of the software and produces a new one, which is in some way more detailed (or more concrete).

The result of each step is not only more detailed but also in some way conforms to the previous one, so that it can be used to replace it.

Refinement typically involves both algorithm and data, since a change in one normally involves a change in the other.

This basis is taken into account in RAISE developments where initial abstract specifications are successively developed by a process of commitment in which degrees of freedom or indeterminacy are removed. Thus the top level specification is developed to give a more committed specification which in turn may be subject to the same development process. In each step data structures and/or control structures are elaborated. Such elaboration may take place within one specification or it may involve the creation of specifications to be developed separately, but whose combination satisfies the properties posed by the previous specification. Development steps also involves justification that each new specification, or combination, in some sense is a correct development of the previous one.

In RAISE the term *development step* is used for the process of performing each step and *development level* for its result. A development level records the specification constructed, the abstract specification it is a development of, and information about separate developments on which the present level depends. A *development* is a sequence of development levels.

The last development step produces a program or a collection of programs written in the programming language chosen for the project. Since the last development level containing specifications written in RSL should be very implementation oriented, the task of producing a program will be automated or semi-automated.

The software development process using RAISE can be illustrated as in Fig. 1.

Note that the requirement to be able to support such a development process, places several requirements on RSL. It must provide a structuring concept so that specifications can be encapsulated and so that representation details can be hidden. Without the possibility of abstracting from particular representations the ability to provide different implementations and prove them correct would be lost. Secondly, RSL must allow for abstraction and underdeterminacy to be expressed, to allow e.g. design decisions to be postponed until appropriate, and to allow re-use. This leads to such notions as parameterisation, specification by property, and underspecification. Thirdly, the RSL needs a range of definition styles - imperative as well as applicative, concurrent as well as sequential, concrete

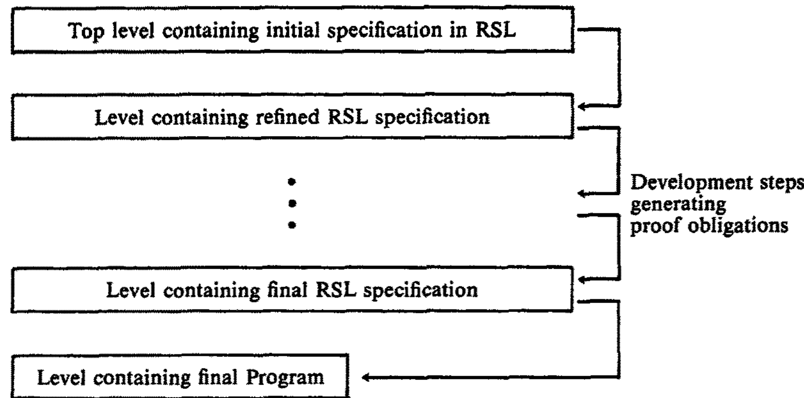

Fig. 1. Raise software development process

types as well as abstract ones, implicit as well as explicit definition of entities - to enable specifications at different levels of development.

2.2.1. *The Initial Specification*

Initial specifications are intended to both capture the functionality of a system (or, for subsidiary developments, of part of it) and at the same time be as abstract as possible, i.e. make as few design decisions as possible. These are frequently contradictory objectives as many statements of requirements give too much detail - they are expressed in terms that assume the system will be designed in a certain way. Thus the early development stages typically involve a mixture of implementing incomplete specifications by making design decisions and enriching them to include more of the functionality, so that there is no specification that is both complete and fully abstract. The initial specification is then taken to be the first one recorded as part of a development. Thus RAISE allows developers some freedom in how much of the early development work is recorded and retained as part of the development history. The overall guidelines are that the initial specification should say something significant about the system, such as its breakdown into major components, and subsequent development steps should also each have a particular purpose in making and recording a particular design decision.

2.3. Managing the Development Process

Since RAISE is concerned with industrial application the method must also cater for "programming in the large", for large projects with many people working in parallel on different parts. Hence there are a number of "software engineering" requirements:

1. It should be possible to record not only the RSL specifications at each stage, but also the relations between them. Such relations take one of three forms:

   a) One specification may be expressed in terms of another, i.e. it may *use* it.

   b) One specification may be recorded as being in some *semantic* relation to another. In particular, one may *implement* another by preserving all its proper-

ties, or there may be some logical statement relating their properties. Implementation is defined formally as theory inclusion (possibly also including a *fitting* to express how named entities in one specification are implemented by entities in another).

c) One specification may be regarded as a *development* of another, whether or not a semantic relation has also been recorded. Ideally there should be such a relation and it should be implementation, but in practice we sometimes need to change properties in development steps.

The first of these forms of relation is recorded in RSL; the RAISE method allows the others to be recorded as well. Together with the second, the semantic relations, we may want to record not only the precise relation but also its (rigorous) proof. Together with the third, the development relations, we may wish to record the purpose behind the development step - what details we are elaborating on, what design decisions are being taken and why, etc. The underlying purpose of such recording is the usual one of recording the design process - to make it manageable. In particular it is intended to help maintenance and reuse. If certain features are to be changed we need to find where the original development needs changing, and what other changes are required as a result.

2. It should be possible to divide systems into pieces, to develop these pieces separately, and then to combine the results again in such a way that the properties of the result are the same as those of the original (the ideal situation, and guaranteed if the developed pieces are implementations of their initial specifications), or bear some known, stated relation to them. This is a natural extension of the notion that formal specification and rigorous development are intended to discover and deal with problems as early as possible. There should be no unpleasant surprises at system integration time!

3. It should be possible to identify key milestones in the development process so that progress can be checked against plans. It seems to be the case that, compared with more traditional models of the development process, the use of formal methods extends the analysis and design phases but shortens the coding and testing phases. If progress is only measured by lines of target code produced the process appears slow and unmanageable.

In order to do development in practice, standard paradigms for development steps, and the conditions under which they produce implementations, are also part of the method description. The method also involves quality assurance procedures applicable to the various activities, notions of how changes of one part affect others, and guidelines for managing the process.

2.4. Proof Obligations

In a RAISE development, proof obligations will arise in two ways.

Firstly, almost any specification on its own will produce proof obligations because it must be shown to be consistent, i.e. have a possible implementation. It must be shown, for example, that a partial function is not called with a parameter outside its domain of definition. Such obligations are not in general decidable and will not always be provable by tools. However, the tools are capable of identifying what needs to be proven in the form of proof obligations for the user to discharge. Simple checks such as ordinary type checking will be carried out by the tools.

Secondly, we have noted that development involves proving the existence of certain development relations between specifications. The RAISE environment allows the recording of such relations together with the appropriate proof obligations and whatever proofs are given to discharge them.

As discussed earlier, the notion of rigour in RAISE entails the user being able to choose an appropriate level of formality in discharging each proof obligation. This may range from a completely informal statement that the proof is "obvious" or "immediate", over sketching some reasons, to a completely formal proof. To support users in recording proofs in such a range of styles, a proof is regarded as a formal object in RAISE. This will enable tools to assist users in the development and presentation of proofs.

2.5. The Role of Transformation

Note that the notion of stepwise development in RAISE may be distinguished from design by transformation. In RAISE there is a cycle of construction (which generates proof obligations) followed by justification (which involves discharging these obligations). Thus each successive specification must be shown to implement the previous one. Implementation is in this way justified post hoc.

By a transformational design method is generally meant one in which each step is a transformation where the justification is included in the step. Part of this justification is a priori - the transformation has been shown to be correct - and part is usually particular to the specification to which it is being applied, since transformations are frequently only correct if certain semantic conditions are met (such as associativity of an operator).

Since transformational design frequently involves proving the applicability of a transformation, the difference between transformational design and the RAISE method may be small; they share the notion of stepwise development. There is a difference, however, in that the user perceives them differently. In RAISE the use constructs the next development step; in a transformational system the user selects a transformation and then applies it.

The decision not to attempt a transformational design system is purely pragmatic. We suspect that future methods may well be based on a transformation paradigm, and there are projects like CIP [CIP85] actively pursuing this approach. There are, however, problems of providing a sufficiently general and complete set of transformations, and of helping users choose the appropriate ones. It seems unlikely that this approach is adequate for RAISE, which is intended for use in large-scale industrial projects.

3. The RAISE Specification Language

In this section, the basic concepts of the RAISE specification language (RSL) are described. Examples of specifications in RSL are shown in Section 5 and the mathematical semantics of RSL is discussed in Section 6. A more complete documentation of RSL can be found in the report [JPC88]. Here, we survey the language from a conceptual point of view.

The starting point for RSL was experience with model based approaches to specification, mainly gained from the VDM meta language [BjJ82]. However, RSL derived inspiration from many other sources, the most prominent being

Clear [BuG77, BG80], OBJ [FGJ85], ML [HMM86], CSP [Hoa85], and Occam [Inm84].

RSL is a wide-spectrum language, complete with facilities for structuring and concurrency; it offers facilities for implicit and explicit specifications, as well as the possibility of expressing these specifications applicatively or imperatively. We therefore claim that RSL is a major improvement compared with the VDM meta language, removing the criticism mentioned in the introduction.

RSL is intended to provide mathematical abstractions for functional specifications of software. Therefore certain aspects of the description of software systems are outside the scope of RSL. Among these are performance requirements and real-time constraints. However, through the method and tools, such requirements can be recorded and associated with the relevant part of a specification or development for later verification. The RAISE proof system will, however, not be able to support such verification.

Software developed using RAISE will, in many cases, be part of a system containing components which were not developed using RAISE. To achieve integration in these cases, the developer will need to construct interface descriptions in RSL. Such interfaces express the system characteristics on which a piece of software depends. RSL has facilities for abstract description of interfaces.

3.1. Structures

The fundamental structuring unit in RSL is the *structure*. Structures are the building blocks and abstraction units of RSL. Structures constitute the frame in which the RSL entities *types*, *values*, *variables*, *operations* and *processes* are defined. Semantically a structure consists of a *signature* associating type information to the entities in the structure, and a class of *models* where each model is an association of mathematical objects to the entities in the structure. A model constitutes an environment in which the expressions and statements of the structure can be interpreted. The reason for having a model class rather than just a single model is the possible presence of *underspecification*. Each model can be considered as an abstraction of one possible implementation.

Structures can be defined in a number of ways in RSL:

As a *flat structure* which is just the encapsulation of a number of entities (types, values, variables, operations, and processes)

As a *layered structure* which uses previously defined structures to define new entities. A layered structure can be considered as *parameterised* with respect to the structures on which it is building, due to the construct for defining structures by structure substitution

As a result of *structure substitution* where a structure S_2 is derived from a structure S_1 by substituting another structure for one that is used by S_1

As a *fitted structure* which is the result of renaming, hiding, and copying entities in a structure

3.2. Types

Semantically, types should be thought of as characterising non-empty sets of values. RSL provides two styles of type definitions:

1. *Abstract type definitions*, where a type is named without explicitly indicating which values it characterises. These are characterised via the entities (constants, functions, operations and processes) which operate on the type.
2. *Explicit type definitions*, where a type is defined by expressing its equivalence with a type expression. The basic type equivalence is structural, but name equivalence can be obtained through *labelled type definitions*, where a type is defined whose values are labelled copies of the values characterised by a type expression. Recursive type definitions in the style of VDM are allowed.

Type expressions can be either names denoting predefined or abstract types (e.g. integers, Booleans, and characters), or type constructions where the type is constructed from other types (e.g. Cartesian products, disjoint sums, and function spaces), or subtypes of other types consisting of values satisfying additional constraints.

3.3. Values

A *value definition* serves to name a value and to state its type. Values can either be defined explicitly as the semantic meanings of *value expressions*, or they can be characterised implicitly using axioms.

RSL has a rich language of value expressions, including value expressions corresponding to atomic values, to composing and decomposing values of constructed type, applying operations and functions to other values, to expressing values using locally defined entities, and to conditionals.

A subset of the value expressions corresponding to the expressions resulting in values of Boolean type constitute the language of axioms. This is a very powerful language for property oriented specification, offering most of what is known from predicative specification languages.

3.4. States and Operations

A structure may introduce a *state* through the declaration of variables, and different instantiations of the state of a structure can be created through copying of such a structure. A state is an association of values to variables. The state of a structure is not directly accessible outside the structure. Instead, manipulations of the state of a structure are performed by calls of *operations* defined in the structure.

In addition to the value expressions which can be used for specifying value returning operations, RSL has a language of *statements* for specifying proper operations. There are statements assigning values to variables, composing other statements, applying operations to argument values, statements defining operations using locally defined entities, and loop statements.

Additionally, operations can be specified axiomatically by pre- and post-conditions.

3.5. Processes

Parallel activities can be specified via the RSL *process* concept. It is based on communicating sequential processes (CSP) [Hoa85] and the language Occam [Inm84].

A process can be considered as an entity capable of (1) communicating with other processes along (uni-directional) *channels*, and (2) accessing variables. The semantics of processes is based on the failure set model of [Ros84].

Processes can be specified by the statements described above and additionally the so-called *process statements*. These include atomic processes (SKIP, STOP, RUN, CHAOS), communication between processes, choices between processes, parallel composition of processes, interleaving of processes, hiding of communications, application of parameterised processes, renaming of channels, and alphabet extensions.

4. The RAISE Tools

Support tools play an important role in RAISE. The aims of the RAISE tools are to provide a software development environment specifically supporting the RAISE method and language, and to interface to tools supporting the software development aspects outside the scope of RAISE. In this section we concentrate on outlining the facilities of the final RAISE tools.

The RAISE tools will be of professional quality and will include documentation enabling its maintenance (and adaptation to specific organisations); since the tools are developed using the RAISE method, the documentation will be in the RAISE style.

The tools are primarily designed to run on SUN[2] workstations, and will be based on the UNIX[3] operating system, and the X Window System[4] is chosen as the screen interface. There will be rudimentary support for the interface of character terminals. The language related components will all be based on the Cornell Synthesiser Generator (CSG) [ReT87]) which provides the RAISE tools with a uniform user interface. The RAISE tools will use LATEX ([Lam86]) as a document preparation system.

4.1. The RAISE Database

The basis of all tools is the RAISE database. This allows recording of information about RSL structures, semantic relations between structures, and RAISE developments. It is capable of recording all parts of RAISE developments, and additionally it can be adapted so that information specific to a given RAISE application area or an organisation can be stored. The database responds to commands of the usual sort - add, delete, rename, etc. - and it handles version control. It allows several users to work against it at the same time.

A number of tools have a close relation to the database:

The *configuration control tool* which provides facilities for connecting and maitaining related objects (in RAISE this includes RSL structures, relations between

[2] Trademark of SUN Microsystems.
[3] Trademark of Bell Laboratories.
[4] Trademark of Massachusetts Institute of Technology.

structures, and developments). The tool will provide possibilities for both static and dynamic building of configurations.

The *database browser* which can be used to display the contents of the database in various ways. The Browser provides a query language facility allowing users to define which information should be extracted from the database.

The *change propagation tool* which can be used to (a) propagate changes in the database, and (b) analyse the effects of changes.

4.2. The RSL Editor

Another basic tool is the RSL editor. It is a syntax-directed editor, including visibility- and type-checking. There are facilities for multi-window, multi-buffer editing. The editor unparses with extended character sets, such that the RSL specifications shown on the screen, are identical to the ones prepared by LATEX (allowing boldface characters, underscoring, mathematical symbols etc.). As mentioned the editor is constructed using CSG and with XWindows as the screen interface.

Closely related to the editor is the *Text formatting tool* which, for the structures created by the editor, generates LATEX output suitable for incorporation in other documents prepared in LATEX.

4.3. The Proof Tools

The proof tools are the part of the RAISE tools which supports reasoning about properties of RSL specifications and RAISE developments. Together, the proof tools constitute an interactive assistant for theorem proving and specification transformation, with some limited facilities for automatic theorem proving.

The proof tools consist of:

The *proof editor* with proof well-formedness checks and automatic proof simplification. The editor is syntax directed over the syntax of RAISE proofs and theorems. It is generated by CSG.

The *RSL context condition tool* which performs semantic analysis of specification correctness, and generates proof obligations for checks which cannot be carried out automatically. This tool can also be considered part of the RSL editor, as it extends the checks on RSL specifications, carried out there (syntax, visibility, type, etc. checks).

The *transformation tool* which supports transformation of RSL specifications according to a set of transformation rules. As mentioned previously (Section 2) only a few simple rules are defined in RAISE, but the tool will be extensible with respect to new transformation rules. The tool generates proof obligations when the proof of applicability of a transformation cannot be carried out automatically.

The *semantic relation tool* which supports establishment of semantic relations between RSL structures. The tool generates proof obligations when checks, that a certain relation (e.g. an implementation relation) exists between structures, cannot be carried out automatically.

4.4. The Translation Tools

The translation tools support the translation from an executable, implementation oriented subset of RSL to various programming languages. The translation tools generate target code of a high quality and is not to be considered primarily as prototying tools.

The basic translation tool will be generic and translators to C, Modula-2, and possibly Ada will be constructed as instantiations of the generic translation tool.

5. An Example Development – Evaluating Reverse Polish Expressions

This section aims to describe the main features of the RAISE development method by means of a fairly simple example - the evaluation of reverse Polish expressions using a stack. The example is small to allow it to be developed within the confines of a short paper. It might be part of a much larger system. While describing the method as applied to the example, some features of RSL are also illustrated.

First we give a brief note for those not familiar with the example. An expression like "(1+2) * 3" may be converted into a "reverse Polish" expression which is a list, in this case "⟨1, 2, +, 3, *⟩". The reason for the conversion is that such a list can now be evaluated using a stack. The list is read from left to right. Any number is pushed onto the stack; any operator is applied to the top two elements on the stack, the two values are removed from the stack, and the result pushed onto it. At the end the number at the top of the stack is the result of the evaluation.

5.1. Planning the Example

We can define the following milestones in the development:

The evaluation of expressions

The generation of reverse Polish lists and the applicative evaluation of such expressions using an applicative stack

An imperative version of the previous milestone

5.2. Evaluating Expressions

We start with a structure EVAL0 (Fig. 3) defining the function *eval* to evaluate expressions. We use a subsidiary structure CALC (Fig. 2) defining the type *Exp* of expressions and the function *calc*. For simplicity we take expressions with operators *plus* and *minus* only.

Some comments on the features of RSL displayed so far are in order. An RSL specification takes the form of a *structure*, which may in turn use other structures, just as EVAL0 uses CALC. Other possible constituents of structures include *type* and *value* definitions. In CALC are defined two types, *Exp* and *Op*. Each are union types, where a union type is expressed as a collection of (tag : type) pairs (as in *Exp*) or just of tags (as in *Op*). Thus an expression of type *Exp* is either a simp(le) expression which is an integer, or a comp(ound) expression which is a Cartesian product of two expressions of type *Exp* and an operator of

```
CALC = structure
    type
      Exp = [|simp:Int,comp:Exp × Op × Exp|],
      Op = [|plus,minus|]
    value
      calc (l:Int,opn:Op,r:Int) ≜
        match opn with
          [|plus|] then l+r
          [|minus|] then l−r
        end
  end CALC
```

Fig. 2. The structure CALC

```
EVAL0 = structure
    use CALC
    value
      eval (e:Exp):Int ≜
        match e with
          [|simp = i|] then i,
          [|comp = (l,opn,r)|] then calc(eval(l),opn,eval(r))
        end
  end EVAL0
```

Fig. 3. The structure EVAL0

type *Op*. An operator is either *plus* or *minus*. In CALC is also defined a function *calc* used to evaluate a pair of integers and an operator. Note how the constituents of a union type (in this case *Op*) may be discriminated by a match expression. Another match expression is employed in the definition of *eval* in the structure EVAL0. The match is used here not only to discriminate the constituents in the union but also to bind identifiers to the component values of the matched value.

5.3. Generating Reverse Polish Expressions

To show that the function, *eval_rp*, say, that evaluates reverse Polish expressions is equivalent to *eval* we will need to define the function, *make_rp*, say, to convert from expressions to reverse Polish expressions. We can then prove that the composition of *eval_rp* and *make_rp* computes the same result as *eval*. We first, then, define reverse Polish expressions in a new structure RP (Fig. 4).

A reverse Polish expression is represented as a non-empty list (indicated by the type "Rp_el$^+$") of either integers or operators. "^" is the concatenation symbol and "⟨···⟩" is the list constructor.

```
RP = structure
    use CALC
    type Rp_el = [|intel:Int,opel:Op|]
    value
      make_rp (e:Exp) :Rp_el⁺ ≜
        match e with
          [|simp = i|] then ⟨[|intel = i|]⟩,
          [|comp = (l,opn,r)|] then make_rp(l)^make_rp(r)^⟨[|opel = opn|]⟩
        end
  end RP
```

Fig. 4. The structure RP

5.4. An Abstract Stack

Evaluation or reverse Polish expressions requires a stack. We can define a stack abstractly in the structure STACK0 (Fig. 5). Several things are worth noting about STACK0. Firstly, we have used a structure ELEMENT that does nothing other than define a type name *El* (Fig. 6).

```
STACK0 = structure
           use ELEMENT
           type
             Stack,
             Stack1 = those st:Stack . ~is_empty(st)
           value
             empty: Stack,
             is_empty: Stack → Bool,
             push: El × Stack → Stack1,
             pop: Stack1 → Stack,
             top: Stack1 → El
           axiom
             ∀ st:Stack, x:El .
               pop(push(x,st)) = st ∧
               top(push(x,st)) = x ∧
               is_empty(empty) ∧
               ~is_empty(push(x,st))
         end STACK0
```

Fig. 5. The structure STACK0

```
ELEMENT = structure
            type El
          end ELEMENT
```

Fig. 6. The structure ELEMENT

We can regard STACK0 as parameterised, since we can substitute for the use of ELEMENT the use of any structure that implements it. In particular we shall want stacks of integers. But the development of STACK0 in which we introduce various representations of stack (as lists, arrays or whatever) can proceed quite separately from developments of other structures using stacks of integers, etc.

Secondly, we note that the type *Stack* in STACK0 has no definition - we say it is "abstract" as opposed to our previous types like *Exp* which are "concrete". When employing such abstract types we typically use axioms rather than definitions to define the values (including functions) involving these types.

Lastly, note we have used the "subtype" notion (**those** ···) to define the type *Stack1* of non-empty stacks. This allows us to record that *pop* and *top* are defined for all non-empty stacks and that *push* produces a non-empty stack.

5.5. A Concrete Stack

We can also define a structure STACK1 using a concrete type definition for *Stack* as a list of elements (Fig. 7).

We can easily show that STACK1 implements STACK0 (it amounts to little more than showing the axioms in STACK0 are true in STACK1) and hence that any structure using STACK0 can safely use STACK1 instead. So why do we

```
STACK1 = structure
          use ELEMENT
          type
            Stack = El*,
            Stack1 = those st:Stack . ~is_empty(st)
          value
            empty:Stack = ⟨ ⟩,
            is_empty (st:Stack) :Bool ≜ st = empty,
            push (x:El,st:Stack) :Stack1 ≜ ⟨x⟩^st,
            pop (st:Stack1) :Stack ≜ tl st,
            top (st:Stack1) :El ≜ hd st
        end STACK1
```

Fig. 7. The structure STACK1

produce two versions? The abstract STACK0 is useful when used in other structures. They can use the properties expressed in the axioms without having to be concerned with any particular representation of the data structures that will eventually be developed as the stack implementation. Hence we describe STACK0 as the "view" of the stack development. This development can now proceed quite independently of that of the evaluator. It is in general the case that axiomatic specifications like STACK0 are harder to write than those like STACK1, particularly for those brought up in the VDM tradition. But it is possible, of course, to formulate STACK1 first and then abstract from it to produce STACK0 as containing precisely those properties that the stack developer intends to maintain. It should also be the case that such axiomatic specifications will be written comparatively rarely because they are easy to re-use. Indeed we would expect in practice to find our stack development in the RAISE library.

5.6. Evaluating Reverse Polish Expressions Applicatively

We are now ready to formulate the structure EVAL1 defining the function *eval_rp* and a new *eval* function, hopefully equivalent to the original one in EVAL0 (Fig. 8).

There are two things to notice about EVAL1. Firstly, the **use** clause

ST = STACK0 **with** INT **providing fit Int for** El **in** ELEMENT

```
EVAL1 = structure
          use
            RP,
            ST = STACK0 with INT providing fit Int for El in ELEMENT
          value
            eval_rp (st:Stack, rp:Rp_el*) :Int ≜
              match rp with
                ⟨⟩ then top(st),
                ⟨[|intel = i|]⟩^t then eval_rp(push(i,st),t),
                ⟨[|opel = opn|]⟩^t then
                  let v = calc(top(pop(st)),opn,top(st))
                  in eval_rp(push(v,pop(pop(st))),t) end
              end omit,
            eval (e:Exp) :Int ≜ eval_rp(empty,make_rp(e))
        end EVAL1
```

Fig. 8. The structure EVAL1

means that instead of using STACK0 we are using a structure ST obtained by substituting the built-in structure INT for ELEMENT. At the same time we use a "fitting" (**fit** ⋯) to replace the type name *El* with the name of the built-in type **Int**. Thus we have obtained integer stacks from the generic ones. Secondly, note that the definition of *eval_rp* is regarded as local to EVAL1 and so is "omitted". A structure using EVAL1 can mention *eval* but not *eval_rp*.

We can now prove that EVAL1 implements EVAL0, because EVAL1'eval (the full name of the "eval" defined in EVAL1) computes the same result as EVAL0'eval. (A rigorous proof can be found in George [Geo88].) Thus we have completed our second milestone.

5.7. Evaluating Reverse Polish Expressions Imperatively

We will firstly need our stack developers to provide an imperative stack structure IMP_STACK0 (Fig. 9).

Note that we have used STACK0 in IMP_STACK0 so that we can define the imperative stack in terms of the applicative one. We have also added an operation *tpop_op* that both pops the stack and returns the old head value - this is useful in our imperative evaluator IMP_EVAL0 (Fig. 10).

```
IMP_STACK0 = structure
                use STACK0
                variable stack : Stack
                operation
                  empty_op write stack is stack := empty end,
                  is_empty_op :Bool read stack ≜ is_empty(stack) end,
                  push_op (x:El) write stack is stack := push(x,stack) end,
                  pop_op write stack pre stack ≠ empty is stack := pop(stack) end,
                  top_op :El read stack pre stack ≠ empty ≜ top(stack) end,
                  tpop_op :El write stack pre stack ≠ empty ≜
                    let r = top(stack) in stack := pop(stack) return r end
                  end
              end IMP_STACK0
```

Fig. 9. The structure IMP_STACK0

It is not the case that IMP_EVAL0 implements EVAL0 because we have changed from an applicative function *eval* to an imperative operation *eval_op*. But there is, we hope, a relation between the function and the operation - they should return the same value. So we want to record and prove the semantic relation between IMP_EVAL0 and EVAL0, namely

$$\forall\ e{:}Exp\ .\ eval_op(make_rp(e)) = EVAL0'eval(e) \qquad (1)$$

We describe how this is done in Section 5.9.

5.8. A Stack Process

There is insufficient space in this paper to describe the features of RSL dealing with concurrency, but to give at least a flavour of what is involved, Fig. 11 shows a stack process.

The concurrency features of RSL are based on those of CSP. The stack process has two input channels *empty* and *push*, the second of which also inputs a value,

```
IMP_EVAL0 = structure
              use
                RP,
                S = IMP_STACK0 with INT providing fit Int for El in ELEMENT
              operation
                eval_op (rp:Rp_el+) :Int write S ≙
                  block
                    variable rpl:Rp_el* := rp, x,y :Int
                  in loop
                      match hd rpl with
                        [|intel = i|] then push_op(i),
                        [|opel = opn|] then
                              x := tpop_op;
                              y := tpop_op;
                              push_op(calc(y,opn,x))
                      end;
                      rpl := tl rpl
                    until rpl = ⟨⟩
                    return tpop_op
                  end
                end eval_op
            end IMP_EVAL0
```

Fig. 10. The structure IMP_EVAL0

```
PROC_STACK0 = structure
                use ST = IMP_STACK0
                process
                  stack in [|empty,push:El|] out [|tpop:El|] write ST
                  is while true loop
                        empty? → empty_op;SKIP
                        ▯
                        push?x → push_op(x);SKIP
                        ▯
                        when ~is_empty_op do tpop!tpop_op → SKIP
                     end
                  end stack
              end PROC_STACK0
```

Fig. 11. The structure PROC_STACK0

and an output channel *tpop* that outputs a value. The process *stack* is a non-terminating loop that in each cycle can set the stack to empty or push a value on to it or, provided it is not empty, pop the stack and output the previous top value.

Note the form of use clause employed in this structure. If we had written "**use** IMP_STACK0" we would have shared IMP_STACK0 with any other structures having a similar use clause. When we write instead "**use** ST = IMP_STACK0" we make a local copy of IMP_STACK0, called ST, that can only be accessed in PROC_STACK0 or structures using it in turn. Thus in this case we have prevented our stack being shared accidentally with any other processes.

5.9. Recording the Development

We will not take this example any further in this paper, but we will consider how the collection of structures formulated so far are organised in the RAISE library.

Firstly, of course, all the structures are stored against their names, so that they can be referred to in other places, such as in use clauses. Secondly, we want to record some semantic relations between structures. Thirdly, we want to record the development relations.

5.9.1. Recording Semantic Relations

You will recall that semantic relations are relations between the properties of structures. The relation may be the general one of implementation, or it may be one or more particular properties like (1) above. For each relation we can record the name of the source structure, the name of the target structure (where if A implements B then A is the source and B the target), the fitting (if any), the property or properties being asserted, and the (rigorous) proof. We have established those shown in Table 1.

Each of these relations may be stored as an item in the library, indexed by the (source, target) pair.

Table 1. Semantic relations

Source	Target	Fitting	Property
INT	ELEMENT	Int for El	Implementation
STACK1	STACK0	—	Implementation
EVAL1	EVAL0	—	Implementation
IMP_EVAL0	EVAL0	—	∀ e:Exp . eval_op(make_rp(e)) = EVAL0'eval(e)

5.9.2. Recording Development Relations

We have assumed for our example that there are two separate developments, one for stacks and one for the evaluator. We want to record the sequence of development steps, the rationale behind the step and the connections, called "contracts" between developments. We also need to distinguish between structures used as "views", which are the specifications seen by other developments while the development work is proceeding, and structures used as "bodies", which contain the current state of the developing specification. Developments will also have a set of requirements to be met, by which we mean an informal statement of both the non-functional and functional requirements. For a main development these requirements will be the original system requirements; for subsidiary developments (like that for the stack) they will be the requirements relevant to the subsidiary development. To capture all this, the RAISE library contains named objects called "developments". These consist of the set of requirements and a sequence of "levels". Each level consists of a triple - a "body" (which is a structure name), a "view" (also a structure name) and a (possibly empty) set of "contracts" (which are names of other developments on which this development is immediately dependent). There is a requirement that the body and view are either the same or the source and target of a semantic relation which is an implementation. This is necessary because it must be possible for structures in developments having a contract with this one to be able to substitute a body for a view. There is no necessary relation between the bodies and views of successive

levels, but the same view appearing in them suggests that implementation is being maintained; a different view suggests that implementation has not been maintained and some change in contracts may be necessary.

In our example there are two developments, which we shall call STACK_DEV (Table 2) and EVAL_DEV (Table 3) respectively.

Table 2. The development STACK_DEV

Level	Body	View	Contracts
1	STACK1	STACK0	{ }
2	IMP_STACK0	IMP_STACK0	{ }

Table 3. The development EVAL_DEV

Level	Body	View	Contracts
1	EVAL0	EVAL0	{ }
2	EVAL1	EVAL0	{STACK_DEV}
3	IMP_EVAL0	IMP_EVAL0	{STACK_DEV}

Note that it is always possible to make the first level have the same body and view, as with EVAL_DEV, if we want to establish the development with its name, requirements and initial view before we have actually done any development of that view.

This notion of developments is quite separate from the other relations in the library; projects may establish as many or as few separate developments as they find convenient. For example, we could have established IMP_STACK_DEV as a separate development for imperative stacks, with a contract to STACK_DEV . IMP_STACK_DEV would then have appeared as the contract in level 3 of EVAL_DEV instead of STACK_DEV.

6. The Semantics of the RAISE Specification Language

This section presents some of the principles used to give a semantics to RSL. The semantics is written in a combination of a transformational and denotational style, where certain constructs are transformed into a kernel, which is then given a denotational semantics. The kernel is, however, quite large compared with the complete RSL and the semantics is thus mainly written in a denotational style.

This approach differs somewhat from an earlier attempt, where we attempted to base the semantics on a very small, purely applicative, kernel language. The transformation of processes into this kernel, however, became too complex, and we seemed to lose the advantage of a small kernel language: clarity.

This section is organised as follows. Section 6.1 presents an applicative subset of RSL, large enough to illustrate the major semantic techniques used in the semantics of RSL. Section 6.1 contains the syntax of the RSL subset, an example of a specification and a summary of intrinsic characteristics. Then follows an outline of the semantics of the RSL subset in the form of the semantic domains and the major semantic functions. We then illustrate the semantics of the example

in Section 6.1.2. Finally, we discuss the semantics of types and the choice of logic, and we give the semantic domains for states, operations and processes in RSL.

6.1. An RSL Subset

6.1.1. Syntax

The syntax is written in a BNF-like formalism:

```
1.  spec            ::=id where struc-def-list
2.  struc-def-list  ::=struc-def
 .2                  | struc-def struc-def-list
3.  struc-def       ::=id=struc
4.  struc           ::=structure
                              def-list
                          end
 .2                  |...
5.  def-list        ::=def
 .2                  | def def-list
6.  def             ::=use id
 .2                  | type id
 .3                  | type id=type
 .4                  | value id:type
 .5                  | axiom exp
 .6                  | ...
```

A specification (1) is a list of structure definitions (2) together with an identification of a main structure (the name of one of the structures in the structure definition list).

A structure definition (3) names a structure (4), which in the atomic version is build from a list of definitions. A definition (6) can (here) have one of five forms: an import of another named structure (6.1), a type definition (6.2, 6.3), the introduction of a value (6.4) or the restriction of such a value by an axiom (6.5). Note that types can be given without a type equation (6.2) corresponding to sorts in algebraic specification languages.

6.1.2. Example

The following example consists of two structures, where the one (*ELEM*) is imported into the other (*BOXES*).

```
BOXES where
ELEM =
  structure
    type
      Elem
    value
      eq : Elem × Elem → Bool
```

```
    axiom
      ∀ e1, e2, e3 : Elem .
        eq(e1,e1) ∧
        (eq(e1,e2)⇒eq(e2,e1)) ∧
        (eq(e1,e2) ∧ eq(e2,e3)⇒eq(e1,e3))
  end
BOXES =
  structure
    use
      ELEM
    type
      Box
      Boxes = Box*
    value
      empty : Box
      add : ELEM'Elem × Box → Box
      isin : ELEM'Elem × Box → Bool
    axiom
      ∀ e1, e2 : ELEM'Elem . ∀ b : Box .
        isin(e1,empty) = false ∧
        isin(e1,add(e2,b)) = (ELEM'eq(e1,e2) ∨ isin(e1,b))
    value
      exists : ELEM'Elem × Boxes → Bool
    axiom
      ∀ e : ELEM'Elem . ∀ bs : Boxes .
        exists(e,bs) = (isin(e,hd(bs)) ∨ exists(e,tl(bs)))
  end
```

The *ELEM* structure specifies a type *Elem* and an equivalence relation *eq* on the elements of that type. An equivalence relation must be reflexive, symmetric and transitive. These properties are specified by "algebraic equations".

The *BOXES* structure specifies boxes of these elements (a box could for example be represented as a set, a bag or a list) and sequences of such boxes. Note the mixture of sorts (*Box*) and type equations (*Boxes*). Note also how one of the functions (*isin*) is specified in an algebraic style, while the specification of the other (*exists*) looks more like what is usually called a definition.

6.1.3. *Semantic Characteristics*

RSL has, among others, the following semantic characteristics:

The denotation of a specification is the denotation of the main structure, which is the set of models satisfying the axioms of the main structure and the structures used transitively by the main structure.

Name clashes introduced by combining structures (by use clauses) are avoided by prefixing entities from imported structures.

Types can be given as sorts (without a defining equation) as well as with a defining equation. Sorts are defined indirectly through the functions defined over them.

A value is defined by a signature and possibly one or more axioms.

Since recursive function definitions are just special axioms, we do not find least

fixed points for these. A recursive function definition may thus have more than one solution (fixed point).
The logic for interpreting an axiom in a model is a two valued logic with existential equality.

6.2. Semantic Domains

The denotation of a specification is a set of models, where a model is a mapping from identifiers to components:

$$\text{Model} = \text{Id} \xrightarrow{f} \text{Component}$$

A component can be of one of four kinds depending on whether the identifier represents a type, a value, an operation or a process:

Component = Carrier | Value | Operation | Process

The domains *Carrier*, *Value*, *Operation* and *Process* will be elaborated in succeeding sections.

In obtaining the denotation of a specification consisting of a structure definition list and the name of a main structure, the first step is to evaluate the structure definition list to give a set of environments. Each environment maps each structure name to a model:

$$\text{Env} = \text{Id} \xrightarrow{f} \text{Model}$$

6.3. Semantic Functions

The semantics of RSL is given in a denotational style where semantic functions map syntactic objects into semantic (mathematical) objects. Each semantic function is defined by a signature and a defining equation.

The semantics of a specification is as follows:

Spec : spec → Model-**set**

Spec[|id **where** struc-def-list|] =
  let envs = {env : Env | Struc-Def-List[|struc-def-list|]env} **in**
  {env(id) | env ∈ envs}

The first step in constructing the denotation of a specification is to obtain all environments that satisfy the structure definition list. This is perhaps an untraditional way to regard "declarations" when comparing with programming language semantics, where a declaration usually denotes some function taking an environment and giving a new environment. In our approach, which is inspired by [Mon85], a declaration denotes a function that takes an environment and gives a Boolean which is true if and only if the environment satisfies the declaration.

Having obtained the set of environments, a new set of all models denoted by the main structure name in each environment is returned.

The semantics of a structure definition list is just the conjunction of the semantics of the structure definitions in it:

Struc-Def-List : struc-def-list → Env → **Bool**

Struc-Def-List[|struc-def|]env =
  Struc-Def[|struc-def|]env

Struc-Def-List[|struc-def struc-def-list|]env =
  Struc-Def[|struc-def|]env ∧
  Struc-Def-List[|struc-def-list|]env

In the semantics of a structure definition, it should be noted that the structure expression denotes a set of models in the given environment, and that the structure identifier on the left hand side in that environment is supposed to denote one of these models:

Struc-Def : struc-def → Env → **Bool**

Struc-Def[|id = struc|]env =
  env(id) ∈ Struc[|struc|]env

One could say that the equality sign in the syntax is misleading syntax, since it really means "belongs to".

In the semantics of a structure expression, observe the same view on declarations (here *def-list*) as seen above for *struc-def-list*: a definition (whether it is a structure import, a type definition, a value signature or an axiom) denotes a function that takes a model and gives a Boolean which is true if and only if the model satisfies the definition:

Struc : struc → Env → Model-**set**

Struc[|**structure** def-list **end**|]env =
  let ms = {m : Model | Def-List[|def-list|]env m} **in**
  let new-concrete-types = New-Concrete-Types[|def-list|] **in**
  find-least-fixed-points(ms,new-concrete-types)

In the first line, all models satisfying the definitions are obtained. This would be the result, if we were not to find least fixed points for type equations. The next two lines are concerned with exactly this. First the names of those newly introduced types which have been defined by equations are selected. An auxiliary function then throws those models away that do not represent least fixed points.

The semantics of a definition list is the conjunction of the semantics of the definitions it contains:

Def-List : def-list → Env → Model → **Bool**

Def-List[|def|]env m =
  Def[|def|]env m

Def-List[|def def-list|]env m =
  Def[|def|]env m ∧
  Def-List[|def-list|]env m

Definitions have the following semantics:

Def : def → Env → Model → **Bool**

Def[|**use** id|]env m =
  prefix(id,env(id)) ⊆ m

Def[|**type** id|]env m =
  m(id) ∈ Carrier

Def[|**type** id = type|]env m =
  m(id) = Type[|type|]m

Def[|**value** id : type|]env m =
  m(id) ∈ Type[|type|]m

Def[|**axiom** exp|]env m =
  Exp[|exp|]m = <u>true</u>

Note that the environment is only used for giving semantics to use clauses. A sort is just required to denote some arbitrary carrier in our type universe (Section 6.5). A type equation must be satisfied in the obvious way. Note that least fixed points for type equations are found in *Struc* above. An axiom is just a Boolean expression.

The following auxiliary function prefixes all names in a model which have not already been prefixed:

prefix : Id × Model →̃ Model

prefix(id,m) =
  [id'x → m(x) | x ∈ **dom**(m) ∧ ~ prefixed(x)]
  ∪ [x → m(x) | x ∈ **dom**(m) ∧ prefixed(x)]

Avoiding prefixing already prefixed names implies that the prefix of an entity is always just the name of its defining structure. This principle makes sharing of one structure between several structures possible. Note, however, that RSL also allows copying of structures, and in this case entities may get longer prefixes.

The following functions are only defined here by their signature:

Type : type → Model → Carrier
Exp : exp → Model → Value
New-Concrete-Types : def-list → Id-**set**
find-least-fixed-points : Model-**set** × Id-**set** → Model-**set**
prefixed : Id → **Bool**

6.4. Semantics of Example

In this section we illustrate the semantics of the example from Section 6.1.2. We do not show the semantics of the entire specification, but rather the last part of it, viz. the sequence of structure definitions. In the semantic functions this point is reached in the function *Spec*:

Spec[|id **where** struc-def-list|] =
  let envs = {env : Env | Struc-Def-List[|struc-def-list|]env} **in**
  {env(id) | env ∈ envs}

where *envs* is created. It is *envs* we show in the following denotion.

envs maps structure names (here *ELEM* and *BOXES*) to possible models. There is one environment for each possible choice of combination of models for the structures in the *struc-def-list*. Here we have only shown one of the environments, one where the type *Elem* is bound to the natural numbers.

Another choice that is specific for the shown environment is the representation of the abstract type *Box*. *Box* is bound to a function domain (a function being a set of pairs) mapping elements from *Elem* (which are natural numbers in this environment) to a natural number saying how many times the element occurs in

the box. If the element does not occur in a box, however, it is not mapped into 0, but absent from the domain of the function representing the box. The first element shown in *Box* is the box containing two 0s and a 1. Many other representations could have been chosen, as long as they fulfil the axioms. These other choices are present in other environments in *envs.*

There is a difference for *Boxes* which is constructed explicitly. Its representation is fixed as a list of whatever the representation of *Box* elements are.

Note, how the elements from *ELEM* are prefixed with their structure name when imported into *BOXES*, in order to resolve possible name clashes.

```
envs =
  {...,
    [ELEM →
      [Elem → {0, 1, 2, ...},
      eq → {(0, 0), (1, 1), (2, 2), ...}
      ],
    BOXES →
      [ELEM'Elem → {0, 1, 2, ...},
      ELEM'eq → {(0, 0), (1, 1), (2, 2), ...},
      Box → {{(0, 2), (1, 1)}, ...},
      Boxes → {⟨ ⟩, ⟨{(0, 2), (1, 1)}⟩, ...},
      empty → { },
      add → {((5, { }), {(5, 1)}), ...},
      isin → {((5, {(5, 1)}), true), ...},
      exists → {((5, ⟨ ⟩), false), ...}
      ]
    ],
  ...
  }
```

6.5. Semantics of Types

Since we do not find least fixed points of recursive definitions of values, but are happy with the presence in the models of all values satisfying the axioms, we need not order values according to content of information, and types need not be cpos. They are just sets.

Types are different in this respect. We wish that (recursive) type definitions have unique solutions, apart from the part that depends on abstract types (which range over the whole type universe). We have ordered types according to size in a subset ordering. Some type operators are continuous with respect to that ordering, others are not. We allow recursive type definitions with recursion through the continuous type operators, but not through the non-continuous ones. The following type operators are continuous:

Cartesian product
Finite subsets (-**finset**)
Finite lists (*)
Finite, non-empty lists ($^+$)

Finite functions ($\overset{f}{\rightarrow}$)
Union type ([| |])
Record type ({| |})
Optional ([ ])

The type constructors for infinite lists, infinite subsets, total functions and partial functions are not continuous, and types must not be defined recursively through these operators. They can however be used in recursive type definitions if the recursion does not go through them, such as in this example.

type T = T × (**Bool** → (**Bool-set**))

The following, however, is not legal.

type T = T → **Bool**

The semantic domain corresponding to types is called *Carrier. Carrier* is thus the type universe. It is a set of sets of values, and its elements (the carriers) are ordered subsetwise. *Carrier* is closed under arbitrary applications of type operators and recursion through continuous type operators (i.e. under least upper bounds of chains).

6.6. Semantics of Axioms

The denotation of an axiom is a model filter: it maps a model into either **true** or **false** according to whether the model satisfies the axiom or not. Axioms are logical expressions and we have the following semantic function:

Def : def → Env → Model → **Bool**

Def[|**axiom** exp|]env m =
  Exp[|exp|]m = <u>true</u>

The environment is not essential here. It is used for other kinds of definitions (uses).

There is a problem with undefined expressions. In general expressions can be undefined in a model. For instance the expression 1/0 is undefined in every model, and the expression $1/x$ is undefined in models binding x to 0. The question is how to handle such expressions, and our solution (highly inspired by the PROSPECTRA approach [Bre88]) can be sketched in the following way.

1. All user defined functions are strict, so if a user defines a function f: **Int** → **Int** then the expression $f(1/0)$ is always undefined.
2. Predefined operators (like **hd**, **card** etc.) are all strict except **if-then-else**.
3. Boolean expressions are treated separately:

 a) We imagine that every Boolean expression, b say, is understood as if the expression were b = **true**. Together with the notion of equality explained in the next point this ensures that all Boolean expressions in effect evaluate to either **true** or **false**. This means that we can employ a two valued logic for axioms. They are never undefined.

 b) We employ existential equality, so that the expression $x = y$ is **true** if both x and y are defined and equal. Otherwise the expression is **false**.

The example from Section 6.1.2 illustrates some of the points behind the scheme. Consider the axiom for *exists*:

$\forall$ e : ELEM'Elem . $\forall$ bs : Boxes .
exists(e,bs) = (isin(e,**hd**(bs)) $\vee$ exists(e,**tl**(bs)))

According to (3a) we should interpret this axiom as if " = **true**" was appended everywhere a Boolean expression occurs. It is only the innermost occurrences of " = **true**" that make a difference, so we can rewrite the axiom to the following:

$\forall$ e : ELEM'Elem . $\forall$ bs : Boxes .
(exists(e,bs) = **true**) =
((isin(e,**hd**(bs)) = **true**) $\vee$ (exists(e,**tl**(bs)) = **true**))

Now, if *bs* is non-empty, everything works as usual - the " = **true**" means nothing. If however, *bs* is the empty list, **hd**(*bs*) is undefined according to (2), and according to (1), so is *isin*(e,**hd**(*bs*)) for every *e*. With existential equality the expression *isin*(e,**hd**(*bs*)) = **true** then becomes **false**, according to (3b). So does *exists*(e,**tl**(*bs*)) = **true**, and, to make the axiom **true**, *exists*(e,*bs*) must be either undefined or **false** for *bs* = ⟨ ⟩. According to the signature of *exists* in the example, *exists* is a total function, so it can only be **false**. Thus, only models in which *exists*(*e*, ⟨ ⟩) is **false** for every *e* are accepted.

If one wants to allow the axiom for *exists*, then we think that the chosen interpretation is the desired one. However, one could get the feeling that the rather indirect way undefined expressions are treated here suggests a too "clever" style of programming, where undefinedness is treated in a rather subtle manner.

6.7. Operations and Processes

In this presentation we have concentrated on the applicative part of RSL for the sake of brevity. However, in RSL one can specify states, operations on states, and processes, possibly reading and writing states. Processes and operations are bound in models just like values. This is reflected in the semantic domain *Model* that maps names into components which include operations and processes.

The semantic domains for states, operations, and processes are as follows:

$\text{State} = \text{Id} \xrightarrow{f} \text{Value}$

$\text{Operation} = (\text{State} \times \text{Value}) \rightarrow (\text{State} \times \text{Value})$

Processes are modelled by failure sets like in [Ros84]:

$\text{Process} = (\text{State} \times \text{Value}) \rightarrow \text{Failures} \times \text{Termination}$

Failures = Failure-**set**

$\text{Failure} = \text{Trace} \times \text{Refusal}$

Trace = Value*

Refusal = (Value × {<u>tick</u>})-**set**

Termination = Trace → (State-**finset** ∪ {<u>div</u>})

For the sake of brevity we do not elaborate on this, but refer to [Ros84].

6.8. Conclusion

We have presented a subset of the RAISE specification language (RSL), and illustrated its semantics. In order to keep the presentation to a reasonable size we have dealt only with the applicative aspects of RSL.

We have focused on points that we think characterise RSL or its semantics. These points include the structuring and naming scheme of RSL, the mixture of abstract types (sorts) and types specified by equations, the axiomatic style of defining values (in contrast to the "definitional" style of VDM) implying that we do not find least fixed points of recursive value definitions, and finally, the way in which we treat undefined expressions, using a two-valued logic for axioms together with existential equality.

7. The RAISE Project

The RAISE project is being carried out by a consortium formed by:

Dansk Datamatik Center (DDC), Lundtoftevej 1C, DK-2800 Lyngby, Denmark

STC Technology Limited (STL), London Road, Harlow, Essex CM17 9NA, United Kingdom

Asea Brown Boveri A/S (ABB), Ved Vesterport 6, DK-1612 København V, Denmark

International Computer Limited (ICL), ICL House, Putney, London SW15 1SW, United Kingdom

Dansk Datamatik Center is the main contractor. The RAISE project is part of the ESPRIT programme (ESPRIT 315) partly funded by the Commission of the European Communities. The project has a size of 115 staff-years. The project was started 1 January 1985 and runs for 5 years.

The project is divided into two phases. Phase I involves 64 staff-years of effort over (roughly) the first three project years. Phase I concerned with research and development of the method, language and prototype tools, while phase II is concerned with development of the final tools and training and technology transfer material. The industrial trial applications link the two phases. The trial applications will be carried out in the industrial environments of ICL and ABB. These trials ensure that RAISE meets the requirements of the software producing industry.

The project plan is illustrated in Table 4.

Table 4. The RAISE project plan

	Phase I			Phase II	
	1985	1986	1987	1988	1989
Fundamental issues	├──	────	──┤		
Method, language and tool specification		├──	────	──┤	
Industrial trial			├──	────	──┤
Final tools and technology transfer				├──	──────┤

The project is approaching the second year of phase II, and the definitions of the method and the language are undergoing the final revision. The industrial trial projects, based on the preliminary method, language, and tools, have been going on for some time now, and the feedback has been encouraging. The experience gained from these industrial trials has and will provide input to the final revision of method and language.

Acknowledgements

The work described here is the result of a collective effort by the RAISE project team. We would like to thank the following people for encouragement and inspiration: C. B. Jones, M. Broy, D. Bjørner, D. Sannella, A. Blikle and B. Monahan.

References

[BjJ82] Bjørner, D. and Jones, C. B.: *Formal Specification and Software Development.* Prentice Hall International, 1982.

[Bre88] Breu, M., Broy, M., Grünler, T. and Nickl, F.: *PA""dA-S Semantics.* PROSPECTRA Study Note M.2.1.S1-SN-1.3, Universität Passau, Fakultät für Mathematik und Informatik, 1988.

[BuG77] Burstall, R. M. and Goguen, J. A.: Putting Theories Together to Make Specifications. In: *Proc. Fifth International Joint Conference on Artificial Intelligence. Cambridge, Mass.*, pp. 1045–1058, 1977.

[BuG80] Burstall, R. M. and Goguen, J. A.: The Semantics of Clear, a Specification Language. In: *Proc. 1979 Copenhagen Winter School on Abstract Software Specifications. Lecture Notes In Computer Science* Vol 86, pp. 292–332, Springer-Verlag, 1980.

[CIP85] The Munich CIP Group: *The Munich Project CIP, The Wide Spectrum Language CIP-L. Lecture Notes in Computer Science Vol 183,* Springer-Verlag, 1985.

[Dij76] Dijkstra, E. W.: *A Discipline of Programming.* Prentice-Hall International, 1976.

[FGJ85] Futatsugi, K., Goguen, J. A., Jouannaud, J-P. and Meseguer, J.: Principles of OBJ2. In: *Eleventh Annual ACM Symposium on Principles of Programming Languages,* Association for Computing Machinery, Inc., 1985.

[Geo88] George, C. W.: *Practical Aspects of Development.* RAISE Report CWG/28/V4, STC Technology Limited, April 1988.

[GHW85] Guttag, J. V., Horning, J.J. and Wing, J. M.: *Larch in Five Easy Pieces.* Digital Systems Research Center, 1985. Report 5.

[HMM86] Harper, R., MacQueen, D. and Milner, R.: *Standard ML.* LFCS Report Series ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1986.

[Hoa85] Hoare, C. A. R. (ed.): *Communicating Sequential Processes. P-H Series in Computer Science,* Prentice-Hall International, 1985.

[Inm84] Inmos Ltd.: *Occam Programming Manual.* Prentice-Hall International, 1984.

[Jon86] Jones, C. B.: *Systematic Software Development Using VDM. P-H Series in Computer Science,* Prentice-Hall International, 1986.

[JPC88] Jørgensen, J., Palm, S. U., Christensen, P., Haff, P., Henriksen, L. W. and Sestoft, P.: *Preliminary Definition of the RAISE Specification Language.* RAISE Report JJ/14/V6, Dansk Datamatik Center, February 1988.

[Lam86] Lamport, L.: *LATEX: a Document Preparation System.* Addison-Wesley Publishing Company, 1986.

[Mon85] Monahan, B.: A Semantic Definition of the STC VDM Reference Language. November 1985. Unpublished notes.

[Pre87] Prehn, S.: From VDM to RAISE. In: *Proc. VDM '87 Symposium,* D. Bjørner and C. B. Jones (ed.). *Lecture Notes in Computer Science Vol 252,* pp. 141–150, Springer-Verlag, March 1987.

[ReT87] Reps, T. W. and Teitelbaum, T.: *The Synthesizer Generator Reference Manual,* 2nd Edn. Cornell University, Dept of Computer Science, July 1987.

[Ros84] Roscoe, A. W.: Denotational Semantics for Occam. In: *Seminar on Concurrency*, G. Winskel, S. D. Brookes and A. W. Roscoe (ed.), *Lecture Notes in Computer Science Vol 197*, Springer-Verlag, July 1984.

[Wir71] Wirth, N.: Program Development by Stepwise Refinement. *Communications of the ACM*, **14**, 221-227 (1971).

Received October 1988
Accepted October 1988 by C. B. Jones