Computing and Informatics, Vol. 20, 2001, 1–27, V 2003-Jun-10

THE LOGIC OF THE RAISE SPECIFICATION LANGUAGE

Chris George

International Institute for Software Technology,
United Nations University, Macao
e-mail: cwg@iist.unu.edu
www: http://www.iist.unu.edu/~cwg

Anne E. Haxthausen

Department of Informatics and Mathematical Modelling
Technical University of Denmark, Lyngby, Denmark
e-mail: ah@imm.dtu.dk
www: http://www.imm.dtu.dk/~ah

Abstract. This paper describes the logic of the RAISE Specification Language, RSL. It explains the particular logic chosen for RAISE, and motivates this choice as suitable for a wide spectrum language to be used for designs as well as initial specifications, and supporting imperative and concurrent specifications as well as applicative sequential ones. It also describes the logical definition of RSL, its axiomatic semantics, as well as the proof system for carrying out proofs.

Keywords: Formal methods, logics, RAISE

1 INTRODUCTION

An important technique for increasing the reliability of software systems is to use formal development methods. Formal methods provide mathematically based languages for specifying software systems and proof systems for verification purposes. During the last decade a whole range of formal methods have been developed. One of these is RAISE.

The goal of this paper is to describe and motivate the logic of the RAISE specification language, RSL. This logic is non trivial and interesting because the language supports many different specification styles.

It should be noted that for a given, formal language the term 'logic' can be used in two different but related senses. It may refer to the meanings of the 'logical' (truth-valued) expressions of a language. Alternatively, 'logic' may refer to the proof system, to the inference rules by which one may reason about terms of the language. In designing a language, choices made in the assignment of meanings to expressions influence the possible design of the proof system. For this reason we have chosen to use the term 'logic' as encompassing both senses.

In the remaining part of this section, we give a short introduction to RAISE including a survey of the major specification styles supported by RSL. Then, in section 2, we describe the rationale behind the design choices made for the meanings of 'logical' (truth-valued) expressions in RSL. Next, in section 3, we outline how RSL formally is given an axiomatic semantics in the form of a collection of inference rules that defines well-formedness and meanings of RSL constructs. In section 4 we describe how a proof system is derived from the axiomatic semantics in such a way that it is suitable for doing proofs in practice using a computer based tool. Finally, in section 5, we state our conclusions and report on new emerging proof tools.

1.1 RAISE Background

RAISE ("Rigorous Approach to Industrial Software") is a product consisting of a formal specification language (RSL) [31], an associated method [32] for software development and a set of supporting tools.

The Method The RAISE method is based on stepwise refinement using the invent and verify paradigm. Specifications are written in RSL. The notion of refinement will be described in section 2.4.

The Language RSL is a formal, wide-spectrum specification language that encompasses and integrates different specification styles in a common conceptual framework. Hence, RSL enables the formulation of modular specifications which are algebraic or model-oriented, applicative or imperative, and sequential or concurrent. Below, we outline the major syntactic aspects of the language.

A basic RSL specification is called a class expression and consists of declarations of types, values, variables, channels, and axioms. Specifications may also be built from other specifications by renaming declared entities, hiding declared entities, or adding more declarations. Moreover, specifications may be parameterized.

User-declared types may be introduced as abstract sort types as known from algebraic specification, e.g.

type Colour

or may be constructed from built-in types and type constructors in a model-oriented way, e.g.

type
  Database = Key $\xrightarrow[m]{}$ **Nat-set**,
  Key = **Text**

In addition RSL provides predicative subtypes, union and short record types as known from VDM, and variant type definitions similar to data type definitions in ML.

Values may be defined in a signature-axiom style as known from algebraic specification, e.g.

value
  black, white : Colour
axiom
  black $\neq$ white

in a pre-post style, e.g.

value
  square_root : **Real** $\to$ **Real**
  square_root(x) **as** r **post** r $\geq$ 0.0 $\wedge$ r $*$ r = x
  pre x $\geq$ 0.0

or in an explicit style as known from model-oriented specification, e.g.

value
  reverse : **Int*** $\to$ **Int***
  reverse(l) $\equiv$
    if l = $\langle\rangle$ **then** $\langle\rangle$ **else** reverse(**tl** l) $\frown$ $\langle$**hd** l$\rangle$ **end**

Functions may be imperative, reading from and/or writing to declared variables:

variable v : **Int**
value
  add_to_v : **Int** $\to$ **write** v **Unit**
  add_to_v(x) $\equiv$ v := v + x

where **Unit** is the type containing the single value (). In the function type it is stated which variables the function may access.

Functions may describe processes communicating synchronously with each other via declared channels:

channel i : **Int**, o : **Bool**
value
  test : **Int** $\to$ **in** i **out** o **Unit**
  test(x) $\equiv$ **let** inval = **in**? **in** o!(inval=x) **end**

In section 2 various kinds of value expressions and their meaning will be described.

RSL has also been extended to support real time [15, 22, 13, 16].

Semantic Foundations of RSL RSL is given a denotational [25] and an axiomatic semantics [26], and a subset of the language is given an operational semantics [4]. The construction of the denotational model and a demonstration of its existence is presented in [3].

Tool Support The RAISE tools provide support for constructing, checking and verifying specifications and development relations, and for translating concrete specifications into several programming languages.

History RAISE was developed during the years 1985-1995 in the CEC funded ESPRIT RAISE (315) and LaCoS (5383) projects. RAISE builds upon ideas reflected in a number of other formal methods and languages. The model-oriented language features were inspired by VDM [21] and Z [33], the algebraic features by algebraic specification languages like OBJ [10], Clear [5], ASL [36], ACT ONE [9] and Larch [14], the concurrency features by CCS [28] and CSP [20], and the modularity features by ML [24], Clear, ASL, ACT ONE and Larch.

Applications RAISE has been used on many applications. The initial ones were within the LaCoS project [6]. It has been used for many years at UNU/IIST, and a collection of case studies from there, illustrating a wide range of styles of use, has been published recently [7]. Also at the Technical University of Denmark, a range of applications have been done, e.g. [18, 23, 19, 17].

2 THE RSL LOGIC

There are a number of possible choices for the logic of a specification language. In this section we present the rationale behind the design of the RSL logic. Subsection 2.1 introduces the problem of potentially undefined expressions, 2.2 presents the logic of the applicative subset of RSL, 2.3 extends this to imperative and concurrent RSL, 2.4 presents the RSL definition of refinement and relates it to the logic, and 2.5 introduces the notion of confidence conditions.

2.1 Definedness

A fundamental question to decide about a logic for a specification language is what to do about problematic expressions like 1/0, or **hd** $\langle\rangle$, or **while true do skip end**. Such expressions do not have very obvious denotations (meanings).

Expressions like these may seem more likely to arise in early specifications (like the second) or in implementations (like the other two), but as RSL is a "wide spectrum" language, intended to support both initial specification and development to code, any of these kinds of expression may occur.

There are two facts to make clear from the start. First, in a reasonably expressive language, such expressions cannot be avoided by purely mechanical means. The equality of an integer expression with zero, for example, is not decidable. If we wish to ensure such expressions do not occur then we will need to do proof. We can choose to do such proof as part of "type checking", as in PVS [30] for example, or at some later time. In contrast, it is possible in a typed language mechanically to either reject an expression as ill-typed (like 1 + **true**, for example) or assign it a type. So our "problematic" expressions will have types.[1]

The second fact is that there is a variety of schemes available to deal with such expressions in a logic. This is not a question of fundamental research, but of choosing from the options available. The choices made will affect the ease with which people can learn and use a language, and the ease of creating and using proof tools for that language. There are two factors in particular that influenced the choices made in the design of RSL:

1. As mentioned above, RSL is a "wide spectrum" language intended to support development to specifications very close to programming languages. This in turn means that the ability to conveniently translate to a programming language at least the constructs likely to appear in such detailed specifications is something to consider.
2. The design of RSL is as regular as possible. This means that apart from having type **Bool** there are as few restrictions as possible placed on what kind of expressions may occur in a predicate. In particular they do not need to be applicative: they may have *effect*s by accessing variables and even channels.

One possible approach to problematic expressions like 1/0 is to say that "every expression denotes a value in its type, but it might not be possible or sensible to say which". In this approach 1/0 could be some unknown integer, but **while true do skip end** would have to be equivalent to **skip**, since **skip** is the only value in the type **Unit**. This (a) seems counter-intuitive and (b) seems to preclude any analysis of termination since the logic would equate a non-terminating expression with a terminating one.

Some languages, Z [33] and B [1] for example, take the approach that all expressions denote. It is argued that this gives a simple and intuitive logic. Examples like those based on non-termination can be considered as less relevant to Z, which aims to define initial specifications rather than implementations, though this seems less justified for B. These languages, unlike VDM [21] and RSL, for example, also distinguish between Boolean expressions, *predicates*, and expressions of other types: they have no Boolean type.

[1] In languages like RSL which allow overloading there may be a (finite) collection of possible types, but this does not materially affect the following discussion.

RSL, like VDM, from which came much of RSL's inspiration, has a Boolean type, **Bool**, and allows in its logic for expressions that might not denote values in their types. The *definedness* of expressions then becomes a concept needed in the proof theory. The semantics of such languages may, as in RSL's case, be *loose*: there are some expressions whose definedness is not specified, and 1/0 is an example. A programming language implementation in which it raises an exception is acceptable, as is one where it evaluates to 1, say. Looseness is not a critical issue: much more important is how in writing and developing specifications we can avoid such expressions occurring. We will return to this issue later in section 2.5.

2.2 Applicative RSL

The discussion about logic for RSL becomes more complicated when we include expressions that can write to variables, or access channels, i.e. expressions that can have *effects*. We will try to give a simple exposition by dealing with applicative expressions first, and explain the additions we need for imperative expressions later. But we shall also try to avoid misleading readers by indicating in the first part where the explanation is only for applicative expressions.

2.2.1 Equivalence

A basic issue when expressions might not denote values is the meaning of equality. RSL has two "equality" symbols, $\equiv$ and $=$. The first of these, equivalence, is more relevant to the discussion on logic, and we will discuss equality $=$ later in this section.

$\equiv$ is sometimes called a "strong" equality, as it allows undefined expressions to be compared. It has the mathematical properties of a congruence, which means that it is an equivalence relation (it is reflexive, transitive and commutative) and (for applicative expressions) it allows substitution: an expression can be replaced by an equivalent one in any context.

The important properties we shall need for equivalence are

1. A defined expression is never equivalent to an undefined one, e.g. the equivalence

   while true do skip end $\equiv$ **skip**

   is **false**.
2. Equivalence always returns **true** or **false**, i.e. it is always defined.

Equivalence is in fact the same as semantic equality: two expressions are equivalent if they have the same semantics.

A logic that allows undefined expressions and includes a strong equality is often referred to as "three valued". We prefer to say that there are just two Boolean values (**true** and **false**), and say that only defined expressions have values. There are three 'basic' undefined expressions in RSL: **chaos** (equivalent to **while true do skip**

end), **stop** (the external choice over the empty set) and **swap** (the internal choice over the empty set). **stop** and **swap** arise in RSL because it includes concurrency, and both represent forms of deadlock. In practice we normally want to avoid the possibility of undefined expressions in specifications, and making the main choice one between definedness and otherwise is mostly sufficient. We shall in our examples for simplicity usually use **chaos** as the archetypal undefined expression.

2.2.2 Convergence

RSL includes concurrency, and so includes the notion of internal (nondeterministic) choice. This also arises if relations or mappings that are "non-functional" are allowed. For example, what happens if the map $[\,1 \mapsto 2,\ 1 \mapsto 3\,]$ is applied to the value 1? In RSL the result is equivalent to the expression $2 \sqcap 3$. This expression is defined, but will not always evaluate to the same result. We use the term *convergent* to mean defined and having a unique value.

We shall see that definedness and convergence often arise in the proof theory because we need them as conditions for rules to be sound. For example, we will see that

$\mathrm{A} \wedge \mathrm{B} \equiv \mathrm{B} \wedge \mathrm{A}$ when A and B are defined
$\mathrm{A} \vee \sim\mathrm{A} \equiv$ **true** when A is convergent

(In the non-applicative case these also need the effects of A and B to be at most "read-only")

However, the case of defined but nondeterministic is (a) rare and (b) dealt with by other rules, so in practice we always use convergence even though definedness is occasionally sufficient.

2.2.3 Connectives

How do we define the logical connectives $\wedge$ (and), $\vee$ (or), $\sim$ (not) and $\Rightarrow$ (implies)? The approach in VDM is to use a logic called LPF [2], the "Logic of Partial Functions". The intuition in LPF's definition of $\wedge$, for example, is that for an expression $\mathrm{A} \wedge \mathrm{B}$, if either A or B evaluates to **false** the whole expression should be **false**, even if the other is undefined. If either evaluates to **true** then the expression evaluates, if at all, to the value of the other. So it is undefined if one is **true** and the other undefined, or both are undefined. Note that this explanation is symmetric in A and B, and indeed in LPF, as in classical logic, $\wedge$ is symmetric (commutative).

The main problem with LPF is that $\wedge$ is hard to implement in a programming language, because it requires parallel evaluation of the two component expressions, such that if one evaluates to **false** then the whole evaluation immediately terminates and returns **false**. For mainly this reason, RSL chose instead to define the logical connectives in terms of if-expressions, in a "conditional logic".

$\mathrm{A} \wedge \mathrm{B} \equiv$ **if** A **then** B **else false end**

A $\vee$ B $\equiv$ **if** A **then true else** B **end**
A $\Rightarrow$ B $\equiv$ **if** A **then** B **else true end**
$\sim$A $\equiv$ **if** A **then false else true end**

These are not new inventions. This version of $\wedge$, for example, appears as `cand` in some languages, as `andalso` in some others.

So if-expressions are fundamental, and we need to explain what they mean. We do this formally in terms of proof rules, but here is the intuitive explanation of the meaning of **if** A **then** B **else** C **end** (in the applicative case):

1. If A is undefined, then the expression is equivalent to A.
2. Otherwise, if A is nondeterministic (so it must be **true** $\lceil\rceil$ **false**) the expression is equivalent to B $\lceil\rceil$ C.
3. Otherwise, if A is true then the expression is equivalent to B, and if A is false then the expression is equivalent to C.

This coincides with the meaning of if-expressions in programming languages, and has the immediate consequence that if-expressions, and hence the logical connectives, are easy to implement. This is the main advantage of RSL's conditional logic. The main disadvantage is that we lose the unconditional commutativity of $\wedge$ and $\vee$. For example:

chaos $\wedge$ **false** $\equiv$ **chaos**
but
false $\wedge$ **chaos** $\equiv$ **false**

$\wedge$ and $\vee$ are in general only commutative when both the components are defined.

This is by no means the only case where we need to be concerned with definedness, and we decided that the implementability of our logic was the overriding concern.

Incidentally, many other rules of classical logic hold for conditional logic. For example, $\wedge$ and $\vee$ are associative, and what is sometimes used as the definition of $\Rightarrow$ holds:

A $\Rightarrow$ B $\equiv$ $\sim$A $\vee$ B

All the laws of classical logic hold when expressions are convergent. The ones needing convergence are commutativity and "excluded middle". LPF also needs definedness for "excluded middle".

2.2.4 Quantifiers

RSL includes the quantifiers $\forall$, $\exists$, and $\exists!$. They all quantify over values in the appropriate type. Hence they say nothing about undefined expressions. For example, we can say, correctly, that

$\forall$ x : **Bool** • x = **true** $\vee$ x = **false**

without being able to conclude anything about an undefined expression such as **chaos** being either **true** or **false**.

2.2.5 Functions

λ-expressions only admit beta reduction (application) when applied to values in their domain. E.g.

λ (x : **Int**, y: **Int**) • x

cannot be applied to (0, **chaos**) to give 0. In fact the semantics of function application is standard call by value: if any argument expression is undefined then so is the application.

2.2.6 Axioms

In RSL axioms may be declared. In addition, all value declarations are short for value signatures plus axioms. For example, suppose we have the value declaration

value
  factorial : **Nat** $\stackrel{\sim}{\to}$ **Nat**
  factorial(n) $\equiv$
    if n = 1 **then** 1 **else** n $*$ factorial(n$-$1) **end**
  pre n $\geq$ 1

This is short for

value
  factorial : **Nat** $\stackrel{\sim}{\to}$ **Nat**
axiom
  $\forall$ n : **Nat** •
    factorial(n) $\equiv$
      if n = 1 **then** 1 **else** n $*$ factorial(n$-$1) **end**
    pre n $\geq$ 1

and in turn "e **pre** p" is short for "(p $\equiv$ **true**) $\Rightarrow$ e". The inclusion of "$\equiv$ **true**" is just a technique to ensure that if p is undefined the precondition reduces to **false**.

So when can we use this axiom to "unfold" an application of factorial to replace it with its defining expression? We want to use the equivalence within the axiom, remembering that $\equiv$ is a congruence, i.e. allows substitution. We see that:

1. The actual parameter must be a value in the type **Nat**, i.e. it must be a non-negative integer, because of the meaning of $\forall$. We cannot unfold, say, factorial(**chaos**) or factorial($-$1).

2. The precondition must then be equivalent to **true**. We cannot unfold, say, factorial(0).

So the rules of the logic ensure that the apparent aim of the specifier, that factorial should only be applied to strictly positive integers, is respected.

2.2.7 Equality

We need to define the symbol =. In RSL its definition is just like that of any other infix operator, such as +. RSL adopts a general "left-to-right" evaluation rule, so the meaning of

A = B

is

1. Evaluate A. If it is undefined so is the whole expression.
2. Otherwise, evaluate B. If it is undefined so is the whole expression.
3. Otherwise, compare the results of evaluating A and B and return true if they are the same value, false otherwise.

= is therefore given a definition in terms of the underlying = for the carrier set of every type in RSL. If either of A or B is undefined then so is the equality. If either of them is nondeterministic then so is the equality. Otherwise, in the applicative case, it is the same as $\equiv$.

The other important feature of = is that it is implementable. Such an equality is sometimes called "programming language equality", as its evaluation is the same as in programming languages (except that many languages decline to fix the evaluation order, preferring the convenience of compiler writers to the confidence of users).

Is it confusing to have both = and $\equiv$? The advice to users is simple: always use = in your expressions, and take care to avoid undefinedness. Users should only write $\equiv$ in function definitions, where it is part of the syntax, and as the main equality in axioms.

We have seen that (for applicative expressions), when expressions are defined and deterministic, equality and equivalence coincide, so there should be no problem. What happens if a user accidentally forgets to check for definedness? Take the RSL version of an example quoted by Stoddart et. al. [34], for example:

value
  s : **Int-infset**
axiom
  card s = 5

Are there any models (implementations) of this specification in which the set s is infinite? (In languages in which all values denote there may be such models, which is why the example is quoted.)

If s is infinite, then in RSL **card** s is undefined. So the axiom expression is apparently undefined, following our rules for equality. So what does it mean to have an undefined axiom? In fact we avoid this question: every axiom implicitly has an "≡ **true**" added (like the precondition we discussed earlier). So if s is infinite the axiom would reduce to **false**, and we conclude there can be no such model: s must be finite as we presumably expected.

There are a few other places where "≡ **true**" is included to make sure that undefined predicates reduce to **false**. As well as in axioms and preconditions, these are postconditions and restrictions. Restrictions are the predicates following the "bullet" in quantified expressions, implicit let-expressions, comprehensions, comprehended expressions, and for-expressions.

2.3 Imperative and Concurrent RSL

When we consider expressions that can have effects i.e. that can read or write variables, input from or output to channels, we need to extend the logic a little. In this section we explain the extensions.

First it is perhaps worth noting another problem with the LPF approach if expressions may be imperative. We noted earlier that LPF has to assume some kind of parallel evaluation rule to allow, for example, for one expression in a conjunction to be undefined when the other is false. But if the expressions may write to variables it is unclear how to deal with such effects with LPF's parallel evaluation. What should be the effect on the variable v, for example, of evaluating:

(v := v + 1 ; **true**) ∧ (v := v − 1 ; **false**)

Imperative specifications, like imperative programs, depend very heavily on evaluation order.

2.3.1 Equivalence

The general semantics of expressions is that they may have effects as well as returning results. For two expressions to be equivalent we require that they have equivalent effects as well as equivalent results.

The equivalence expression "e1 ≡ e2" expresses a purely logical equivalence. It evaluates to **true** if the expressions would have the same effects, and would return the same results: there is no actual evaluation. Hence, unlike equality, evaluating an equivalence does not generate any effects.

If v is an integer variable, then in some context, after assigning to v, we may know that "v ≡ 1". But clearly we cannot assert this in an arbitrary context, because v may have been assigned some other value there. To obtain a congruence relation we introduce an extra connective "always" □. The expression "□ p" means "p is true in any state", i.e. regardless of the contents of variables. Thus "□ e1 ≡ e2" is a congruence: it allows e1 to be replaced by e2 in any context.

$\Box$ is implicitly included in all axioms. Since constant and function definitions are just shorthands for signatures and axioms, it is therefore implicit in any value definition. In practice users do not need to write it.

2.3.2 Equality

For expressions with effects the difference between equality and equivalence becomes more marked. As we remarked earlier, an equality is evaluated left-to-right. At the end only the result values are compared. Therefore if the expression on the left has effects, these can affect the result of the expression on the right.

Suppose we have an integer variable v and we declare a function to increment it and return its new value:

variable
  v : **Int**
value
  increment : **Unit** → **write** v **Int**
  increment() ≡ v := v + 1 ; v

Now consider the two expressions

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

The first is equivalent to **true**, and its evaluation does not change v. We say that ≡ has only a "hypothetical" evaluation. Even when expressions have effects, ≡ remains reflexive.

The second has an effect of increasing v twice, as both the increment applications are evaluated. And we see that the result of the equality must be false: whatever the initial value of v, the result on the right will be one greater than the result on the left. We can summarise by concluding

(increment() ≡ increment()) ≡ **true**
and
(increment() = increment()) ≡ (v := v + 2; **false**)

The second result may seem surprising, but it is consistent with most programming languages. This does not mean, of course, that one would encourage anyone to write expressions in such a style!

2.3.3 Evaluation Order

The possibility of effects means that we need to be clear about the evaluation order of all expressions. For example, there is an evaluation rule for if-expressions:

if A **then** B **else** C **end** ≡ **let** x = A **in if** x **then** B **else** C **end end**

where the identifier x is chosen so as not to be free in B or C (or such free occurrences would become bound in the let-expression). Such rules, using let-expressions, are very common in the proof rules of RSL, and we need to be clear what the semantics of let-expressions is. Using the let-expression above as an example, its evaluation is as follows:

1. A is evaluated. If it is undefined, then so is the whole expression. Otherwise, it may have effects, and will return a value. Since A must be a Boolean expression, this value must be either **true** or **false**. (If A is nondeterministic, we still get one of these, but we don't know which.)
2. The value returned by A is bound to the identifier x, and then we evaluate the second expression in the let-expression, i.e. the if-expression in this example. So we will then evaluate either B or C according to the value returned by A.

For example, using our previous discussion about the increment function, we could conclude that

if increment() = increment() **then** B **else** C **end** ≡ v := v + 2 ; C

2.3.4 Reasoning Style

It is common in specification methods to use an axiomatic, "equational" style of reasoning for applicative constructs, as one does in mathematics. In RSL we typically use the same style of reasoning, based on equivalences, for imperative sequential and for concurrent descriptions as well as for applicative ones. Other methods typically use reasoning based on Hoare logic or weakest preconditions (wp) for sequential imperative descriptions, and perhaps temporal logic for concurrent ones. This is mostly a question of style rather than substance: RSL includes pre- and postconditions, and reasoning in terms of these is possible, and appropriate in particular for discussing iterative expressions (loops). But we generally find that equational reasoning can be used for all styles of specification.

2.4 Refinement in RSL

Since RSL is a modular language, refinement is aimed in particular at allowing substitution. If a module A, say, depends on another module B, say, then if we have a module B' that refines B, substituting B' for B should produce a module A' that refines A by construction. Refinement is required to be monotonic with respect to module composition. It is then possible to use *separate development* [32] to develop specifications expressed in several modules: modules can be developed independently, and provided each development can be shown to be a refinement then putting the refined modules together will refine the specification as a whole.

A definition of refinement in RSL is that class expression B′ refines class expression B provided:

1. the *signature* of B′ includes that of B
2. all the *properties* of B hold in B′

The *signature* of a class consists of the type identifiers declared in it with the types (if any) for which they are abbreviations, the value, variable and channel identifiers with their types, and the object identifiers with the signatures of their classes. The *properties* of a class are defined in the book on the RAISE method [32].

The first condition for refinement ensures that substituting B′ for B in some context will not generate type errors. It leads to a somewhat more restricted notion of refinement than in some languages that don't meet RSL's requirement to support separate development. Identifiers have to remain the same (though this can easily be fixed by RSL's renaming construct). Types that are abbreviations have to maintain the same abbreviation: if we declare in B, say,

type T = **Int-set**

then we cannot in B′ refine type T to be say, the type of lists of integers (**Int***), because in general we would get type errors when substituting B′ for B to make A′. There is a standard technique in the RAISE method [32] for overcoming this problem, by first abstracting the original definition. We change the definition of T in B to:

type T
value setof : T → **Int-set**

Both the type T and the function setof are left abstract. The abstraction expresses that a set can be extracted from a T value, rather than saying a T value is a set. Other definitions in B will also need changing, of course, using setof. Then we can define in B′:

type T = **Int***
value
  setof : T → **Int-set**
  setof(t) ≡ **elems** t

These definitions in B′ refine those in B.

A feature of RSL is that the language can itself express the properties of any class expression. This in turn means that the logical conditions for refinement can be expressed in RSL, and indeed the RAISE tool [11] can generate these as an RSL theory.

The second condition for refinement effectively says that anything that can be proved about a class can be proved about a class that refines it. So the stronger the properties of a module the less "room" there is to refine it. This is why, in particular, we adopt a particular kind of theory for functions. The definition for factorial given earlier in section 2.2 says nothing about what the factorial function is when the arguments are not in the domain type **Nat** or do not satisfy the precondition: the given definition only applies for strictly positive arguments. It is then possible to refine factorial, if desired, by defining factorial(0) and even factorial for negative arguments. In fact unless the function is declared with a domain type that is maximal (one that has no subtypes, such as **Int** rather than **Nat**), and without any precondition, it is in RSL impossible to say what its domain is. This is intentional: if we could calculate the domain it would be a property of the definition, and allow for no refinement that enlarged the domain.

Another feature of the logic of RSL is that it can distinguish between determinism and nondeterminism. To be more precise, consider

value
  f() : **Unit** $\stackrel{\sim}{\to}$ **Int**
axiom
  f() = 1 ∨ f() = 2 (1)

We term this a *loose* specification: it has more than one model, and so more than one refinement. In fact there are three: one where f() always returns 1, another where f() always returns 2, and a third:

f() : **Unit** $\stackrel{\sim}{\to}$ **Int**
f() ≡ 1 ⌈⌉ 2 (2)

Here f is nondeterministic, and its theory is different from either of the others. So the "more deterministic" refinement ordering supported by some specification languages is not supported by RSL (though it is often not clear when people speak of nondeterminism whether they mean looseness (1) or nondeterminism (2): often they actually mean looseness). Nondeterminism is important in analysing concurrency, so we need to be clear about the distinction between it and looseness.

2.5 Confidence Conditions

We return to the "problematic expressions" we started discussing at the start of this section 2. We have discussed how the logic of RSL can deal with undefined expressions. We can also see how to write expressions that are safe from undefinedness. For example, suppose we have an RSL finite map (many-one relation) m with **Int** domain type and **Text** range type. Suppose we want to specify that all the texts in the map are non-empty. If we write

∀ x : **Int** • m(x) ≠ ""

then m(x) may be undefined for values of x not in the domain of the map. As it happens, in this case, the implicit "≡ **true**" in the quantified expression takes care of the undefinedness, but (a) we don't encourage users to write specifications based on such details of the logic, and (b) the "$\equiv$ **true**" will not be there when we transform this into an implementation with a loop, say, and we should perhaps help the implementor a little. So there is a general rule that you never write a map application without making sure that it is guarded by a check that the argument is in the domain. So there should be a "x $\in$ **dom** m" condition, which may be the left of an implication or conjunction, the condition of an if-expression (with the application in the then part), or part of a precondition of the function definition in which the application occurs. Here (remembering the other rule of thumb that $\forall$ expressions almost always use $\Rightarrow$) we should obviously have written

$\forall$ x : **Int** • x $\in$ **dom** m $\Rightarrow$ m(x) $\neq$ ""

and we see that the conditional logic means that the application is only evaluated when the map argument is in the map's domain.

Guard conditions, like "x $\in$ **dom** m" for the application "m(x)", are called "confidence conditions" in RSL. We use this term because it is not always necessary to include them if our aim is just to avoid undefinedness: the quantified expression above is an example. But if we always include them then we have more confidence that the specification does not include undefined expressions, which means in turn that it is less likely to be inconsistent. The RAISE tool [11] includes a "confidence condition generator" that generates the confidence conditions for a range of potentially undefined expressions. Map arguments being in domains, list arguments being in index sets, and partial operator and function arguments being in domain types and satisfying preconditions, are in practice the most important ones. Checking that they hold in the contexts that generate them is not in general decidable, and it needs proof tools to discharge them formally.

Confidence conditions are usually best checked by inspection: they act as reminders. Unfortunately, as the specifier's skill increases the "hit rate" of conditions requiring attention becomes low, so the possibility of missing them during inspection rises. We now have proof tool support for discharging many of them automatically.

3 THE AXIOMATIC SEMANTICS: A LOGIC FOR DEFINITION

In this section we explain how RSL is given a proof theory [26] that provides the axiomatic semantics of RSL.

3.1 Purpose and Role

The purpose of the proof theory is to provide formation rules for determining whether a specification is well-formed (type correct etc.) and proof rules for reasoning about specifications, e.g. deciding whether two RSL terms are equivalent (have the same meaning) or deciding whether an RSL specification is a refinement of another RSL specification.

The role of the proof theory is to provide an axiomatic semantics that defines the meaning of RSL while the role of the denotational semantics [25] is just to ensure consistency of the proof theory by providing a model. The reason for taking this view of roles is the fact that the proof theory is needed anyway (since we should be able to reason about specifications) and it is much more comprehensible than the denotational one (since its meta language is much simpler). The RSL type checker implements the formation rules, while the RAISE justification editor implements a proof system (see section 4) that is derived from the axiomatic semantics.

3.2 The Form of the Axiomatic Semantics Definition

The axiomatic semantics consists of a collection *inference rules* of the form

$$\frac{\text{premise}_1 \ \ldots \ \text{premise}_n}{\text{conclusion}}$$

where the upper part consists of a possibly empty list of formulae called the premises and the lower part consists of a formula called the conclusion. The most important kinds of formulae are those for expressing static semantics of RSL terms, refinement of RSL terms and equivalences between RSL terms. The formulae may contain term variables.

As usual the inference rules can be instantiated by consistently replacing term variables with actual, variable free terms of the same syntactic category. A rule represents all its legal instantiations. An instantiated rule expresses that if the (instantiated) premises hold then also the (instantiated) conclusion holds.

3.3 The Collection of Inference Rules

In this section we give examples of important classes of inference rules.

Formation Rules For each RSL term that is not defined to be a context independent shorthand (see next paragraph), there is an inference rule defining its static semantics. For instance, the rule

$$\frac{}{\text{context} \vdash \mathbf{true} :\preceq \mathbf{Bool}}$$

states that the RSL term **true** is well-formed and has type **Bool** as its static semantics. This rule is simple having no premises and not referring to the context[2], but many rules have static premises expressing that sub-terms are well-formed, have appropriate types, only refer to names declared in the context, etc.

The formation rules provide a decidable test for whether terms are well-formed.

[2] A context provides assumptions about identifiers and operators. In its most basic form a context is an RSL class expression.

Context Independent Equivalence Rules There is a class of inference rules defining a context independent equivalence relation, $\cong$. Intuitively $\vdash$ $\text{term}_1 \cong \text{term}_2$ asserts that the two terms term_1 and term_2 are equivalent in all respects, i.e. have the same properties (attributes, static semantics and dynamic meaning) and can be substituted for each other anywhere.

The context independent equivalence rules typically express algebraic laws like commutativity of the concurrency operator:

$$\frac{}{\begin{array}{l}\vdash \text{value\_expr}_1 \parallel \text{value\_expr}_2 \cong \\ \qquad \text{value\_expr}_2 \parallel \text{value\_expr}_1\end{array}}$$

A subclass of the rules, the context independent expansion rules, have the role of expressing that certain RSL terms are shorthands for others. For instance, the rule

$$\frac{}{\begin{array}{l}\vdash \text{value\_expr}_1 \wedge \text{value\_expr}_2 \cong \\ \quad \mathbf{if}\ \text{value\_expr}_1\ \mathbf{then}\ \text{value\_expr}_2\ \mathbf{else\ false\ end}\end{array}}$$

states that a conjunction of the form $\text{value\_expr}_1 \wedge \text{value\_expr}_2$ is a shorthand for **if** value_expr_1 **then** value_expr_2 **else false end** (cf. the discussion of the meaning of the RSL connectives in section 2.2.3).

When a term is defined to be a shorthand, there do not need to be any other rules having a conclusion concerning that term – all properties are to be derived from the term that it is a shorthand for. For a term that is not a shorthand, there will typically be several rules having a conclusion concerning that term.

Context Dependent Equivalence Rules There is another class of inference rules defining a context dependent equivalence relation, $\simeq$ for stating that in a given context two terms are equivalent in the more weak sense that they have the same meaning (which may depend on the context), but not necessarily the same static properties. For example, their free variables might differ. An example of such a rule is:

$$\frac{\begin{array}{l}\text{context} \vdash \text{value\_expr} :\preceq \text{opt\_access\_desc\_string}\ \mathbf{Bool} \\ \text{context} \vdash \mathbf{read\text{-}only\text{-}convergent}\ \text{value\_expr}\end{array}}{\begin{array}{l}\text{context} \vdash \\ \quad \mathbf{if}\ \text{value\_expr}\ \mathbf{then}\ \text{value\_expr}\ \mathbf{else\ true\ end} \simeq \\ \qquad \mathbf{true}\end{array}}$$

It states that in a given context the value expression **if** value_expr **then** value_expr **else true end** is equivalent to **true**, if (1) the constituent value_expr is well-formed and has type **Bool**, and (2) the constituent value_expr is readonly and convergent.

The first condition in the rule ensures that the equivalence between the two terms can only be proved when these are both well-formed. Otherwise, the rule would not be sound (one could e.g. prove **if** 1 **then** 1 **else true end** $\simeq$ **true**). The second condition in the rule ensures that the if-expression does not have any side effects and is convergent. Otherwise, one could e.g. prove **if** x := 1; **true then** x := 1; **true else true end** $\simeq$ **true**.

As in the context independent case, there is a subclass of the rules the role of which is to define shorthands.

Refinement Rules A collection of inference rules define the refinement relation.

Auxiliary Rules There are several collections of rules defining auxiliary functions and relations that are used in the premises of the other rules.

For instance, there is a collection of rules defining *attribute* functions, *new* and *free* that take an RSL term as argument and return the set of identifiers and operators that are declared and occur free in the term, respectively. These are used in premises of other rules to express restrictions on identifiers or operators appearing in the conclusion.

3.4 Relation to Denotational Semantics

RSL has been given a denotational semantics in [25]. This denotational semantics can in an obvious way be extended to also covering the formulae of the meta language of the axiomatic semantics. For instance, the meaning of a formula of the form *value_expr* can be defined to have the same as the meaning as the value expression $\Box(value\_expr \equiv \textbf{true})$.

Soundness and completeness wrt. the denotational semantics is discussed in section 5.

4 THE RSL PROOF SYSTEM: A LOGIC FOR PROOF

When we consider a suitable logic for doing proof our concerns become more practical. We have to take care of soundness, of course: we must not enable the proof of invalid theorems. But what the user will in practice be most concerned with is the ability to prove valid theorems, and preferably being able to do so automatically.

This raises the question of completeness. It seems obvious at first that the proof system should be complete (all valid theorems should be provable). But in practice it seems worth sacrificing completeness in a few places to get more ease of proof in the vast majority of cases. We in fact sacrificed completeness in favour of a simplified language for proof rules. In particular, the language does not include contexts, which are in general much more easily handled implicitly by a tool, and as a consequence RSL's local-expressions, which allow local definitions, are not catered for with full generality.

So the proof system is sound but incomplete compared to the axiomatic semantics, though most of the incompleteness is handled by a tool.

4.1 Justification Editor

The proof rules are intended for application by a tool, the RAISE *justification editor*. We can therefore immediately assume a mechanism for type-checking, and make a general assertion that a proof rule may be applied only to well-formed expressions, and application only succeeds when it gives a result that is also well-formed. The tool also handles the *context*, the bindings of names to their definitions, which further simplifies the rule language.

The general form of a rule is the inference rule introduced in section 3. But most of the proof rules take the form

$$term_1 \simeq term_2 \ \mathbf{when} \ term_3$$

where $term_3$ (termed the *side condition*) is the conjunction of the premises. The context is the same for all the terms. Such a rule allows an expression matching $term_1$ to be replaced by the corresponding instantiation of $term_2$ (or vice versa), provided the instantiated $term_3$ can be proved. The important point about the justification editor is that it allows proof rules to be applied to *sub*-expressions of a goal. The basic style of proof is to

- select a sub-expression (mouse drag)
- show the applicable proof rules (menu)
- select and apply a rule (mouse clicks)

The applicable rules are selected by syntax, and are generally few in number, so supporting easy selection. This allows a very user-controlled, natural, and flexible style of proof.

Side conditions generate separate proof obligations, so that a proof becomes a tree. The branches can be proved in any order, so one may choose whether to check a side condition first, in order to check a strategy is applicable, or proceed with the main proof first, in order to check a strategy is appropriate.

The term *justification* was coined for a proof in which not all steps have to be proved formally. The tool accepts the informal assertion that a goal is true, or that an expression may be replaced by another asserted as equivalent. The tool keeps track of such informal steps. A justification may be stored and then loaded again, for the proof to be reviewed, or pretty-printed, or for informal steps to perhaps be made formal.

4.2 Proof Rule Language

The language for expressing proof rules is in fact a small extension of RSL. A correspondingly small extension to the type checker allows proof rules to be type checked. The justification editor supports the input of proof rules: they are mostly not built-in. This allows for greater transparency, and the main document listing the rules [12] is in fact generated from the input given to the justification editor.

The proof rule language has a number of rules for instantiation of term variables that are implicit in their names. For example, names differing only in the number of primes must have the same maximal type, while names may otherwise have different types. Names starting with "p_" may only be matched by pure expressions, those starting with "ro_" may only be matched by read-only expressions, those involving "eb" must be Boolean expressions, etc.

For example, the rule for if-expressions mentioned in section 3 is written

[if_annihilation1]
  if eb **then** eb **else true end** ≃ **true**
    when convergent(eb) ∧ **readonly**(eb)

The first line is the rule name (which appears in selection menus). There are various naming conventions, which indicate that this rule may be used to "annihilate" or remove an if expression. (Or, applied right-to-left, to introduce one.) The side condition uses two of the "special functions" that are used in many proof rules. Their (partial) evaluation is built into the justification editor, so that **readonly**, for example, will generally be discharged automatically when it holds. **convergent** can be more difficult to prove, of course.

"**convergent**(eb)" is just an abbreviation for "eb **post true**". Special functions typically express simple concepts but may have more complicated definitions. There are special functions, for example, to express the conditions necessary for a new binding not to capture free names (**no_capture**), for a replacement binding only to capture names that the previous one did (**no_new_capture**), and for an expression to match a pattern (**matches**).

Of particular importance in a language with *effects* (assignments to variables, input and output), are rules showing the order of evaluation. Without these many constructs would be ambiguous. There is a general "rule-of-thumb" that evaluation is left-to-right. More formally it is defined by "_evaluation" rules. The evaluation of an if-expression, for example, is given by

[if_evaluation]
  if eb **then** e **else** e′ **end** ≃
    let id = eb **in if** id **then** e **else** e′ **end end**
      when no_capture(id, e) ∧ **no_capture**(id, e′)

This rule is always applicable left-to-right — the side condition only requires the choice of an identifier not free in e or e′ — and shows that the guard condition is evaluated before anything else. If eb does not terminate then neither will the let-expression. Otherwise eb will evaluate to either **true** or **false** and the corresponding if_annihilation rule can be applied. To deal with the case that eb is nondeterministic we have

[let_int_choice]
  let b = e ⌈⌉ e′ **in** e1 **end** ≃
    let b = e **in** e1 **end** ⌈⌉ **let** b = e′ **in** e1 **end**

To show that the rules are sound, we divide them into "basic" rules, which are just rewritings of the definitional rules from section 3, and "derived" rules, which should be derivable from the basic ones. There are just over 200 basic rules, and currently well over 2000 derived ones, which shows the importance of convenience in proof.

4.3 Context Rules

No distinction between basic and derived rules is made in the justification editor, as it is generally uninteresting for the user. A distinction that is made is between the rules of RSL and the rules that the users may apply because they are axioms of their specifications, termed *context rules.* For example, when proving something in the context of the axiom

axiom
  [is_in_empty] ∀ x : Elem • ∼is_in(x, empty)

(where the type Elem, the constant empty and the function is_in are also declared in the context) the *context rule*

[is_in_empty]
  is_in(e, empty) ≃ **false**
    when convergent(e) ∧ **pure**(e) ∧ **isin_subtype**(e, Elem)

is available. Note the way the universal quantifier gives rise to a term variable "e". Value definitions, since they are equivalent to signatures and axioms, also generate context rules.

Some type definitions also give rise to context rules. Variant types generate induction rules, and disjointness rules asserting that different constructors generate different values. Record types are treated as singleton variants, and so also generate induction rules.

5 CONCLUSIONS

5.1 Definition Versus Proof

We can now summarise the main differences between rules used for definition and those used for proof:

- The defining rules need to be concerned with well-formedness (scope and type) rules, while the rules for proof can assume terms are well-formed.

- If the defining rules were the only definition they would necessarily be considered sound and complete, because they would be the only reference. But we also have a denotational semantics and can conclude, conventionally, that with respect to that semantics they need to be sound, and it is desirable that they also be complete.

  Rules for proof are more concerned with utility. The smaller a set of defining rules we have, the more easily can we show it to be sound. In general, as long as the search problem is manageable, the larger the set of rules for proof we have the easier proofs will be.
- A simple meta language for proof rules helps users understand, choose and apply rules. Context information is best handled by tools rather than by direct manipulation. This leads, in the case of RSL, to incompleteness for local-expressions, which contain definitions.

There are some questions about how we achieve completeness and soundness for a language as large as RSL. This is not to claim RSL to be particularly large, merely that any rich language will have similar problems.

5.2 Soundness

The defining proof rules can either be asserted as the true definition, and so declared as sound a priori, or else proved against the denotational semantics [27]. In the first case it would still be desirable to check the denotational semantics against the proof rules. But it is difficult to see how this can be done in any but an informal manner. The denotational semantics is some 350 pages of formulae, and there are a number of known errors in it (and an unknown number of others!) It is, however, thought that there are no substantial problems with this document, and that the errors could be "fixed" without radical change. One can therefore take the view that the defining proof rules form an axiomatic semantics, and the denotational semantics provide the evidence of existence of a model satisfying these properties. This is largely the view taken in the book on the RAISE method [32], where specifications are described in terms of their signatures and logical properties, and refinement between specifications correspondingly defined in terms of signature inclusion and property entailment. It is a useful feature of RSL that its logic is powerful enough to itself express the properties needed to show refinement.

5.3 Completeness

Completeness for a set of proof rules is rather easier to deal with. First, there are language constructs that are defined in terms of others. So, for example, the five kinds of value definition in the language can all be expressed as one, a signature plus an axiom.

For the constructs that are left one defines a collection of "normal forms" and adds rules to show how other forms may be made normal. Then the operators are defined in terms of the normal forms. This is the way the defining rules for RSL were written.

Completeness is relative, of course, to rules for the built-in types, including **Bool**, **Int**, and **Real**. The definitional rules ignore rules for these types completely. In proof one would expect to use standard definitions of such types.

5.4 Proof via Translation

The original RAISE tools mostly discussed in this paper are being superseded by a new tool which is much more portable [11]. The new tool provides support for proof via translation [8] to PVS [30], thus making available the power of the PVS proof engine. This approach, based on a "shallow" embedding of RSL into PVS, can only provide a limited proof system because PVS is applicative — imperative and concurrent RSL is excluded. PVS also has a different logic from RSL that excludes undefinedness and nondeterminism (the "every expression denotes a value" approach described in section 2), so special care in the translation, including the generation of extra lemmas based on confidence conditions, is needed to ensure the translation is sound. A "deep" embedding into PVS would entail the modelling of RSL's semantics in the PVS logic. This may be possible, and might be useful for exercises in checking RSL proof rules, but would be unlikely to produce a tool useful in practice for performing proofs about RSL specifications.

The PVS translator does not need many proof rules for RAISE, because the target constructs of the translation are either built in to PVS (like arithmetic), defined by built-in expansions (like abstract data types), or defined in the PVS prelude (like sets and lists). A few additional constructs (including maps) are defined in an "RSL prelude". This only contains a few theorems, most of the definitions being constructive.

As an exercise the applicative proof rules of the justification editor — just over 1000 — were (by hand, but using a number of emacs macros) translated into PVS and proved. These rules turned out to contain 11 erroneous rules, arising from 6 mistakes. Two of these were inadequate side conditions allowing division by zero. Since this is technically undefined in RSL the rules could not lead to contradictions. But the other 4 mistakes, involving 8 rules, were formally unsound. The difficulties of ensuring correctness of formal systems of any size is again illustrated, as is the critical importance of using tools to gain confidence.

The use of PVS and its proof engine improves the capability of automatic proof, which was a weakness of the justification editor. It may be possible to improve this further with special tactics designed for RSL (especially for proving confidence conditions). PVS also provides a possibility of replaying proofs after changes to the specification, a feature of the justification editor that was never implemented. Inventing proofs initially is often hard, but redoing them by hand after changes is extremely tedious.

In another project, proof support is provided via a translation from an applicative subset of RSL to Isabelle/HOL [29]. This translation is based on an institution representation from an institution of RSL to an institution of Higher Order Logic and proved sound with respect to the denotational semantics of RSL. The concept of institutions is described in [35].

REFERENCES

[1] Abrial, J. R. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Barringer, H., Cheng, J. H., and Jones, C. B. A logic covering undefinedness in program proofs. *Acta Informatica 21* (1984), 251–269.

[3] Bolignano, D., and Debabi, M. Higher Order Communicating Processes with Value-passing, Assignment and Return of Results. In *Proceedings of ISAAC '92* (1992), no. 650 in Lecture Notes in Computer Science, Springer-Verlag.

[4] Bolignano, D., and Debabi, M. RSL: An Integration of Concurrent, Functional and Imperative Paradigms. Tech. Rep. LACOS/BULL/MD/3/V12.48, Bull, 1993.

[5] Burstall, R., and Goguen, J. The semantics of CLEAR: a specification language. In *Proceedings of Advanced Course on Abstract Software Specifications* (1980), no. 86 in Lecture Notes in Computer Science, Springer-Verlag, pp. 292–332.

[6] Dandanell, B., Gørtz, J., Pedersen, J. S., and Zierau, E. Experiences from Applications of RAISE. In *Proceedings of FME'93* (1993), no. 670 in Lecture Notes in Computer Science, Springer-Verlag.

[7] Dang Van, H., George, C., Janowski, T., and Moore, R. *Specification Case Studies in RAISE.* FACIT. Springer-Verlag, 2002.

[8] Dasso, A., and George, C. W. Transforming RSL into PVS. Technical Report 256, UNU/IIST, P.O. Box 3058, Macau, May 2002.

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

[10] Futatsugi, K., Goguen, J., Jouannaud, J.-P., and Meseguer, J. Principles of OBJ-2. In *12th Ann. Symp. on Principles of Programming* (1985), ACM, pp. 52–66.

[11] George, C. RAISE Tools User Guide. Technical Report 227, UNU/IIST, P.O. Box 3058, Macau, February 2001. The tools are available from `http://www.iist.unu.edu`.

[12] George, C., and Prehn, S. The RAISE Justification Handbook. Tech. Rep. LACOS/CRI/DOC/7, Computer Resources International, 1994.

[13] George, C., and Yong, X. An Operational Semantics for Timed RAISE. In *FM'99: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, Volume 2* (September 1999), J. M. Wing, J. Woodcock, and J. Davies, Eds., no. 1709 in Lecture Notes in Computer Science, Springer-Verlag, pp. 1008–1027.

[14] Guttag, J., Horning, J., and Wing, J. Larch in five easy pieces. Tech. Rep. 5, DEC SRC, Digital Equipment Corporation System Research Center, Palo Alto, California, USA, 1985.

[15] Haxthausen, A., and Yong, X. A RAISE Specification Framework and Justification Assistant for the Duration Calculus. In *Proceedings of ESSLLI-98 Workshop on Duration Calculus* (1998), pp. 51–58.

[16] Haxthausen, A., and Yong, X. Linking DC together with TRSL. In *Proceedings of 2nd International Conference on Integrated Formal Methods (IFM'2000), Schloss Dagstuhl, Germany, November 2000* (2000), no. 1945 in Lecture Notes in Computer Science, Springer-Verlag, pp. 25–44.

[17] Haxthausen, A. E., and Gjaldbæk, T. Modelling and Verification of Interlocking Systems for Railway Lines. In *Proceedings of 10th IFAC Symposium on Control in Transportation Systems* (2003), Elsevier Science Ltd.

[18] Haxthausen, A. E., and Peleska, J. Formal Development and Verification of a Distributed Railway Control System. *IEEE Transaction on Software Engineering 26*, 8 (2000), 687–701.

[19] Haxthausen, A. E., and Peleska, J. A Domain Specific Language for Railway Control Systems. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California* (June 23-28 2002).

[20] Hoare, C. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[21] Jones, C. B. *Systematic Software Development Using VDM*, second ed. Prentice-Hall International, Englewood Cliffs, New Jersey, 1990. ISBN 0-13-880733-7.

[22] Li Li, and He Jifeng. A Denotational Semantics of Timed RSL using Duration Calculus. In *RTCSA'99: Proceedings of The Sixth International Conference on Real-Time Computing Systems and Applications* (December 1999), IEEE Computer Society Press, pp. 492–503.

[23] Lindegaard, M. P., Viuf, P., and Haxthausen, A. E. Modelling Railway Interlocking Systems. In *Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13-15, 2000, Braunschweig, Germany* (2000), pp. 211–217.

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

[25] Milne, R. E. Semantic Foundations of RSL. Tech. Rep. RAISE/CRI/DOC/4/V1, CRI A/S, 1990.

[26] Milne, R. E. The Proof Theory for the RAISE Specification Language. Tech. Rep. RAISE/STC/REM/12/V3, STC Technology Ltd, 1990.

[27] Milne, R. E. The Formal Basis for the RAISE Specification Language. In *Semantics of Specification Languages* (Utrecht, 1993), D. Andrews, J. Groote, and C. Middelburg, Eds., Workshops in Computing, Springer-Verlag.

[28] Milner, R. *Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[29] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. No. 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[30] Owre, S., Rushby, J. M., and Shankar, N. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY, June 1992), D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 748–752.

[31] RAISE LANGUAGE GROUP, T. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.

[32] RAISE METHOD GROUP, T. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995. Available by ftp from `ftp://ftp.iist.unu.edu/pub/RAISE/method_book`.

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

[34] STODDART, B., DUNNE, S., AND GALLOWAY, A. Undefined expressions and logic in Z and B. *Formal Methods in System Design: An International Journal 15*, 3 (November 1999), 201–215.

[35] TARLECKI, A. INSTITUTIONS: An abstract framework for formal specifications. In *Algebraic Foundations of Systems Specification. IFIP state-of-the-art report*. Springer-Verlag, 1999.

[36] WIRSING, M. *A Specification Language*. PhD thesis, Techn. Univ. of Munich, FRG, 1983.