1

Introduction

- The **prerequisite** for studying this chapter is that you have read and understood Vol. 1 of this series of textbooks on software engineering.
- The **aims** are to motivate why the present volume is written, to motivate why you should read it by outlining what it contains and how it delivers its material, to explain the notion of formal methods "lite", and to briefly recall the main specification language of these volumes, RSL.
- The **objective** is to set you firmly on the way to study this volume.
- The **treatment** is discursive, informal and systematic.

1.1 Introduction

Volume 2 continues where Vol. 1 left off. Having laid the foundations for discrete mathematics, Vol. 1, Chaps. 2–9, abstraction and modelling, Vol. 1, Chaps. 10–18, and specification programming, Vol. 1, Chaps. 19–21, which we consider the minimum for the pursuit of professional software engineering, we need now to expand, considerably, the scope of areas to which we can apply our abstraction, modelling and specification skills.

This chapter has two main sections: First we outline the justification for and contents of this volume as well as how the material in this volume is presented. Then we give an ever-so-short primer on RSL: the syntactic constructs, very briefly their "meaning" and their pragmatics, that is, which "main" uses with respect to abstraction and modelling they serve to fulfill. The primer can, of course, be skipped.

1.1.1 Why This Volume?

It is one thing to learn and be reasonably fluent in *abstraction and modelling* as covered in Vol. 1 of this series. It is another thing to really master the principles, techniques and tools. With the present volume our goal is to educate

you to the level of a professional software engineer in: (i) specifying complicated computing systems and languages, (ii) being aware of major semiotics principles (pragmatics, semantics and syntax), (iii) being well acquainted to means of handling concurrency, i.e., parallel systems, and real-time, and (iv) formally conceiving reasonably sophisticated systems and languages.

1.1.2 Why Master These Principles, Techniques and Tools?

Why master these principles, techniques and tools? Because it is necessary. Because, to be a professional in one's chosen field of expertise, one must know also the formal techniques — just as engineers of other disciplines also know their mathematics. Just as fluid mechanics engineers handle, with ease, their *Navier–Stokes Equations* [83, 496], so software engineers must handle *denotational* and *computational semantics*. Just as radio communications engineers handle, with ease, *Maxwell Equations* [245, 502], so software engineers must handle *Petri nets* [238, 400, 419–421], *message sequence charts* [227–229], *live sequence charts* [89, 195, 268], *statecharts* [174, 175, 185, 193, 197], the *duration calculus* [557, 559], *temporal logics* [105, 320, 321, 372, 403], etc. We will cover this and much more in this volume.

The above explanation of the "why" is an explanation that is merely a claim. It relies on "Proof by authority"! Well, here is the longer, more rational argument: Before we can *design software*, we must understand its *requirements*. Before we can *construct requirements*, we must understand the application *domain*, the area of, say human, activity for which software is desired. To express domain understanding, requirements and software designs we must use language. To claim any understanding of these three areas the language used must be precise, and must be used such as to avoid ambiguities, and must allow for formal reasoning, i.e., proofs. This entails formal languages. To cope with the span from domains, via requirements, to designs the languages must provide for abstraction, and refinement: from abstract to concrete expressibility. The principles, techniques and tools of these volumes provide a state-of-the-art (and perhaps beyond) set of such methods.

The complexities of the computing systems that will be developed in the future are such that we cannot expect to succeed in developing such computing systems without using formal techniques and tools, such as covered and propagated in these volumes.

1.1.3 What Does This Volume "Contain"?

Volume 1 covered basic abstraction and modelling principles, techniques and tools. The major tool was that of the `RAISE` Specification `Language` (`RSL`). The major new, additional tools of this volume will be those of the *Petri nets:* condition event nets, the place transition nets, and the coloured Petri nets [238, 400, 419–421]; the *sequence charts* (SCs): the message SCs (MSCs) [227–229] and the live SCs (LSCs) [89, 195, 268]; the *statecharts* [174, 175,

185, 193, 197]; the *interval temporal logic* (ITL) and the *duration calculus* (DC) [557, 559].

The major principles and techniques of abstraction and modelling covered earlier were: *property-* (sorts, observers, generators, axioms) and/versus *model-oriented abstraction* in general, and the model-oriented techniques of *set, Cartesian, list, map* and *function,* including *type abstractions;* and *functional, imperative* and *concurrent* (parallel) *specification programming techniques* in particular.

The new, additional principles and techniques of abstraction and modelling in this volume fall along five axes:

1. An advanced abstraction and modelling axis, covering hierarchical and compositional modelling and models, denotational and computational semantics, configurations: contexts and state, and time and space concepts. This axis further extends the techniques of Vol. 1. The time concepts will be further treated along axis (4).
2. A semiotics axis, covering pragmatics, semantics and syntax. This axis treats, along more systematic lines, what was shown more or less indirectly in Vol. 1 and previous chapters of Vol. 2 (notably Chap. 3). Axis (5) will complete our treatment of linguistics.
3. A structuring axis, briefly covering `RSL`'s scheme, class and object concepts, as well as `UML`'s class diagram concepts. This "short" axis, for the first time in these volumes, brings other notational tools into our evolving toolbox. This "extension" or enlargement of the variety of notational tools brings these volumes close to covering fundamental ideas of `UML`. The next axis, (4), completes this expansion.
4. A concurrency axis, covering *qualitative* aspects of timing: the *Petri nets* [238, 400, 419–421], the *sequence charts*, SCs, message SCs (MSCs [227–229]) and live SCs (LSCs [89,195,268]), the *statecharts* [174,175,185,193, 197], and *quantitative* aspects of timing in terms of the *interval temporal logic* (ITL) [105,320,321,372,403], and the *duration calculus* (DC) [557, 559]. These specification concepts, available in some form in `UML`, will complete these volumes' treatment of, as we call it, *"UML-ising" Formal Techniques.*
5. A language development axis, covering crucial steps of the development of concrete interpreters and compilers for functional (i.e., applicative), imperative (i.e., "classical"), modular, and parallel programming languages. This axis completes our treatment of programming language linguistics matters. The chapters in axis (5) will cover important technical concepts of run-time structures for interpreted and compiled programs, compiling algorithms, and attribute grammars.

1.1.4 How Does This Volume "Deliver"?

The previous section outlined, in a sense, a *didactics* of one main aspect of software engineering.

So this *didactic* view of software engineering as a field of activity whose individual "tasks" can be "relegated" to one, or some simple combination, of the topics within one or, say, two axes, as listed above offers one way in which this volume "delivers". That is, the reader will be presented with these topics, more or less in isolation, one-by-one, but the practicing software engineer (and the reader as chapter exercise solver) is expected to merge principles and techniques of previous topics and tools when solving problems.

Another way in which this volume delivers is in the manner in which each individual (axis) topic is presented. Each topic is presented by means of many examples. Their "story" is narrated and the problem is given a formal specification. Where needed, as for the qualitative and quantitative aspects of concurrency,[1] a description is given of (i) their notational apparatus, (ii) the pragmatics behind them, (iii) their syntax and (iv) their informal semantics. Method principles and techniques are then enunciated. A heavy emphasis is placed on examples. References are made to more theoretical treatments of, in particular, the concurrency topics.

A third way in which this volume delivers is by presenting a "near-full" spectrum of principles, techniques and tools, as witnessed, for example, by the combination of using the `RSL` tool with those of `UML`'s class diagrams, the Petri Nets, the (Message and Live) Sequence Charts, the Statecharts, the Interval Temporal Logic and the Duration Calculus.

This can also be seen in the span of abstraction topics: hierarchy and composition, denotation and computation, configurations (including contexts and states), temporality (in various guises) and spatiality, and both qualitative and quantitative aspects of concurrency. Volume 3 covers further abstraction principles and techniques. Finally this is also witnessed by the span of application topics: real-time, embedded and safety critical systems, infrastructure components (railways, production, banking, etc.), and programming languages: functional, imperative, modular, and parallel. Volume 3 covers further application topics.

1.2 Formal Techniques "Lite"

Although we shall broach the subject on several occasions throughout this volume, when we cover formal techniques we shall exclusively cover formal specification, not formal proofs of properties of specifications.

That may surprise the reader. After all, a major justification of formal techniques, i.e., formal specifications, is that they allow formal verification. So why do we not cover formal verification? First, we use, and propagate

[1] The qualitative aspects of concurrency are expressible when using the Petri Nets, the Message and Live Sequence Charts and the Statecharts. The quantitative aspects of concurrency are expressible when using the Interval Temporal Logic and the Duration Calculus.

the use of, formal techniques in the "lite"[2] manner. That is, we take formal specification rather seriously. And hence we focus on principles and techniques for constructing effective specifications, i.e., pleasing, elegant, expressive and revealing specifications. We find (and have over more than 30 years found) that systems developed in this manner come very, very close to being perfect!

Second, we find that principles and techniques for theorem proving or proof assistance or model checking, even today (2005) are very much "bound" to the specific notational system (i.e., specification language), and to its proof system of rules and tools. And we also find that there is much less a common consensus on whether proofs should be done in one way or in another way.

For a good introduction to a number of leading approaches to software verification we refer to the following papers:

1. J. U. Skakkebæk, A. P. Ravn, H. Rischel, and Zhou Chaochen. *Specification of embedded, real-time systems.* Proceedings of 1992 Euromicro Workshop on Real-Time Systems, pages 116–121. IEEE Computer Society Press, 1992.
2. Zhou Chaochen, M. R. Hansen, A. P. Ravn, and H. Rischel. *Duration specifications for shared processors.* Proceedings Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, Nijmegen 6-10 Jan. 1992, LNCS, 1992.
3. A. P. Ravn, H. Rischel, and K. M. Hansen. *Specifying and verifying requirements of real-time systems.* IEEE Trans. Software Engineering, 19:41–55, 1992.
4. C. W. George. *A theory of distributing train rescheduling.* In FME'96: Industrial Benefits and Advances in Formal Methods, proceedings, LNCS 1051,
5. C. W. George. *Proving safety of authentication protocols: a minimal approach,* in International Conference on Software: Theory and Practice (ICS 2000), 2000.
6. A. Haxthausen and X. Yong. *Linking DC together with TRSL.* Proceedings of 2nd International Conference on Integrated Formal Methods (IFM 2000), Schloss Dagstuhl, Germany, November 2000, number 1945 in Lecture Notes in Computer Science, pages 25–44. Springer-Verlag, 2000.
7. A. Haxthausen and J. Peleska, *Formal development and verification of a distributed railway control system,* IEEE Transaction on Software Engineering, 26(8), 687–701, 2000.
8. M. P. Lindegaard, P. Viuf and A. Haxthausen, *Modelling railway interlocking systems,* Eds.: E. Schnieder and U. Becker, Proceedings of the 9th IFAC Symposium on Control in Transportation Systems 2000, June 13–15, 2000, Braunschweig, Germany, 211–217, 2000.
9. A. E. Haxthausen and J. Peleska, *A domain specific language for railway control systems,* Sixth Biennial World Conference on Integrated Design

[2] "Lite" is an "Americanism", and, as many such, is a nice one that indicates that we take certain things seriously, but not necessarily all that "seriously".

and Process Technology, (IDPT 2002), Pasadena, California, Society for Design and Process Science, P. O. Box 1299, Grand View, Texas 76050-1299, USA, June 23-28, 2002.
10. A. Haxthausen and T. Gjaldbæk, *Modelling and verification of interlocking systems for railway lines,* 10th IFAC Symposium on Control in Transportation Systems, Tokyo, Japan, August 4–6, 2003.

One runs a danger by adhering too much to the above "liteness" principle (perhaps it is one of lazy convenience?). That danger is as follows: Formulating which property is to be verified, of a specification, or, respectively, which correctness criterion is to be verified "between" a pair of specifications, and carrying through the proofs often helps us focus on slightly different abstractions than if we did not consider lemmas, propositions and theorems to be verified, or verification itself. And sometimes these proof-oriented abstractions turn out to be very beautiful, very much "to the point" and also "just", specification-wise!

So what do we do? Well, we cannot cover everything, therefore we must choose. These volumes have made the above choice. So, instead, we either refer the reader to other seminal textbooks on correctness proving [20, 97, 151, 205, 206, 363, 429], even though these other textbooks pursue altogether different specification approaches, or to two books that pursue lines of correctness development very much along the lines, otherwise, of this book: Cliff Jones' book [247], which uses `VDM`, and the `RAISE` Method book [131].

1.3 An `RSL` Primer

This is an ultrashort introduction to the `RAISE` Specification Language, `RSL`.

1.3.1 Types

We refer the reader to Vol. 1, Chaps. 5 and 18.

The reader is kindly asked to study first the decomposition of this section into its subparts and sub-subparts.

Type Expressions

`RSL` has a number of *built-in* types. There are the Booleans, integers, natural numbers, reals, characters, and texts. 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:

Basic Types

type
[1] **Bool**

[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

Type Expressions

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

The following are generic type expressions:

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, ...
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″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...
7. The set type of finite set values.
8. The set type of infinite set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite list values.
11. The list type of infinite list values.
12. The map type of finite map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,

- or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\overrightarrow{m}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.

16. The postulated disjoint union of types A, B, ..., and C.
17. The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

Type Definitions

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

Some schematic type definitions are:

Variety of Type Definitions

```
[1] Type_name = Type_expr /* without |s or subtypes */
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] 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)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ • 𝒫(v) |}
```

where a form of [2–3] is provided by combining the types:

Record Types

```
Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

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 • $\mathcal{P}$(b) |}

Sorts — Abstract Types

Types can be (abstract) sorts in which case their structure is not specified:

Sorts

type
A, B, ..., C

1.3.2 The RSL Predicate Calculus

We refer the reader to Vol. 1, Chap. 9.

Propositional Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values. 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 are read: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

Simple Predicate Expressions

Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values, let x, y, ..., z (or term expressions) designate non-Boolean values and let i, j, ..., k designate number values, then:

Simple Predicate Expressions

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i>j, ...

are simple predicate expressions.

Quantified Expressions

Let X, Y, ..., C be type names or type expressions, and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free. Then:

Quantified Expressions

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions. They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds.

1.3.3 Concrete RSL Types

We refer the reader to Vol. 1, Chaps. 13–16.

Set Enumerations

We refer the reader to Vol. 1, Chap. 13, Sect. 13.2.

Let the below a's denote values of type A, then the below designate simple set enumerations:

Set Enumerations

{{}, {a}, {a_1,a_2,...,a_m}, ...} ∈ A-**set**
{{}, {a}, {a_1,a_2,...,a_m}, ..., {a_1,a_2,...}} ∈ A-**infset**

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 highly 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) }
```

Cartesian Enumerations

We refer the reader to Vol. 1, Chap. 14, Sect. 14.2.

Let e range over values of Cartesian types involving A, B, ..., C (allowing indexing for solving ambiguity), then the below expressions are simple Cartesian enumerations:

Cartesian Enumerations

```
type
   A, B, ..., C
   A × B × ... × C
value
   ... (e1,e2,...,en) ...
```

List Enumerations

We refer the reader to Vol. 1, Chap. 15, Sect. 15.2.

Let a range over values of type A (allowing indexing for solving ambiguity), then the below expressions are simple list enumerations:

List Enumerations

```
{⟨⟩, ⟨a⟩, ..., ⟨a1,a2,...,am⟩, ...} ∈ A*
{⟨⟩, ⟨a⟩, ..., ⟨a1,a2,...,am⟩, ..., ⟨a1,a2,...,am,... ⟩, ...} ∈ Aω

⟨ ei .. ej ⟩
```

The last line above assumes e_i and e_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.

The last line below expresses list comprehension.

List Comprehension

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

Map Enumerations

We refer the reader to Vol. 1, Chap. 16, Sect. 16.2.

Let a and b range over values of type A and B, respectively (allowing indexing for solving ambiguity), then the below expressions are simple map enumerations:

Map Enumerations

```
type
   A, B
   M = A ~m~→ B
value
   a,a1,a2,...,a3:A, b,b1,b2,...,b3:B

   [], [a↦b], ..., [a1↦b1,a2↦b2,...,a3↦b3] ∀ ∈ M
```

The last line below expresses map comprehension:

Map Comprehension

```
type
   A, B, C, D
   M = A ~m~→ B
   F = A ~→ C
   G = B ~→ D
   P = A → Bool
value
   comprehend: M×F×G×P → (C ~m~→ D)
```

comprehend(m,$\mathcal{F}$,$\mathcal{G}$,$\mathcal{P}$) ≡
[$\mathcal{F}$(a) ↦ $\mathcal{G}$(m(a)) | a:A • a ∈ **dom** m ∧ $\mathcal{P}$(a)]

Set Operations

We refer the reader to Vol. 1, Chap. 13, Sect. 13.2.

Set Operations

value
∈: A × A-**infset** → **Bool**
∉: A × A-**infset** → **Bool**
∪: A-**infset** × A-**infset** → A-**infset**
∪: (A-**infset**)-**infset** → A-**infset**
∩: A-**infset** × A-**infset** → A-**infset**
∩: (A-**infset**)-**infset** → A-**infset**
\: A-**infset** × A-**infset** → A-**infset**
⊂: A-**infset** × A-**infset** → **Bool**
⊆: A-**infset** × A-**infset** → **Bool**
=: A-**infset** × A-**infset** → **Bool**
≠: A-**infset** × A-**infset** → **Bool**
card: A-**infset** $\xrightarrow{\sim}$ **Nat**

Set 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

- $\in$: The membership operator expresses that an element is a member of a set.
- $\notin$: The nonmembership operator expresses that an element is not a member of a set.
- $\cup$: 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.
- $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
- $\setminus$: 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.
- $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
- $\subset$: 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.
- $=$: The equal operator expresses that the two operand sets are identical.
- $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.
- **card**: The cardinality operator gives the number of elements in a finite set.

The operations can be defined as follows ($\equiv$ is the definition symbol):

Set Operation Definitions

```
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 ∗/
```

Cartesian Operations

We refer the reader to Vol. 1, Chap. 14, Sect. 14.2.

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**

List Operations

We refer the reader to Vol. 1, Chap. 15, Sect. 15.2.

List Operations

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

List 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

$\langle a,b,c\rangle^\frown\langle a,b,d\rangle = \langle a,b,c,a,b,d\rangle$
$\langle a,b,c\rangle = \langle a,b,c\rangle$
$\langle a,b,c\rangle \neq \langle a,b,d\rangle$

- **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 gives 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.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operation Definitions

```
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″)
```

Map Operations

We refer the reader to Vol. 1, Chap. 16, Sect. 16.2.

Map Operations

```
value
  m(a): M → A ~→ B, m(a) = b

  dom: M → A-infset [domain of map]
    dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

  rng: M → B-infset [range of map]
    rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

  †: M × M → M [override extension]
    [a↦b,a′↦b′,a″↦b″] † [a′↦b‴,a‴↦b′] = [a↦b,a′↦b‴,a‴↦b′]

  ∪: M × M → M [merge ∪]
    [a↦b,a′↦b′,a″↦b″] ∪ [a‴↦b‴] = [a↦b,a′↦b′,a″↦b″,a‴↦b‴]


  \: M × A-infset → M [restriction by]
    [a↦b,a′↦b′,a″↦b″]\{a} = [a′↦b′,a″↦b″]

  /: M × A-infset → M [restriction to]
    [a↦b,a′↦b′,a″↦b″]/{a′,a″} = [a′↦b′,a″↦b″]

  =,≠: M × M → Bool
```

```
°: (A ~m~> B) × (B ~m~> C) → (A ~m~> C) [composition]
    [a↦b,a′↦b′] ° [b↦c,b′↦c′,b″↦c″] = [a↦c,a′↦c′]
```

- $m(a)$: 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.

The map operations can also be defined as follows:

Map Operation Redefinitions

```
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
```

1.3.4 λ-Calculus+Functions

We refer the reader to Vol. 1, Chaps. 6, 7 and 11.

The λ-Calculus Syntax

We refer the reader to Vol. 1, Chap. 7, Sect. 7.2.

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), ...
```

Sections 8.4–8.5 cover the notion of BNF grammars in detail.

Free and Bound Variables

We refer the reader to Vol. 1, Chap. 7, Sect. 7.3.

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).

Substitution

We refer the reader to Vol. 1, Chap. 7, Sect. 7.4. In RSL, the following rules for substitution apply:

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)).

α-Renaming and β-Reduction

We refer the reader to Vol. 1, Chap. 7, Sect. 7.4.

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)

Function Signatures

We refer the reader to Vol. 1, Chaps. 6 and 11. 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

Function Definitions

We refer the reader to Vol. 1, Chap. 11, Sects. 2–6. Functions can be defined explicitly:

Explicit Function Definitions

value
f: A × B × C → D
f(a,b,c) ≡ Value_Expr

g: B-**infset** × (D $\overrightarrow{m}$ C-**set**) $\xrightarrow{\sim}$ A*
g(bs,dm) ≡ Value_Expr
pre $\mathcal{P}$(bs,dm)

comment: a, b, c, bs and dm are parameters of appropriate types

or implicitly:

Implicit Function Definitions

value
f: A × B × C → D
f(a,b,c) **as** d
post $\mathcal{P}_1$(a,b,c,d)

g: B-**infset** × (D $\overrightarrow{m}$ C-**set**) $\xrightarrow{\sim}$ A*
g(bs,dm) **as** al
pre $\mathcal{P}_2$(bs,dm)
post $\mathcal{P}_3$(bs,dm,al)

comment: a, b, c, bs and dm are parameters of appropriate types

The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

1.3.5 Other Applicative Expressions

Let Expressions

We refer the reader to Vol. 1, Chap. 19, Sect. 19.2.

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))($\mathcal{E}_d$)

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 = **Y**F **in** B(f,a) **end**

where:

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

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).

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 ⟨a⟩^ℓ = list in ... end
let ⟨a,_,b⟩^ℓ = list in ... end

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

Conditionals

We refer the reader to Vol. 1, Chap. 19, Sect. 19.5.

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
```

Operator/Operand Expressions

We refer the reader to Vol. 1, Chap. 19.

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⟩ ::= !

1.3.6 Imperative Constructs

We refer the reader to Vol. 1, Chap. 20.

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.

Variables and Assignment

We refer the reader to Vol. 1, Chap. 20, Sects. 20.2.1–20.2.2.

Variables and Assignment

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

Statement Sequences and skip

We refer the reader to Vol. 1, Chap. 20, Sects. 20.2.5 and 20.2.4.

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

Imperative Conditionals

We refer the reader to Vol. 1, Chap. 20, Sects. 20.2.6 and 20.2.8.

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**

Iterative Conditionals

We refer the reader to Vol. 1, Chap. 20, Sect. 20.2.7.

Iterative Conditionals

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

Iterative Sequencing

We refer the reader to Vol. 1, Chap. 20, Sect. 20.2.9.

Iterative Sequencing

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

1.3.7 Process Constructs

We refer the reader to Vol. 1, Chap. 21, Sect. 21.4.

Process Channels

We refer the reader to Vol. 1, Chap. 21, Sect. 21.4.1.

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:KIdx }

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

Process Composition

We refer the reader to Vol. 1, Chap. 21, Sects. 21.4.4–21.4.7.

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(i) stand for process expressions, then:

Process Composition

```
P() || Q(i)    Parallel composition
P() [] Q(i)    Nondeterministic external choice (either/or)
P() |~| Q(i)   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.

Input/Output Events

We refer the reader to Vol. 1, Chap. 21, Sect. 21.4.2.

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, and respectively "writes" an output.

Process Definitions

We refer the reader to Vol. 1, Chap. 21, Sect. 21.4.3.

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.

1.3.8 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, 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
...

1.4 Bibliographical Notes

The main references to RSL — other than Vol. 1 of this series — are [130,131].