AMoL: **A** [Domain] **Mo**deling **L**anguage

Dines Bjørner
Techncal University of Denmark
Fredsvej 11, DK-2840 Holte
bjorner@gmail.com, https://www.imm.dtu.dk/~dibj/

October 12, 2024

Abstract

We outline the specfication languages, **AMoL**, which is used in describing [i.e., modeling] domains.

Contents

1 Introduction

"To a man with a hammer, everything looks like a nail."

– Mark Twain

To a computing scientist with a firmly established formal specification language, like `VDM SL` [4–6] or `RSL` [7], every domain can be formalized.

So, with `RSL`, the `RAISE` method's [8] formal specification language, I have the foundation for a domain description language.

In this technical note I wish to explore the languages, informal and formal, that "go into" the analysis and description of domains.

2 Values and Types

At the basis of every formal functional specification language are:

- The **value**s denoted by $\mathcal{E}$xpressions [and $\mathcal{S}$tatements] of these languages, and thus
- their **type**s [**sort**s].

2.1 Values and Operators

These are the values and operators that can be expressed in **AMoL**:

- **Atomic:** Atomic values are those for which it is meaningless to talk about their composition from "other" values.
  - **Booleans:** There are two truth values: **false** and **true**.
    Operators are the usual $\sim$ [negation], $\wedge$ [conjunction, "and"], $\vee$ [disjunction, "or"], $\Rightarrow$ [implication, "if ... then ..."], and $\equiv$ [identity, "if and only if"].
  - **Numbers:**
    - **Natural Numbers:** The positive natural numbers: 0, 1, ...
      Operators are: -, +, *, /, =, $\neq$, $\geq$, $\leq$.
    - **Integers:** The negative and positive "whole" numbers: ..., -1, 0, 1, ... Operators are: -, +, *, /, =, $\neq$, $\geq$, $\leq$.
    - **Reals:** Real numbers include rational numbers like positive and negative integers, fractions, and irrational numbers. In other words, any number that we can think of, except complex numbers, is a real number. For example, 3, 0, 1.5, 3/2, 5, and so on are real numbers.
      Operators are: -, +, *, /, =, $\neq$, $\geq$, $\leq$, $\lceil$, $\rceil$, **abs**.
      In mathematics, a real number is a number that can be used to measure a continuous one-dimensional quantity such as a distance, duration or temperature. Here, continuous means that pairs of values can have arbitrarily small differences.[a] Every real number can be almost uniquely represented by an infinite decimal expansion `[Wikipedia]`.
      We shall seldomly use reals in domain descriptions.
  - **Characters:** 'a', 'b', ..., 'z', 'A', 'B', ..., 'Z', '0', '1', ...
    Operators are =, $\neq$.
    We shall seldomly use characters in domain descriptions.
  - **Texts:** Sequences of characters: "a", "aa", ..., "abc...", "A", ..., "CMq59ABc", ...
    Operators are =, $\neq$.
    We shall seldomly use characters in domain descriptions.
- **Composite:**
  - **Sets:** Sets are here considered as sets in the usual mathematical sense: finite, possibly zero, or infinite collections of distinct values as expressed by $\{a, b, c, ...\}$, where a, b, c, ... stand for distinct values.
    Operators are: =, $\neq$, $\in$, $\cup$, $\cap$, **card**.
  - **Cartesians:** Cartesians are finite "groupings" of two or more values as expressed by $(a, b, ..., c)$, where a, b, c, ... stand for not necessarily distinct values.
    Operators are: =, $\neq$.
  - **Lists:** Lists are finite, possibly zero, or infinite "sequences" of zero or more values as expressed by $\langle a, b, ..., c\rangle$, where a, b, c, ... stand for not necessarily distinct values.
    Operators are: =, $\neq$, **len**, **hd**, **tl**, .[.].

– **Maps:** Maps are discrete, finite definition set functions from non-function values to values: [a↦b,c↦d,...,y↦z].
Operators are: =, ≠, **dom**, **rng**, ∪, †, ↦

– **Functions:** Functions can be explained as follows. There are *definition sets* and there are *range sets*. A function is then "something", which "maps" definition set elements to range set elements. If there are definition set elements for which the "mapping" is not defined, then the function is said to be *partial*, otherwise it is *total*. That is: a function is "something" which, when applied to an element of its definition set "yields" an element of its range set.[1]
The only operator is: .(.), i.e., function application. We cannot have functions, say $\mathcal{D}$[efinition set] and $\mathcal{R}$[ange set], which yields these quanties: it is undecidable.

2.2 Types and Sorts

2.2.1 Atomic Type Names

Values can be ascribed types. That is: We can give names, i.e., type names, to certain collections of values. To atomic values the type names are the literals:

- **Boolean**,
- **Int**,
- **Char** and
- **Nat**,
- **Real**,
- **Text**.

2.2.2 Type Operators

Let T, T1, T2, ..., Tm be type names. Then these are the type operators:

- **-set**, suffix,
- **-infset**, suffix,
- ×, infix,
- *, suffix,
- ω, suffix,
- $\overrightarrow{m}$, infix,
- →, infix,
- $\xrightarrow{\sim}$, infix,
- |, infix,
- {| ... | ... |}, distributed fix.

2.2.3 Type Expressions

Let T, T1, T2, ..., Tm be type names. Then the following are *type expressions:*

- (a) T**-set** stands for the collection of all finite sets of T elements;
- (b) T**-infset** stands for the collection of all finite and infinite sets of T elements;
- (c) T1×T2×...×Tm stands for the collection of all Cartesians (groupings) of T1, then T2, ... finally Tm elements;
- (d) T* stands for the collection of all finite length lists of T elements;
- (e) Tω stands for the collection of all finite and infinite length lists of T elements;
- (f) T1 $\overrightarrow{m}$ T2 stands for the collection of all finite maps from T1 into T2;
- (g) T1→T2 stands for the collection of all total functions fromT1 into T2;
- (h) T1$\xrightarrow{\sim}$T2 stands for the collection of all partial functions from T1 into T2;

[1] We justify classifying functions as composite values in that we consider each "pairing", (d, r), of a definition set element d and a range set element r as a composable pair.

- (i) T1|T2 stands for the collection of all T1 and T2; and
- (j) {| v | v:T • $\mathcal{P}$(v) |} stands for the **sub-type** of T for which the predicate $\mathcal{P}$ holds.

The operand Ts may be type expressions – as are the type identifiers.

3 RSL Definition Units

In this section we shall unfold some core aspects of a formal specification languge, the RSL [7].

The presentation is "traditional": It describes certain language constructs from the point of viw of the language.

The **AMoL** domain description language builds on RSL. In this section we list, and briefly explain, the RSL clauses, here referred to as the RSL definition units, that form, so-to-speak, an RSL "core" that **AMoL** builds upon.

This "core" can be summarized as the

- **type** definitions,
- **value** definitions,
- **axiom** definitions,
- **variable** declarations and the
- **channel** declarations.

3.1 Type Definitions

So far we have introduced the concepts of values and types. Now we show how to introduce named types into a specification.

3.1.1 Abstract Types, Sorts

When, in a specification, we express:

type T

we mean to let the type identifier T stand for an "abstract type", called a *sort*. A *sort* is a collection of values. That collection and those values are not further described.

3.1.2 Concrete Types

When, in a specification, we express:

type T = Type_Expression

where Type_Expression is one of the forms outlined in Sect. 2.2.3 on page 6. that is, we mean to let the type identifier T stand for an "concrete type".

3.2 Value Definitions

Let v be identifier and T be a type expression, usually a type identifier. Then

value v:T
value v:T = $\mathcal{E}$, where $\mathcal{E}$ is an RSL expression whose evaluation yields a value named v

introduces the v as a value of type T.

3.2.1 Function Signatures

Let f be an identifier A, B, ..., C and P, Q, ..., R be type expressions, usually "just" type identifiers. Then

value f:(A×B×...×C) → (P×Q×...×R)

introduces f as a total (→) function whose *signature* is (A×B×...×C) → (P×Q×...×R). Similar for partial functions $\xrightarrow{\sim}$.

3.3 Axiom Definitions

An axiom, postulate, or assumption is a statement that is taken to be true, to serve as a premise or starting point for further reasoning and arguments. The word comes from the Ancient Greek word $\alpha\xi\iota\omega\mu\alpha$ (axíŏma), meaning 'that which is thought worthy or fit' or 'that which commends itself as evident'.

axiom P – predicate expression

The predicate, $\mathcal{P}$, usually expresses a type or a value constraint.

3.4 Variable Declaration

Variables are seldom used. Typically, in a domain description, there are two variables. One to contain all the *parts* of a domain, another to contain all the *unique identifiers* of those parts.

variable v:T := $\mathcal{E}$

3.5 Channel Declarations

A channel is a further undefined "medium". Channels (in `CSP` and in `RSL-Text`) communicate messages sent by behaviours to behaviours We can, for example, have just two behaviours in our specification, so their channel would be that declared in a., or we could have that one behaviour communicates with either of a number of other behaviours, so the communication channel would be that declared in b., or we could have that any number of behaviours communicates with either of a number of other behaviours, so the communication channel would be that declared in c.

a. **channel** ch T
b. **channel** { ch[i] | i:I • ... } T
c. **channel** { ch[{i,j}] | i,j:UI • ... } T

declare ch to be an array of channels. i, j are unique identifier indices. The channels carry "messages" of type T between behaviours.

4 A Domain Analysis & Description Ontology

By a *domain* we shall understand a *rationally describable* segment of a *discrete dynamics* fragment of a *human assisted* reality: the world that we daily observe – in which we work and act, a reality made significant by human-created entities. The domain embody *endurants* and *perdurants*.

AMoL is the formal specification language in which we describe domains. The description is based on the use of *analysis and description* prompts. These prompts "derive" from the domain ontology as shown in Fig. 1. We shall briefly explain this figure.

In this section we shall reveiew the RSL-like language constructs used in describing domains – see from the point of view of the specific domain ontology of [**?**, 1–3].

4.1 A Domain Description Ontology

Figure 1 shows an ontology for describing domains. Bulleted (•) items refer to concepts that are thus "structurally related" by the ontology.

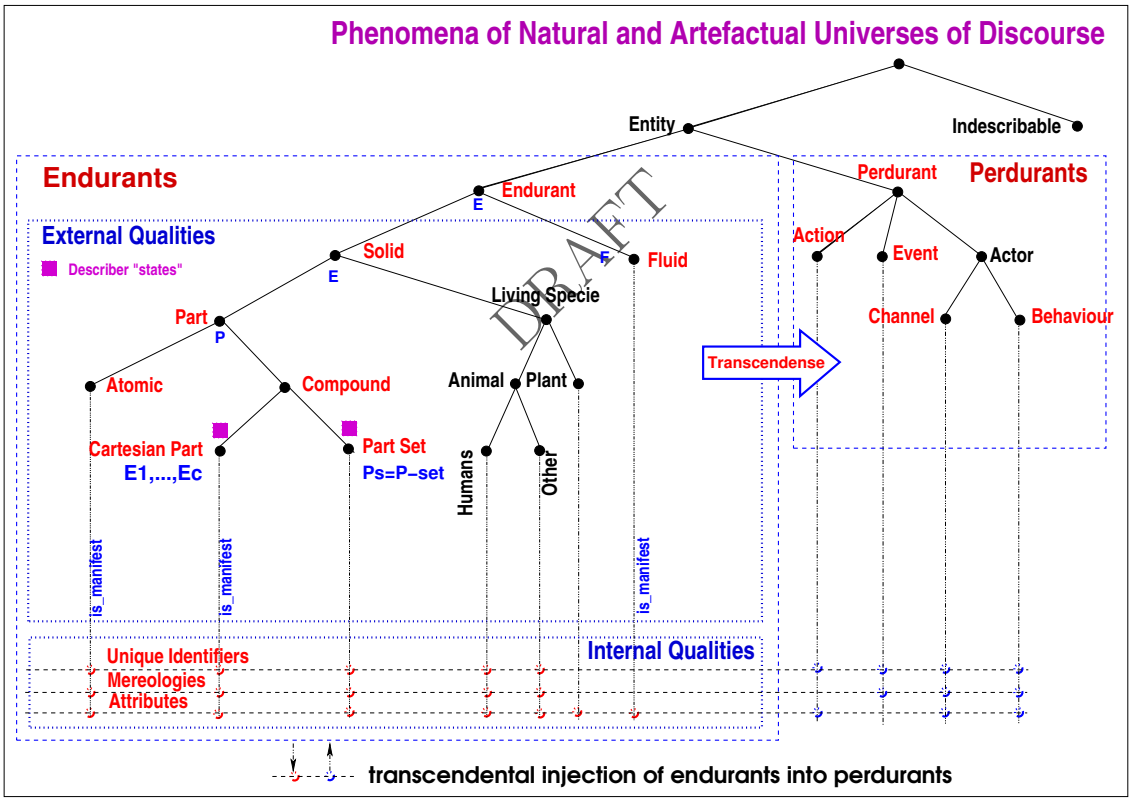

Figure 1: A Domain Analysis & Description Ontology

The idea of Fig. 1 is the following:

- It presents a recipe for how to **analyze** a domain.
- You, the *domain* **analyzer** *cum describer*, are *'confronted'*[2] with, or by a domain.
- You have Fig. 1 in front of you, on a piece of paper, or in Your mind, or both.

[2]By 'confronted' we mean: You are reading about it, in papers, in books, in postings on the Internet, visiting it, talking with domain stakeholders: professional people working "in" the domain; You may, yourself, "be an entity" of that domain!

- You are then "asked"/"urged"/"prompted", by the domain **analysis** & description method, to "start" at the uppermost •, just below and between the '**r**' and the first '**s**' in the main title, Phenomena of Natural and Artifactual Universes of Discourse.
- The **analysis** & description ontology of Fig. 1 then *directs* You to inquire as to whether the phenomenon – whichever You are "looking at/reading about/..." – is either *rationally describable*, i.e., is an *entity* (`is_entity`) or is *indescribable*.
- That is, You are, in general, "positioned" at a bullet, •, labeled α, "below" which there may be two alternative bullets, one, β, to the right and one to the left, γ.
- It is Your decision whether the answer to the "query" that each such situation warrants, is yes, `is_`β, or no, `is_`γ.
- The characterizations of the concepts whose names, α, β, γ etc., are attached to the •s of Fig. 1 are given in [**?**, 1–3].
- Whether they are precise enough to guide You in Your obtaining reasonable answers, "yes" or "no", to the •ed queries is, of course, a problem. I hope they are.
- If Your answer is "yes", then Your **analysis** is to proceed "down the tree", usually indicated by "yes" or "no" answers.
- If one, or the other is a "leaf" of the ontology tree, then You have finished examining the phenomena You set out to **analyze**.
- If it is not a leaf, then further **analysis** is required.
- (We shall, in this paper, leave out the analysis and hence description of *living species*.)
- If an **analysis** of a phenomenon has reached one of the (only) two ▪'s, then the **analysis** at that • results in the domain describer **describing** some of the properties of that phenomenon.
- That **analysis** involves "setting aside", for subsequent **analysis & description**, one or more [thus **analysis** etc.-pending] phenomena (which are subsequently to be tackled from the "root" of the ontology).

We do not [need to] prescribe in which order You analyze & describe the phenomena that has been "set aside".

4.2 Analysis Predicates

- `is_entity`,
- `is_endurant`,
- `is_perdurants`,
- `is_solid`,
- `is_fluid`,
- `is_part`,
- `is_living_species`,
- `is_atomic`,
- `is_compound`,
- ○ `is_Cartesian`,
- ○ `is_part_set`,
- etc.

Satisfaction of the ○'ed `is_Cartesian` and the `is_part_set` predicates "triggers" the the respectively ○'ed analysis function mentioned next.

4.3 Analysis Functions

Analysis functions yield type names.

- ○ `record_Cartesian_part_type_names: P → ηP1×ηP2×...×ηPn`,
- ○ `record_part_set_part_type_names: P → ηPS×ηPE`, and
- • `record_attribute_type_names: P → A1×A2×...×Am`

where P is the part being observed. If Cartesian, then P1, P2, ..., Pn are the types of its observed components. If part set, the PS are the types of its set of parts, and these are of type PE. The Ais are the types chosen to be attributes of P.

4.4 Description Prompts

4.4.1 Cartesian Parts

- `Describe Cartesian Parts: P → RSL-Text`

Describe_Cartesian _Parts

```
1   let (ηP1,ηP2,...,ηPn) = record_Cartesian_part_type_names(p) in
2   "type P1, P2, ..., Pn
3    value obs_P1: P→P1, obs_P2: P→P2, ..., obs_Pn: P→Pn "
4   end
```


Lines 1 and 4 are expessed in the domain analysis language, lines 2 and 3 in `RSL-Text`.

4.4.2 Part Set Parts

- `Describe_Part_Set_Parts P → RSL-Text`

Describe_Part_Set_Parts

```
1   let (ηPS,ηPE) = record_part_set_part_type_names(p) in
2   "type PS = PE-set, PE
3    value obs_P: P→PS"
4   end
```

Lines 1 and 4 are expessed in the domain analysis language, lines 2 and 3 in `RSL-Text`.

4.4.3 Unique Identification

Describe_Unique_Identifier

```
1   "type PUI
2    value uid_P: P→PUI"
```

Line 1 expresses that PUI is a sort. Line 2 expresses that parts p:P have unique identifiers of sort PUI.

4.4.4 Mereology

Describe_Mereology

```
1   "type M = 𝓜(UI1,UI2,,...,UIm)
2    value " mereo_P: P → M
```

Line 1 expresses that mereology type M is a concrete type over unique identifier types.

4.4.5 Attributes

Describe_Attributes

```
1   let {A1,A2,...,An} = record_attribute_type_nmes(p) in
2   "type A1, A2, ..., An
3    value attr_A1: P → A1, retr_A2: P → A2, ..., attr_An: P → An"
4   end
```

Lines 1 and 4 are expressed in the domain analysis language. Lines 2 and 4 3 in `RSL-Text`.

4.4.6 Channels

Describe_Attributes

```
1   channel { ch[ {i,j} ] | i,j:UI • 𝓒(...) }"
```

Line 1 expresses that th channel is a [triangular] channel array over all the [unordered] unique identifiers of the domain.

4.4.7 Behaviour Signature

Describe_Attributes

```
1  value
2      ℬ[p:P]:
3          UI
4          → mereo_P(p)
5              → static_attrs(p)
6                  → monitorable_attrs(p)
7                      → programmable_attrs(p)
8                          → Unit"
```

Line 1 expresses a value. Lines 2–6 that this value is a function.
Line 8 that the function "newer ends", i.e., never "returns, yields" a typed value.
Line 2 expresses the name $\mathcal{B}_{p:P}$ of the function;, that is that it "derives" from part p of type P.
Lines 3–7 expresses that the function has a number, four, of arguments and that these are expressed in the *Schönfinckel*'ed, i.e., the *Curried* manner.
Line 3 expresses that each behaviour over parts p of type P is unique.
Line 4 expresses that the mereology of p is an argument.
Lines 5–7 expresses that three groups of attributes are separate aguments. We do not define the selector functions. They are defined in [**?**, 1–3].

4.4.8 Behaviour Functionality

Describe_Behaviour_Functionality

```
1   value
2       "B_{p:P}(ui)(mereo)(sta_attrs)(mon_attrs)(pro_attrs) ≡
3           let pro_attrs' = C_{p:P}(ui)(mereo)(sta_attrs)(mon_attrs)(pro_attrs) in
4           B_{p:P}(ui)(mereo)(sta_attrs)(mon_attrs)(pro_attrs') end"
```

Line 2 expresses the invocation of behaviour $\mathcal{B}_{p:P}$.
Line 3 expresses the elaboration of clause $\mathcal{C}_{p:P}$ with the arguments of the invoking $\mathcal{B}_{p:P}$.
Line 4 expresses the tail-recursive ["never-ending"] invocation of $\mathcal{B}_{p:P}$ with the updated programmable attributes.

4.4.9 Domain Initialization

Describe_Domain_Initializaion

```
B_{p:P}(uid_P(p))(mereo_P(p))(stat_attrs(p))(moni_attrs(p))(prog_attrs(p))
```

5 AMoL Description Units

In this section we shall reveiew the RSL-like language constructs used in describing domains – seen from the point of view of the RSL [7].

An **AMoL** domain description consists of a set (textually ordered in any linear sequence) of *domain description units*. We shall only treat five kinds of such units.

Domain descriptions focus on

The external qualities:

- **endurants**,
- **states**,

the internal qualities:

  - **unique identification**
    - **states**,
  - **mereology**,
  - **attributes**,

- the perdurants:
  - **channels**,
  - **behaviour signatures**, and
  - **behaviour functionality [definitions]**.

The **AMoL** description units

5.1 AMoL Sort Specifications

5.1.1 The Universe of Discourse

Prefixed by the keyword (literal) **type**, type specification units introduce distinct type names.

The basic form of a type specification unit is:

type T

where T is a distinct (type) identifier.

5.1.2 The Composite Endurant Sorts

Let T stand for a sort of composite parts. Then the

type
  T, T1, T2, ... Tn
value
  obs_T1 : T→ T1, obs_T2 : T → T2, ..., obs_Tn : T → Tn

For more on types, see Sect. A.1.

5.2 AMoL Value Specifications

Prefixed by the keyword (literal) **value** value specification units introduce distinct value names.

The basic form of a value specification unit is:

value a:A

The value, a, is further unspecified. It is of type A, but which ... !

The basic form of a value specification unit is:

value a:A = $\mathcal{E}$(...)

where $\mathcal{E}$(...) is a value expression.

Value specification units may introduce several (new, distinctly named) values:

value a_1:A_1 = $\mathcal{E}_1$(...), a_2:A_2 = $\mathcal{E}_2$(...), ..., a_n:A_n = $\mathcal{E}_n$(...)

Quite often the value specification is of the form:

value f: A → B, f(a) ≡ $\mathcal{E}$(..,a,.) or f = λa.$\mathcal{E}$(..,a,.)

that is: the value is a function, f, whose *signature* gives the name, f, and the type of the functionality A → B (or A $\xrightarrow{\sim}$ B).[3]

5.3 AMoL Axiom Specifications

Prefixed by the keyword (literal) **axiom**, axiom specification units serve to limit values. The general form of an axiom specification unit is:

axiom $\mathcal{A}$(...)

where $\mathcal{A}$(...) is some predicate expression over (specification unit) defined quantities. For more on axiomatic expressions, see Sect. A.2.

5.4 AMoL Variable Specifications

Prefixed by the keyword (literal) **variable**, variable specification units introduce distinct variable names. The general form of a variable specification unit is:

variable v:T := expression

where v is ..., T is ..., and expression is a value expression. For variables, see Sect. A.8.2.

5.5 AMoL Variable Specifications

Prefixed by the keyword (literal) **channel**, channel specification units introduce distinct channel names.

The general form of a channel specification unit is:

channel { ch[{i,j}] | i,j:UI • ... } M

where ch is a distinct, here channel, name,we are declaring an array of channels. ch[$\{i, j\}$] expresses that , $\{i, j\}$ ranges of so-called unique identifier indices of type UI, and M is a type expression. For more on channels, see Sect. A.9.1.

[3] We use the identity sign ≡ [instead of =] to allow for f to be recursively defined.

6 Conclusion

7 Bibliography

7.1 Bibliographical Notes

References

[1] Dines Bjørner. Domain Science & Engineering – A Foundation for Software Development. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg, Germany, 2021. A revised version of this book is [3].

[2] Dines Bjørner. Domain Modelling – A Primer. A short version of [3]. xii+202 pages[4], May 2023.

[3] Dines Bjørner. Domain Science & Engineering – A Foundation for Software Development. Revised edition of [1]. xii+346 pages[5], January 2023.

[4] Dines Bjørner and Cliff B. Jones, editors. The Vienna Development Method: The Meta-Language, volume 61 of LNCS. Springer, Heidelberg, Germany, 1978.

[5] Dines Bjørner and Cliff B. Jones, editors. Formal Specification and Software Development. Prentice-Hall, London, England, 1982.

[6] John Fitzgerald and Peter Gorm Larsen. Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[7] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. The RAISE Specification Language. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[8] Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. The RAISE Development Method. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[9] James Gosling and Frank Yellin. The Java Language Specification. Addison-Wesley & Sun Microsystems. ACM Press Books, 1996. 864 pp, ISBN 0-10-63451-1.

[10] Michael Reichhardt Hansen and Hans Rischel. Functional Programming Using F#. Cambridge University Press, 2013.

[11] R. Milner, M. Tofte, and R. Harper. The Definition of Standard ML. The MIT Press, Cambridge, Mass., USA and London, England, 1990.

[12] Peter Sestoft. Java Precisely. The MIT Press, 25 July 2002.

[13] N. Wirth. The Programming Language Oberon. Software — Practice and Experience, 18:671–690, 1988.

[4] This book is currently being translated into Chinese by Dr. Yang ShaoFa, IoS/CAS (Institute of Software, Chinese Academy of Sciences), Beijing and into Russian by Dr. Mikhail Chupilko and his colleagues, ISP/RAS (Institute of Systems Programming, Russian Academy of Sciences), Moscow

[5] Due to copyright reasons no URL is given to this document's possible Internet location. A primer version, omitting certain chapters, is [2]

A A RAISE Specification Language Primer

We present an RSL Primer. Indented text, in slanted font, such as this, presents informal material and examples. Non-indented text, in roman font, presents narrative and formal explanation of RSL constructs.

This RSL Primer omits treatment of a number of language constructs, notably the RSL module concepts of *schemes*, *classes* and *objects*. Although we do cover the imperative language construct of [declaration of] variables and, hence, assignment, we shall omit treatment of structured imperative constructs like **for** ..., **do** s **while** b, **while** b **do** s loops.

Section A.12 on page 39 introduces additional language constructs, thereby motivation the $^+$ in the RSL$^+$ name.

A.1 Types and Values

Types are, in general, set-like structures[6] of things, i.e., values, having common characteristics.

A bunch of zero, one or more apples (type *apples*) may thus form a [sub]set of type *Belle de Boskoop* apples. A bunch of zero, one or more pears (type *pears*) may thus form a [sub]set of type *Concorde* pears. A union of zero, one or more of these apples and pears then form a [sub]set of entities of type *fruits*.

A.1.1 Sort and Type Expressions

Sort and type expressions are expressions whose values are types, that is, possibly infinite set-like structures of values (of "that" type).

A.1.1.1 Atomic Types: Identifier Expressions and Type Values Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully "taken apart".

RSL has a number of [so-called] built-in atomic types. They are expressed in terms of literal identifiers. These are the **Boolean**s, **int**egers, **Nat**ural numbers, **Real**s, **Char**acters, and **Text**s. **Text**s are free-form texts and are more general than just texts of RSL-like formulas. RSL-Text's will be introduced in Sect. A.12 on page 39.

We shall not need the base types **Char**acters, nor the general type **Text**s for domain modelling in this primer. They will be listed below, but not mentioned further.

The base types are:

Basic Types

```
type
  [1] Bool
  [2] Int
  [3] Nat
  [4] Real
  [5] Char
  [6] Text
```

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...

[6] We shall not, in this primer, go into details as to the mathematics of types.

4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period (“.”), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″bbb″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

A.1.1.2 Composite Types: Expressions and Type Values Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully “taken apart”.

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then these are the composite types, hence, type expressions:

Composite Type Expressions

[7] A-**set**
[8] A-**infset**
[9] A × B × ... × C
[10] A^*
[11] A^ω
[12] A $\xrightarrow[m]{}$ B
[13] A → B
[14] A $\xrightarrow{\sim}$ B
[15] A | B | ... | C
[16] mk_id(sel_a:A,...,sel_b:B)
[17] sel_a:A ... sel_b:B

1

Section A.12 on page 39 introduces the extended RSL concepts of type name values and the type, $\mathbb{T}$, of type names.

A.1.2 Type Definitions

A.1.2.1 Sorts — Abstract Types Types can be (abstract) sorts in which case their structure is not specified:

Sorts

type
A, B, ..., C

A.1.2.2 Concrete Types Types can be concrete in which case the structure of the type is specified by type expressions:

Type Definition

type
A = Type_expr

RSL Example: Sets. Narrative: *H* stand for the domain type of street intersections – we shall call then hubs, and let *L* stand for the domain type of segments of streets between immediately

neighboring hubs – we shall call then links. Then Hs and Ls are to designate the types of finite sets of zero, one or more hubs, respectively links. **Formalisation:**

type H, L, Hs=H-**set**, Ls=L-**set** •

RSL Example: Cartesians. Narrative: Let RN stand for the domain type of road nets consisting of hub aggregates, HA, and link aggregates, LA. Hub and link aggregates can be observed from road nets, and hub sets and link sets can be observed from hub, respectively link aggregates. **Formalisation:**

type RN = HA×LA, Hs, Ls
value obs_HA: RN→HA, obs_LA: RN− LA, obs_Hs: HA→Hs, obs_Ls: LA→Ls

Observer functions, obs_... are not further defined – beyond their signatures. They will (subsequently) be defined through axioms over their results •
Some schematic type definitions are:

Variety of Type Definitions

```
[18] Type_name = Type_expr /∗ without |s or subtypes ∗/
[19] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[20] Type_name ==
         mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
         ... |
         mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[21] Type_name :: sel_a:Type_name_a  ...  sel_z:Type_name_z
[22] Type_name = {| v:Type_name′ • P(v) |}
```


where a form of [19–20] is provided by combining the types:

Record Types

```
[23] Type_name = A | B | ... | Z
[24] A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
[25] B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
[26] ...
[27] Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Of these we shall almost exclusively make use of [23–27].

Disjoint Types. Narrative: A pipeline consists of a finite set of zero, one or more [interconnected][7] pipe units. Pipe units are either wells, or are pumps, or are valves, or are joins, or are forks, or are sinks. **Formalisation:**

type PL = P-**set**, P == WU|PU|VA|JO|FO|SI, Wu,Pu,Vu,Ju,Fu,Su
WU::mkWU(swu:Wu), PU::mkPU(spu:Pu), VA::mkVU(svu:Vu),
JO::mkJu(sju:Ju), FO::mkFu(sfu:Fu), SI::mkSi(ssu:Su)

where we leave types Wu, Pu, Vu, Ju, Fu and Su further undefined •
Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

axiom
∀ a1:A_1, a2:A_2, ..., ai:Ai •
s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧

```
∀ a:A • let mk_id_1(a1′,a2′,...,ai′) = a in
    a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) end
```

Note: Values of type A, where that type is defined by A::B×C×D, can be expressed A(b,c,d) for b:B, c:D, d:D.

A.1.2.3 Subtypes In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

Subtypes

```
type
  A = {| b:B • 𝒫(b) |}
```

Subtype. Narrative: The subtype of even natural numbers.
Formalisation: **type** ENat = {| en | en:**Nat** • is_even_natural_number(en) |} •

A.2 The Propositional and Predicate Calculi

A.2.1 Propositions

In logic, a proposition is the meaning of a declarative sentence. [A declarative sentence is a type of sentence that makes a statement]

A.2.1.1 Propositional Expressions Propositional expressions, informally speaking, are quantifier-free expressions having truth (or **chaos**) values. ∀, ∃ and ∃! are quantifiers, see below.

Below, we will first treat propositional expressions all of whose identifiers denote truth values. As we progress, in sections on arithmetic, sets, list, maps, etc., we shall extend the range of propositional expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]). Then:

Propositional Expressions

```
false, true
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b
```

are propositional expressions having Boolean values. ∼, ∧, ∨, ⇒, =, ≠ and □ are Boolean connectives (i.e., operators). They can be read as: **not**, **and**, **or, if then** (or **implies**), **equal, not equal** and **always**.

A.2.1.2 Propositional Calculus Propositional calculus is a branch of logic. It is also called propositional logic, statement logic, sentential calculus, sentential logic, or sometimes zeroth-order logic. It deals with propositions (which can be true or false) and relations between propositions, including the construction of arguments based on them. Compound propositions are formed by connecting propositions by logical connectives. Propositions that contain no logical connectives are called atomic propositions [Wikipedia]
A simple two-value Boolean logic can be defined as follows:

```
type
  Bool
```

value
  true, **false**
  $\sim$: **Bool** $\rightarrow$ **Bool**
  $\wedge$, $\vee$, $\Rightarrow$, $=$, $\neq$, $\equiv$: **Bool** $\times$ **Bool** $\rightarrow$ **Bool**
axiom
  $\forall$ b,b′:**Bool** •
    $\sim$b $\equiv$ **if** b **then false else true end**
    b $\wedge$ b′ $\equiv$ **if** b **then** b′ **else false end**
    b $\vee$ b′ $\equiv$ **if** b **then true else** b′ **end**
    b $\Rightarrow$ b′ $\equiv$ **if** b **then** b′ **else true end**
    b = b′ $\equiv$ **if** (b$\wedge$b′)$\vee$($\sim$b$\wedge\sim$b′) **then true else false end**
    (b $\neq$ b′) $\equiv$ $\sim$(b = b′)
    (b $\equiv$ b′) $\equiv$ (b = b′)

We shall, however, make use of a three-value Boolean logic. The model-theory explanation of the meaning of propositional expressions is now given in terms of the *truth tables* for the logic connectives:

$\vee, \wedge$, **and** $\Rightarrow$ **Syntactic Truth Tables**

$\vee$	**true**	**false**	**chaos**
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

$\wedge$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

$\Rightarrow$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

The two-value logic defined earlier 'transpires' from the **true,false** columns and rows of the above truth tables.

A.2.2 Predicates

Predicates are mathematical assertions that contains variables, sometimes referred to as predicate variables, and may be true or false depending on those variables' value or values[8]

A.2.2.1 Predicate Expressions Let x, y, ..., z (or term expressions) designate non-Boolean values, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ be propositional or predicate expressions, then:

Simple Predicate Expressions

[28] $\forall$x:X • $\mathcal{P}(x)$
[29] $\exists$y:Y • $\mathcal{Q}(y)$
[30] $\exists!$z:Z • $\mathcal{R}(z)$

are quantified, i.e., predicate expressions. $\forall$, $\exists$ and $\exists!$ are the quantifiers.

A.2.2.2 Predicate Calculus 1

[28–30] The predicates $\mathcal{P}(x)$, $\mathcal{Q}(y)$ or $\mathcal{R}(z)$ may yield **chaos** in which case the whole expression yields **chaos**.

[8] https://calcworkshop.com/logic/predicate-logic/, and: predicate logic, first-order logic or quantified logic is a formal language in which propositions are expressed in terms of predicates, variables and quantifiers. It is different from propositional logic which lacks quantifiers https://brilliant.org/wiki/predicate-logic/.

A.3 Arithmetics

RSL offers the usual set of arithmetic operators. From these the usual kind of arithmetic expressions can be formed.

Arithmetic

```
type
  Nat, Int, Real
value
  +,−,∗: Nat×Nat→Nat | Int×Int→Int | Real×Real→Real
  /: Nat×Nat~→Nat | Int×Int~→Int | Real×Real~→Real
  <,≤,=,≠,≥,> (Nat|Int|Real) → (Nat|Int|Real)
```

A.4 Comprehensive Expressions

Comprehensive expressions are common in mathematics texts. They capture properties conveniently abstractly

A.4.1 Set Enumeration and Comprehension

A.4.1.1 Set Enumeration Let the below a's denote values of type A:

Set Enumerations


```
{{}, {a}, {e1,e2,...,en}, ...} ∈ A-set
{{}, {a}, {e1,e2,...,en}, ..., {e1,e2,...}} ∈ A-infset
```

A.4.1.2 Set Comprehension The expression, last line below, to the right of the ≡, expresses set comprehension. The expression "builds" the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

Set Comprehension

```
type
  A, B
  P = A → Bool
  Q = A ~→ B
value
  comprehend: A-infset × P × Q → B-infset
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}
```

A.4.1.3 Cartesian Enumeration Let e range over values of Cartesian types involving A, B, ..., C, then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations

```
type
  A, B, ..., C
  A × B × ... × C
```

```
value
  (e1,e2,...,en)
```

A.4.2 List Enumeration and Comprehension

A.4.2.1 List Enumeration Let a range over values of type A, then the below expressions are simple list enumerations:

List Enumerations

```
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ...} ∈ A*
{⟨⟩, ⟨e⟩, ..., ⟨e1,e2,...,en⟩, ..., ⟨e1,e2,...,en,... ⟩, ...} ∈ Aω

⟨ a_i .. a_j ⟩
```

The last line above assumes a_i and a_j to be integer-valued expressions. It then expresses the set of integers from the value of e_i to and including the value of e_j. If the latter is smaller than the former, then the list is empty.

A.4.2.2 List Comprehension The last line below expresses list comprehension.

List Comprehension

```
type
  A, B, P = A → Bool, Q = A ~→ B
value
  comprehend: Aω × P × Q ~→ Bω
  comprehend(l,P,Q) ≡ ⟨ Q(l(i)) | i in ⟨1..len l⟩ • P(l(i))⟩
```


A.4.3 Map Enumeration and Comprehension

A.4.3.1 Map Enumeration Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively, then the below expressions are simple map enumerations:

Map Enumerations

```
type
  T1, T2
  M = T1 ~m~→ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [ ], [ u↦v ], ..., [ u1↦v1,u2↦v2,...,un↦vn ] ∀ ∈ M
```

A.4.3.2 Map Comprehension The last line below expresses map comprehension:

Map Comprehension

```
type
  U, V, X, Y
  M = U ~m~→ V
```

```
  F = U ~→ X
  G = V ~→ Y
  P = U → Bool
value
  comprehend: M×F×G×P → (X ~m~> Y)
  comprehend(m,F,G,P) ≡ [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

A.5 Operations

A.5.1 Set Operations

A.5.1.1 Set Operator Signatures

Set Operator Signatures

```
value
  7  ∈: A × A-infset → Bool
  8  ∉: A × A-infset → Bool
  9  ∪: A-infset × A-infset → A-infset
  10 ∪: (A-infset)-infset → A-infset
  11 ∩: A-infset × A-infset → A-infset
  12 ∩: (A-infset)-infset → A-infset
  13 \: A-infset × A-infset → A-infset
  14 ⊂: A-infset × A-infset → Bool
  15 ⊆: A-infset × A-infset → Bool
  16 =: A-infset × A-infset → Bool
  17 ≠: A-infset × A-infset → Bool
  18 card: A-infset ~→ Nat
```


A.5.1.2 Set Operation Examples

Set Operation Examples

```
examples
  a ∈ {a,b,c}
  a ∉ {}, a ∉ {b,c}
  {a,b,c} ∪ {a,b,d,e} = {a,b,c,d,e}
  ∪{{a},{a,b},{a,d}} = {a,b,d}
  {a,b,c} ∩ {c,d,e} = {c}
  ∩{{a},{a,b},{a,d}} = {a}
  {a,b,c} \ {c,d} = {a,b}
  {a,b} ⊂ {a,b,c}
  {a,b,c} ⊆ {a,b,c}
  {a,b,c} = {a,b,c}
  {a,b,c} ≠ {a,b}
  card {} = 0, card {a,b,c} = 3
```

A.5.1.3 Informal Set Operator Explication The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as

definitions, ≡, as [a kind of] identities.

7. ∈: The membership operator expresses that an element is a member of a set.
8. ∉: The nonmembership operator expresses that an element is not a member of a set.
9. ∪: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
10. ∪: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
11. ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
12. ∩: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
13. \: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
14. ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
15. ⊂: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
16. =: The equal operator expresses that the two operand sets are identical.
17. ≠: The nonequal operator expresses that the two operand sets are not identical.
18. **card**: The cardinality operator gives the number of elements in a finite set.

A.5.1.4 Set Operator Explications The set operations can be "equated" as follows:

Set Operator Explications

```
value
  s′ ∪ s″ ≡ { a | a:A • a ∈ s′ ∨ a ∈ s″ }
  s′ ∩ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∈ s″ }
  s′ \ s″ ≡ { a | a:A • a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A • a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A • a ∈ s″ ∧ a ∉ s′
  s′ = s″ ≡ ∀ a:A • a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
  card s ≡
    if s = {} then 0 else
    let a:A • a ∈ s in 1 + card (s \ {a}) end end
    pre s /∗ is a finite set ∗/
  card s ≡ chaos /∗ tests for infinity of s ∗/
```

A.5.2 Cartesian Operations

Cartesian Operations

type
A, B, C
g0: G0 = A × B × C
g1: G1 = (A × B × C)
g2: G2 = (A × B) × C
g3: G3 = A × (B × C)

value
va:A, vb:B, vc:C, vd:D
(va,vb,vc):G0,
(va,vb,vc):G1
((va,vb),vc):G2
(va3,(vb3,vc3)):G3

decomposition expressions
let (a1,b1,c1) = g0,
(a1′,b1′,c1′) = g1 **in** .. **end**
let ((a2,b2),c2) = g2 **in** .. **end**
let (a3,(b3,c3)) = g3 **in** .. **end**

A.5.3 List Operations

A.5.3.1 List Operator Signatures

List Operator Signatures

value
hd: $A^\omega \xrightarrow{\sim} A$
tl: $A^\omega \xrightarrow{\sim} A^\omega$
len: $A^\omega \xrightarrow{\sim}$ **Nat**
inds: $A^\omega \rightarrow$ **Nat-infset**
elems: $A^\omega \rightarrow$ **A-infset**
.(.): $A^\omega \times$ **Nat** $\xrightarrow{\sim} A$
$\frown$: $A^* \times A^\omega \rightarrow A^\omega$
=: $A^\omega \times A^\omega \rightarrow$ **Bool**
≠: $A^\omega \times A^\omega \rightarrow$ **Bool**

A.5.3.2 List Operation Examples

List Operation Examples

examples
hd⟨a1,a2,...,am⟩=a1
tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩
len⟨a1,a2,...,am⟩=m
inds⟨a1,a2,...,am⟩={1,2,...,m}
elems⟨a1,a2,...,am⟩={a1,a2,...,am}
⟨a1,a2,...,am⟩(i)=ai
⟨a,b,c⟩$\frown$⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩
⟨a,b,c⟩=⟨a,b,c⟩
⟨a,b,c⟩ ≠ ⟨a,b,d⟩

A.5.3.3 Informal List Operator Explication The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as

definitions, ≡, as [a kind of] identities.

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.
- $\frown$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- =: The equal operator expresses that the two operand lists are identical.
- ≠: The nonequal operator expresses that the two operand lists are not identical.

The operations can also be defined as follows:

A.5.3.4 List Operator Explications The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.

List Operator Explications

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat • 1 ≤ i ≤ len q },
      false → { i | i:Nat • i≠0 } end

  elems q ≡ { q(i) | i:Nat • i ∈ inds q }

  q(i) ≡
    if i=1
      then
        if q≠⟨⟩
          then let a:A,q′:Q • q=⟨a⟩^q′ in a end
          else chaos end
      else q(i−1) end

  fq ^ iq ≡
      ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
       | i:Nat • if len iq≠chaos then i ≤ len fq+len end ⟩
```

```
    pre is_finite_list(fq)

  iq′ = iq″ ≡
    inds iq′ = inds iq″ ∧ ∀ i:Nat • i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

  iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

A.5.4 Map Operations

A.5.4.1 Map Operator Signatures

Map Operator Signatures

```
value
[30] ·(·): M → A ~→ B
[31] dom: M → A-infset [domain of map]
[32] rng: M → B-infset [range of map]
[33] †: M × M → M [override extension]
[34] ∪: M × M → M [merge ∪]
[35] \: M × A-infset → M [restriction by]
[36] /: M × A-infset → M [restriction to]
[37] =,≠: M × M → Bool
[38] °: (A ~m~> B) × (B ~m~> C) → (A ~m~> C) [composition]
```


A.5.4.2 Map Operation Examples

Map Operation Examples

```
value
[30] m(a) = b
[31] dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}
[32] rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}
[33] [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]
[34] [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]
[35] [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]
[37] [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]
[38] [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]
```

A.5.4.3 Informal Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which maps to in a map.
- **rng**: Range/Image Set gives the set of values which are mapped to in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.

- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are not identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

A.5.4.4 Map Operator Explication The following is **not** a definition of RSL semantics. In RSL formulas we present an explication of RSL operators. Read, what appears as definitions, ≡, as [a kind of] identities.
The map operations can also be defined as follows:

Map Operator Explications

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
              a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```


A.6 λ-Calculus + Functions

The λ-Calculus is a foundation for the abstract specification language that RSL is

A.6.1 The λ-Calculus Syntax

λ-Calculus Syntax

```
type /∗ A BNF Syntax: ∗/
    ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | ( ⟨A⟩ )
    ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
```

⟨F⟩ ::= λ⟨V⟩ • ⟨L⟩
⟨A⟩ ::= (⟨L⟩⟨L⟩)
value /∗ Examples ∗/
⟨L⟩: e, f, a, ...
⟨V⟩: x, ...
⟨F⟩: λ x • e, ...
⟨A⟩: f a, (f a), f(a), (f)(a), ...

A.6.2 Free and Bound Variables

Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- ⟨V⟩: Variable x is free in x.
- ⟨F⟩: x is free in $\lambda y \bullet e$ if $x \neq y$ and x is free in e.
- ⟨A⟩: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

A.6.3 Substitution

1

Substitution

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx•P)) ≡ λ y•P;
- **subst**([N/x](λ y•P)) ≡ λy• **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy•P)) ≡ λz•**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

A.6.4 α-Renaming and β-Reduction

α and β Conversions

- α-renaming: λx•M

  If x, y are distinct variables then replacing x by y in λx•M results in λy•**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.
- β-reduction: (λx•M)(N)

All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx•M)(N) ≡ **subst**([N/x]M)

A.6.5 Function Signatures

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: B×C → A
```

A.6.6 Function Definitions

Functions can be defined explicitly:

Explicit Function Definitions

```
value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments ~→ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)
```


Or functions can be defined implicitly:

Implicit Function Definitions

```
value
  f: Arguments → Result
  f(args) as result
  post P1(args,result)

  g: Arguments ~→ Result
  g(args) as result
  pre P2(args)
  post P3(args,result)
```

1

A.7 Other Applicative Expressions

RSL offers the usual collection of applicative constructs that functional programming languages (Standard ML [11, 11] or F# [10]) offer

A.7.1 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

Let Expressions

let a = $\mathcal{E}_d$ **in** $\mathcal{E}_b$(a) **end**

is an "expanded" form of:

(λa.$\mathcal{E}_b$(a))($\mathcal{E}_d$)

A.7.2 Recursive let Expressions

Recursive **let** expressions are written as:

Recursive **let** Expressions

let f = λa:A • E(f) **in** B(f,a) **end**

is "the same" as:

let f = **YF** **in** B(f,a) **end**

where:

F $\equiv$ λg•λa•(E(g)) and YF = F(YF)

A.7.3 Predicative let Expressions

Predicative **let** expressions:

Predicative **let** Expressions

let a:A • $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

A.7.4 Pattern and "Wild Card" let Expressions

Patterns and wild cards can be used:

Patterns

let {a} $\cup$ s = set **in** ... **end**
let {a,_} $\cup$ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let $\langle$a$\rangle^\frown\ell$ = list **in** ... **end**
let $\langle$a,_,b$\rangle^\frown\ell$ = list **in** ... **end**

```
let [a↦b] ∪ m = map in ... end
let [a↦b,_] ∪ m = map in ... end
```

A.7.4.1 Conditionals

Various kinds of conditional expressions are offered by RSL:

Conditionals

```
if b_expr then c_expr else a_expr
end

if b_expr then c_expr end ≡ /∗ same as: ∗/
   if b_expr then c_expr else skip end

if b_expr_1 then c_expr_1
elsif b_expr_2 then c_expr_2
elsif b_expr_3 then c_expr_3
...
elsif b_expr_n then c_expr_n end

case expr of
   choice_pattern_1 → expr_1,
   choice_pattern_2 → expr_2,
   ...
   choice_pattern_n_or_wild_card → expr_n
end
```


A.7.5 Operator/Operand Expressions

Operator/Operand Expressions

```
⟨Expr⟩ ::=
          ⟨Prefix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
        | ⟨Expr⟩ ⟨Suffix_Op⟩
        | ...
⟨Prefix_Op⟩ ::=
        − | ∼ | ∪ | ∩ | card | len | inds | elems | hd | tl | dom | rng
⟨Infix_Op⟩ ::=
         = | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
        | ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | °
⟨Suffix_Op⟩ ::= !
```

A.8 Imperative Constructs

RSL offers the usual collection of imperative constructs that imperative programming languages (Java [9, 12] or Oberon (!) [13]) offer

A.8.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

Statements and State Change

```
value
  stmt: Unit → Unit
  stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

A.8.2 Variables and Assignment

Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

A.8.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

Statement Sequences and **skip**

2. **skip**
3. stm_1;stm_2;...;stm_n

A.8.4 Imperative Conditionals

Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

A.8.5 Iterative Conditionals

Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

A.8.6 Iterative Sequencing

Iterative Sequencing

8. **for** e **in** list_expr • P(b) **do** S(b) **end**

A.9 Process Constructs

RSL offers several of the constructs that CS [?] offers

A.9.1 Process Channels

As for channels we deviate from common RSL [7] in that we directly *declare* channels – and not via common RSL *objects* etc.

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

Process Channels

channel c:A
channel { k[i]:B • i:Idx }
channel { k[i,j,...,k]:B • i:Idx,j:Jdx,...,k:Kdx }

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values of the designated types (A and B).

A.9.2 Process Composition

Let P and Q stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let P() and Q stand for process expressions, then:

Process Composition

P ‖ Q Parallel composition
P⫿⫾Q Nondeterministic external choice (either/or)
P ⫾ Q Nondeterministic internal choice (either/or)
P ⫲ Q Interlock parallel composition

express the parallel (‖) of two processes, or the nondeterministic choice between two processes: either external (⫿⫾) or internal (⫾). The interlock (⫲) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

A.9.3 Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

Input/Output Events

```
c ?, k[i] ?      Input
c ! e, k[i] ! e  Output
```

expresses the willingness of a process to engage in an event that "reads" an input, respectively "writes" an output.

A.9.4 Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

Process Definitions

```
value
  P: Unit → in c out k[i]
  Unit
  Q: i:KIdx →  out c in k[i] Unit

  P() ≡ ... c ? ... k[i] ! e ...
  Q(i) ≡ ... k[i] ? ... c ! e ...
```


The process function definitions (i.e., their bodies) express possible events.

A.10 RSL Module Specifications

We shall not include coverage nor use of the RSL module concepts of *schemes*, *classes* and *objects*.

A.11 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemas, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

Simple RSL Specifications

```
type
  ...
variable
  ...
channel
  ...
value
  ...
axiom
  ...
```

A.12 RSL$^+$: Extended RSL

Section A.1 on page 19 covered standard RSL types. To them we now add two new types: Type names and RSL-Text.

We refer to Sect. **??** (the *An RSL Extension* box) Page **??** for a first introduction to extended RSL.

A.12.1 Type Names and Type Name Values

A.12.1.1 Type Names

- Let T be a type name.
- Then ηT is a type name value.
- And $\eta\mathbb{T}$ is the type of type names.

A.12.1.2 Type Name Operations

- η can be considered an operator.
  - It (prefix) applies, then, to type (T) identifiers and yields the name of that type.
  - Two type names, nT$_i$, nT$_j$, can be compared for equality: nT$_i$ = nT$_j$ iff $i = j$.
- It, vice-versa, suffix applies to type name (nT) identifiers and yields the name, T, of that type: nTη = T.

A.12.2 RSL-Text

A.12.2.1 The RSL-Text Type and Values

- RSL-Text is the type name for ordinary, non-extended RSL texts.

We shall not here give a syntax for ordinary, non-extended RSL texts – but refer to [7].

A.12.2.2 RSL-Text Operations

- RSL-Texts can be compared and concatenated:
  - rsl-text$_a$=rsl-text$_b$
  - rsl-text$_a$⌢rsl-text$_b$

The ⌢ operator thus also applies, besides, lists (tuples), to RSL texts – treating RSL texts as (if they were) lists of characters.

A.13 Distributive Clauses

We clarify:

A.13.1 Over Simple Values

⊕ { a | a:A • a ∈ {a_1,a_2,...,a_n} } =
  if n>0 **then** a_1⊕a_1⊕...⊕a_n **else**
    case ⊕ **of**
      + → 0, − → 0, ∗ → 1, / → **chaos**, ∪ → {}, ∩ → {}, ...
    end end

(f_1,f_2,...,f_n)(a) ≡ **if** n>0 **then** (f_1(a),f_2(a),...,f_n(a)) **else chaos end**

A.13.2 Over Processes

‖ { p(i) | i:I • i ∈ {i_1,i_2,...,i_n} } ≡ **if** n>0 **then** p(i_1)‖p(i_2)‖...‖p(i_n) **else** () **end**
⫿ { p(i) | i:I • i ∈ {i_1,i_2,...,i_n} } ≡ **if** n>0 **then** p(i_1)⫿p(i_2)⫿...⫿p(i_n) **else** () **end**
⫼{ p(i) | i:I • i ∈ {i_1,i_2,...,i_n} } ≡ **if** n>0 **then** p(i_1)⫼p(i_2)⫼...⫼p(i_n) **else** () **end**

A.14 Indexes