MASTER OF SCIENCE THESIS

Integration of "UML-ised" Formal Techniques and Tools with RSL and the RSL toolset

Steffen Andersen
s973987

Steffen Holmslykke
s991311

No. 2005-20
LYNGBY
March 2005

IMM

DTU

Preface

This master thesis was carried out at the Department of *Informatics and Mathematical Modelling* (IMM) at the *Technical University of Denmark* (DTU). It was supervised by Professor Dines Bjørner and co-supervised by Associate Professor Anne Haxthausen. The master thesis was written in the period August 2004 to March 2005:

- at *United Nations University - International Institute for Software Technology* (UNU-IIST) from August 2004 to November 2004,
- at *National University of Singapore - School of Computing* (NUS-SoC) from December 2004 to January 2005 and
- at DTU from February 2005 to March 2005.

We would like to thank our supervisor Dines Bjørner for his invaluable guidance and for a great stay in Singapore. Likewise we would like to thank Chris George for helping us and discussing our work during and after our stay at UNU-IIST. Thanks also to our co-supervisor Anne Haxthausen for helping us in the final phase of our thesis. Finally we would like to thank staff and students alike at UNU-IIST and NUS-SoC for constructive discussions and an incredible experience.

Lyngby, March 21st, 2005

Steffen Andersen, s973987

Steffen Holmslykke, s991311

Abstract

Formal methods and graphical notations are tools in software engineering and much attention is given to improve the integration of the two. In particular the Unified Modelling Language (UML) seems to have been adopted as a de facto standard in industry as a graphical notation and has attracted much research interest.

Many have focused on the formalisation of UML Class Diagrams with various success. In this thesis we go in the opposite direction. We "UML'ise" the formal specification language RSL by presenting a new diagram called Scheme Diagram which displays the structure of a RSL model. The diagram is visually inspired by the UML Class Diagram but is semantically directly mapped to RSL. A plug-in has been developed for the Eclipse Editor, which enables the user to draw diagrams and translate them into RSL.

Secondly we look at the rather new Live Sequence Charts (LSCs), which are a successor to Message Sequence Charts (MSCs) and hence Sequence Diagrams in UML. We formalise a subset in RSL and examine the usefulness of LSC in a RSL context. An equivalent of LSCs in RSL is presented which allows refinement of the initial model.

Keywords: RAISE, RSL, Graphical Notations, UML, Live Sequence Charts, Eclipse

Resumé

Formelle methoder og grafiske notationer er værktøjer i software engineering og meget opmærksomhed er blevet givet til integrationen mellem de to. Specielt Unified Modelling Language (UML) ser ud til at være blevet godtaget som en de facto standard i industrien som en grafisk notation. Forskningsmæssigt har det også fået meget opmærksomhed.

Mange har fokuseret på formaliseringen af UML's klasse diagram med varierende success. I denne afhandling går vi i den modsatte retning. Vi "UML'iserer" det formelle specificationssprog RSL ved at præsentere et nyt diagram som vi kalder Scheme Diagram som viser strukturen i en RSL model. Diagrammet er visuelt inspireret af UML's klasse diagram men er semantisk set direkte relateret til RSL. Et plug-in er blevet udviklet til Eclipse Editoren som sætter brugeren i stand til at tegne diagrammer og oversætte dem til RSL.

Desuden kigger vi på de forholdsvis nye Live Sequence Charts (LSCs) som er en efterfølger til Message Sequence Charts (MSCs) og dermed Sequence Diagrammer fra UML. Vi formaliserer en delmængde af RSL og undersøger brugbarheden af LSCs i RSL sammenhæng. RSL-versionen af LSC bliver præsenteret. Denne tillader forfining af den initielle model.

Nøgleord: RAISE, RSL, Grafiske notationer, UML, Live Sequence Charts, Eclipse

摘要

形式化方法和图形标记都是软件工程中的工具，如何提高两者的结合度吸引了许多的注意力。 特别地，UML（统一建模语言）似乎已经被工业界作为图形标记的一个实际的标准，并且吸引了许多研究人员的兴趣。

大多数的研究集中于形式化 UML 的类图并取得了不同程度的成功。在这份论文中，我们进行的是反方向的研究。 我们“UML 化”（UML'ise）形式化规格说明语言 RSL (RAISE Specification Language)，并提出了一种新的图示 – Scheme Diagram， 用以展示一个 RSL 模型的结构。 Scheme Diagram 表面看上是根源于 UML 的类图，但是在语义上它是直接与 RSL 相对应的。我们为 Eclipse 编辑器开发了一个插件， 它使用户能够画 Scheme Diagram 并且把它们翻译为 RSL。

其次，我们研究了 Live Sequence Charts (LSC)。它相当的新，是消息序列图（Message Sequence Charts）的后继者，因此也是 UML 中的顺序图（Sequence Diagrams）的后继者。 我们形式化 RSL 的一个子集并检查 LSC 在 RSL 环境中的可用性。我们提出了 LSC 在 RSL 中的一个对等物， 它将允许初始模型的改进与提炼。

Presented above is our abstract in simplified chinese. The words written in english have no corresponding representation in the chinese language. Thank you to Liang Hui from NUS-SoC for helping us with the translation.

Contents

I Prelude **1**

1 Appetisers **3**
1.1 Scheme Diagram: Railway nets . . . 3
1.2 Live Sequence Chart: Light . . . 4

2 Introduction **5**
2.1 Motivation . . . 5
2.2 Contents . . . 6
2.3 Previous work . . . 6
2.4 Thesis structure . . . 6
2.5 The big picture . . . 7
2.6 Conventions . . . 7
2.7 Assumptions . . . 8

II Abstract models **9**

3 Introduction **11**

4 RSL syntax **13**
4.1 Types: RSL_α . . . 13
4.2 Print: $\text{RSL}_\alpha \rightarrow \text{RSL}_\gamma$ . . . 14

5 Scheme diagrams **15**
5.1 Introduction . . . 16
5.2 Previous work . . . 22
5.3 Summary of preliminary thesis . . . 24
5.4 Final Scheme Diagram . . . 25
5.5 Narrative of the Scheme Diagram syntax . . . 32
5.6 Examples . . . 46
5.7 Translation: $\text{SD}_\delta \rightarrow \text{RSL}_\alpha$ . . . 50
5.8 Future work . . . 50
5.9 Conclusion . . . 51

6 Live Sequence Charts **53**
6.1 Before we start . . . 54
6.2 Structured narrative of LSC . . . 56
6.3 Previous work . . . 60
6.4 The LSC subset chosen: RSC . . . 66
6.5 Formal description of RSC . . . 70
6.6 Example: RSC RSL specification . . . 78
6.7 Translation: $\text{RSC}_\delta \rightarrow \text{RSL}_\alpha$ . . . 82
6.8 Example: Applicative RSC . . . 89
6.9 Future work . . . 92
6.10 Conclusion . . . 92

III Concrete implementation **93**

7 Introduction **95**

8 Language and library **97**
8.1 Requirements . . . 97
8.2 Candidates . . . 97
8.3 Selection and rationale . . . 99

9 System description **101**
9.1 Overview . . . 101
9.2 Eclipse plug-in . . . 102
9.3 Eclipse Scheme Diagram Editor . . . 105
9.4 Imperative RSL model specification of Scheme Diagram . . . 107
9.5 Gluing the Eclipse plug-in and the RSL model together . . . 108
9.6 Test . . . 109

IV Postlude **111**

10 Conclusion **113**

Bibliography **114**

V Appendix **119**

A Glossary **121**
A.1 Scheme Diagram . . . 121
A.2 LSC . . . 122

B Description of RSL types in RSL **125**
B.1 rslsyntax.rsl . . . 125
B.2 rslprint.rsl . . . 143

C RSL specifications for the Scheme Diagram **179**
C.1 Scheme Diagram syntax . . . 179
C.2 Translation of Scheme Diagram to RSL. . . . 209
C.3 Imperative Scheme Diagram . . . 225
C.4 Test . . . 245

D RSL specifications for the RSC **251**
D.1 RSC syntax . . . 251
D.2 RSC semantics for one chart . . . 271
D.3 RSC collections . . . 287
D.4 Test . . . 294
D.5 CSP and LSC . . . 329
D.6 Applicative RSC . . . 346

E Contents of companion CD **363**

F Use of ESDE CASE Tool **365**
F.1 Installation . . . 365
F.2 User manual . . . 366

G Conferences **371**
G.1 ICTAC 2004, Guiyang . . . 371
G.2 SEFM 2004, Beijing . . . 373

H Seminars **375**
H.1 Seminars at UNU-IIST 375

I A tale of two cities **379**
I.1 Macau 379
I.2 Singapore 380

Part I

Prelude

Chapter 1

Appetisers

The following two sections present examples of the two diagrams that are discussed and used in this thesis. Being appetisers they will be superficial but the diagrams are elaborated later. The first example is about railway nets and demonstrates the Scheme Diagram. The second example shows the process of turning on a light illustrated by a Live Sequence Chart.

1.1 Scheme Diagram: Railway nets

In [31] a specification is presented which describes simple railway nets. The following example is a part of the specification covering sequences of linear rail units. First an informal description:

1. A rail *unit* is either a linear, switch, simple crossover or switchable crossover.
2. A rail *unit* has one or more *connectors*.
3. A *line* is a linear sequence of one or more linear rail *units*.
4. A *track* is a linear sequence of one or more linear rail *units*.
5. For every *connector* there are at most two rail *units* which have that *connector* in common.

Each of the artifacts/concepts written in italics in the informal description is placed in its own scheme in the specification. Both line and track is a sequence of linear rail units. An additional scheme named *sequence* is introduced which describe these similarities. Figure 1.1 show the Scheme Diagram of the specification which follows on the next page. Both the *Lines* and *Tracks* scheme extend the *Sequence* which is shown by the solid line with a hollow equal-sided triangle. The *Units* and *Sequence* schemes are parameterised. The formal parameters are shown as solid lines with a hollow diamond at the end.

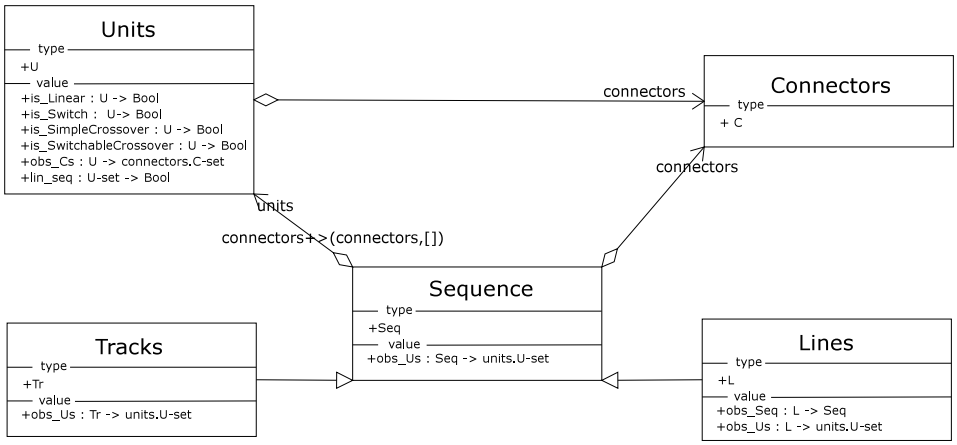

Figure 1.1: Scheme Diagram of a part of the railway nets specification.

```
scheme Units(connectors : Connectors) =
    class
        type U
        value
            is_Linear : U → Bool,
            is_Switch : U → Bool,
            is_SimpleCrossover : U → Bool,
            is_SwitchableCrossover : U → Bool,
            obs_Cs : U → connectors.C-set,
            lin_seq : U-set → Bool
    end
```

```
scheme Connectors = class type C end
```

```
scheme Sequence(
     connectors : Connectors,
     units : Units(connectors)) =
    class
        type Seq
        value obs_Us : Seq → units.U-set
    end
```

```
scheme Tracks(
     connectors : Connectors,
     units : Units(connectors)) =
    extend Sequence(connectors, units) with
    class
        type Tr
        value obs_Us : Tr → units.U-set
    end
```

```
scheme Lines(
     connectors : Connectors,
     units : Units(connectors)) =
    extend Sequence(connectors, units) with
    class
        type L
        value
            obs_Seq : L → Seq,
            obs_Us : L → units.U-set
    end
```

1.2 Live Sequence Chart: Light

The diagram 1.2 is an example of a Live Sequence Chart. It is a simple description of how a light may be switched on. The upper dotted hexagon is called a prechart, the lower box a mainchart. Whenever the events prescribed in the prechart happen, the modelled system must conform to the events prescribed in the mainchart.

The hexagon within the prechart is a condition. It describes that the light must be off. The arrow *Press* is a message, denoting that the user has pressed some button so that it is *On*. This means that when the light is off and the on button is pressed, the following **must** happen:

A message *SetState* with the parameter *On* is sent from the *Switch* to the *Light*. Then the *Light* performs a local action, *EnlightenRoom*. This is denoted with a the solid line box. The local action is further unspecified, and only the name may give a hint to what is being done.

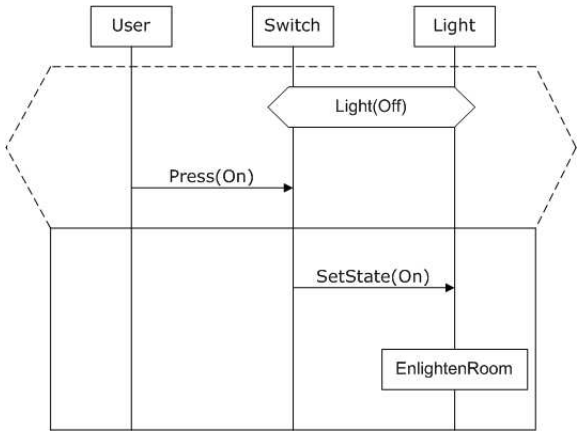

Figure 1.2: An example of a LSC where a user turns on a light using a switch.

Chapter 2

Introduction

Contents

2.1 Motivation **5**
2.2 Contents **6**
2.3 Previous work **6**
2.4 Thesis structure **6**
2.5 The big picture **7**
2.6 Conventions **7**
2.7 Assumptions **8**

2.1 Motivation

The current image of software engineering is not as good as one could wish for as an engineer. Too often there is news of faulty, delayed and overly expensive software projects. Though this problem, termed the software crisis, has existed for decades, it remains unsolved. The increasing complexity and scale of software projects have made the software crisis a bigger problem than ever.

In order to control this escalating problem the use of proper software engineering methods and tools is required, i.e. there is a need for sound engineering principles. This includes the successful phases of domain description, requirements prescription and design specification [5].

One of the core problems during these phases is to communicate and share abstract ideas and concepts. A very popular mean to do so is the Unified Modelling Language (UML). UML is primarily a set of graphical notations to describe, prescribe and specify software. The main force of UML is its use of diagrams. They are intuitive and are easy understood, also by project share-holders which may have little or no insight in software engineering. Unfortunately UML and other graphical notations lack a complete formal foundation. In addition, the current state of formal specification languages do not address some of the more complex language features used in UML. Nor is there necessarily a motivation to do so. A primary goal of formal languages is soundness which is difficult to attain with some features of UML.

A more rigorous and concise approach in software engineering is to use formal specification languages, e.g. the RAISE Specification Language (RSL). They have a well-defined syntax and a complete mathematical semantics. They may also include proof systems for formal reasoning about specifications. These are highly desirable properties which to a great extend are able to reduce error-rates and misunderstandings. Their drawback is that they can only be developed by trained professionals with knowledge of computer science.

Having recognised these two different approaches to software engineering, we come to the core of this thesis: *Combining the strengths of the two worlds*.

2.2 Contents

We will present two different approaches for extending RSL with graphical notations.

The first is to "diagram'ise" RSL specifications with a notation inspired by the UML Class Diagram. We call this new diagram for *Scheme Diagram*. The Scheme Diagram is semantically directly linked to RSL. It is only visually inspired by the UML Class Diagram with regards to boxes and arrows, i.e. the presentation of the structure of a model.

In order to demonstrate the capabilities of Scheme Diagrams we created a Scheme Diagram CASE tool. It is an integrated plug-in for the Java based IDE *Eclipse*. We call this tool *Eclipse Scheme Diagram Editor* (ESDE). It can be used for creating Scheme Diagrams, checking the well-formedness of the corresponding RSL model and save it as a concrete RSL specification in `.rsl` files.

In order to specify the mapping from the Scheme Diagramsyntax to RSL, the RSL types are described using RSL. Furthermore, a specification for printing these RSL types as concrete RSL specification text will be given for the above mentioned use.

The second approach is to formalise a subset of the rather new graphical notation of Live Sequence Charts (LSCs) which we call RSL Sequence Charts (RSCs). It can be used for describing/prescribing inter-object communication. We will integrate and explore the possibility of creating a useful RSL specification based on RSCs. The types and well-formedness conditions are given. Furthermore semantics for these are given. Finally an equivalent applicative RSL version of RSC s is presented.

2.3 Previous work

This master thesis is based on several reports written at IMM[1] that are worth mentioning. The first are [42] and [41]. They are the preliminary thesis and master thesis by Christian Krogh Madsen. They discuss the integration of graphical specification techniques with the formal specification language RSL.

The next paper is a special course report [31] which analyses the relationship between UML class diagrams and RSL. This master thesis is preceded by a preliminary thesis. In the preliminary thesis [2] the possibility to add a graphical notation to RSL was investigated. Furthermore the goal was to look at the integration of the notation of Live Sequence Charts. The latter being based on the work on Message Sequence Charts by Krogh Madsen [42, 41].

More detailed information about the above papers and other used literature are given in the applicable chapters.

2.4 Thesis structure

In this *Introduction* part the graphical notations that are discussed in this thesis are introduced in chapter 1. This should give the reader an immediate idea of the notations used in this thesis.

The part *Abstract models* discusses the abstract models that are created. It presents the main achievements in the thesis. Concepts are described and corresponding formal models are given. The third part, *Concrete Implementation*, describes the concrete implementation of ESDE. It describes the choices and architecture of the editor. Finally a discussion of the results will be presented.

Most of the specifications and auxiliary information have been put in appendices in the end. The thesis is also accompanied by a CD-ROM which contains the work presented here. Relevant papers, source code and an executable version of ESDE is also given.

A glossary of terms in Scheme Diagrams and RSCs are given first in the appendix. It may aid the reader when reading the part on the abstract models.

[1] Informatics and Mathematical Modelling Institute at the Technical University of Denmark

2.5 The big picture

In the beginning of the preliminary thesis [2] the "project tree" as seen in figure 2.1 was created. It served as a guide in the project by identifying branches of work that could be pursued.

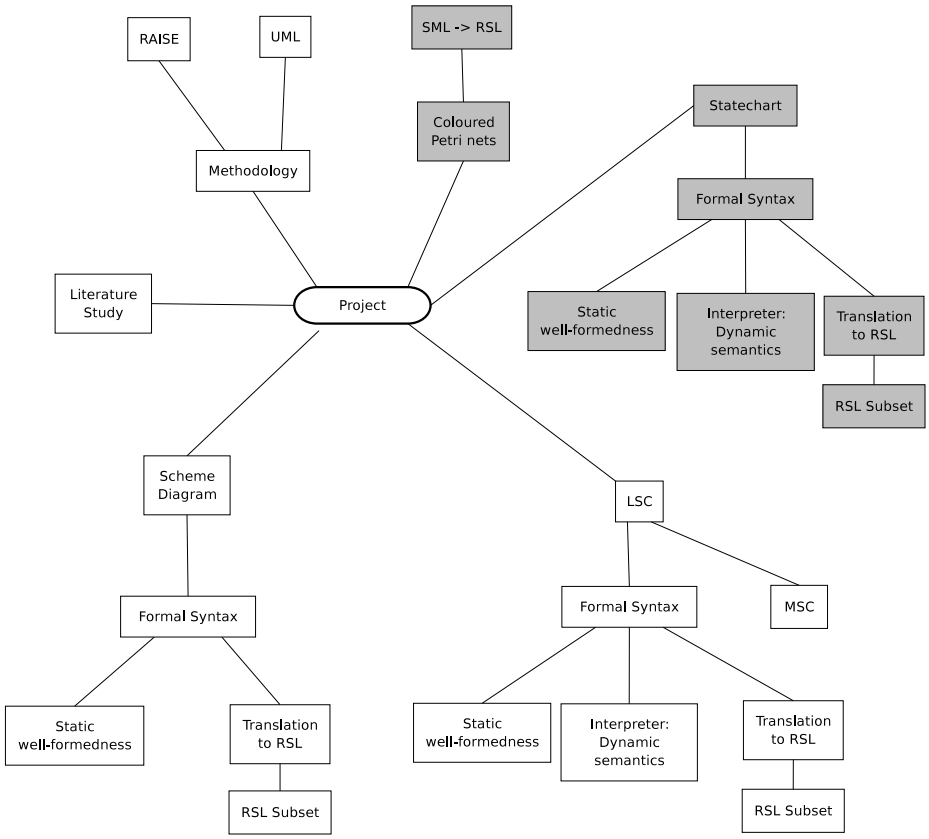

Figure 2.1: The initial project tree of the project. The items with grayed out boxes have not been treated.

Petri Nets were quickly dismissed as it is already a graphical notation with a formal foundation. The initial idea of including State Charts was that they should complement LSCs. LSCs describe inter-object communication, whereas State Charts are used for intra-object communication. However, as our knowledge about LSCs increased it became evident that this coupling was not very promising as a result of the already existing research in the field of executable LSCs. Furthermore, attempts of formalising state charts had already been made.

The implementation of ESDE was given a higher priority than an implementation for RSCs. Therefore the translation for RSCs was not needed, and therefore omitted.

2.6 Conventions

All specifications presented in the appendix of this report type check with the *rsltc* tool v. 2.5 [9] provided by UNU-IIST. Note that there have recently been changes to the language such as *hd* of a set and *isin* on maps.

Throughout the report we talk about *translatable* RSL specifications. By this we mean concrete RSL specifications that are translatable to C++ by the *rsltc* tool. Note that only a subset of RSL is translatable.

In order to distinguish concepts we introduce the following symbols as subscript:

- α denotes **A**bstract
- γ denotes **C**oncrete
- δ denotes **D**iagram

2.7 Assumptions

As many aspects in this report are rather technical it must be assumed that the reader has a working knowledge of RSL. Furthermore knowledge about UML class diagrams and scenario-based graphical notations (e.g. Message Sequence Charts or Live Sequence Charts) is an advantage, but is not required.

Part II

Abstract models

Chapter 3

Introduction

In the following chapters the concepts, theory and the abstract models are presented. Initially an abstract RSL syntax in RSL is given. This was used as a foundation for creating RSL specification using ESDE. The work regarding Scheme Diagram is presented, followed by the work on LSCs/RSCs.

Common for most specifications is that they are translatable from RSL to C++ using the *rsltc* tool [9]. This allowed for RSL test case generation. Furthermore it allowed the direct linking from the Scheme Diagram specification to ESDE. I.e. using the translated Scheme Diagram specification to check well-formedness of a model drawn in ESDE.

But the translatability posed quite some constraints, as only a subset of RSL is translatable to C++. Therefore the specifications from the preliminary thesis [2] had to be severely altered before they could be reused. As an example RSL union constructs cannot be translated. These had to be rewritten to equivalent variant definitions. Another constraint was that the support for quantifiers is limited. As a result the specifications looks unnecessary complicated. This is due to the need of auxiliary functions for determining sets that can be used for the quantification.

Chapter 4

RSL syntax

Contents

4.1 Types: RSL$_\alpha$ **13**
4.2 Print: RSL$_\alpha$ → RSL$_\gamma$ **14**

4.1 Types: RSL$_\alpha$

The syntax for the Scheme Diagram and LSC have all been specified in RSL. A mapping for each of these diagrams to RSL was to be specified, thus requiring an abstract formal syntax for RSL in RSL. This will be presented in this section. It is only the type declarations for the RSL syntax that have been included and not the well-formed conditions. For our purpose the type declarations are sufficient, since it can be made concrete.

The syntax is based on the input for the original RAISE tool and updated with the following changes made to RSL since:

- **with** ... **in** expressions
- Prefix + and -
- == symbol
- Finite maps

The names of the type declarations in the specification are the same as in the *RSL Reference Description* found in part II in [18].

See appendix B for the complete formal RSL types. The following is an excerpt of the class expression types.

class_expr ==
    class_expr_from_basic_class_expr(class_expr_to_basic_class_expr : basic_class_expr) |
    class_expr_from_extending_class_expr(class_expr_to_extending_class_expr : extending_class_expr) |
    class_expr_from_hiding_class_expr(class_expr_to_hiding_class_expr : hiding_class_expr) |
    class_expr_from_renaming_class_expr(class_expr_to_renaming_class_expr : renaming_class_expr) |
    class_expr_from_with_class_expr(class_expr_to_with_class_expr : with_class_expr) |
    class_expr_from_scheme_instantiation(class_expr_to_scheme_instantiation : scheme_instantiation),

basic_class_expr :: decl*,

extending_class_expr :: class_expr class_expr,

hiding_class_expr :: {| dil : defined_item* • **len** dil $>$ 0 |} class_expr,

4.2 Print: $\text{RSL}_\alpha \rightarrow \text{RSL}_\gamma$

The following specification was created in order to allow the creation of a textual RSL specification based on a specification given in abstract RSL. It thus needs to convert the various RSL constructs in RSL_α to an equivalent textual representation with appropriate RSL keywords and delimiters.

The foundation for this function was the already existing structure of RSL_α. This was converted one at a time to a print function. This tremendously eased the process and minimised the chance of errors, as the structure was preserved. The correct amount of delimiters and newlines were not of concern. The *rsltc* pretty printer functionality was to be used to arrange the output and make it more readable. As the RSL syntax was given using may short record definitions and variant definitions a lot of the work could be done using automated Emacs macros.

As the specification is rather large, it is presented in appendix C. The following is an example that shows how a basic class expression is printed:

```
print_basic_class_expr(x) ≡
    case (x) of
        mk_basic_class_expr(a) →
```

A class expression needs the keywords *class* and *end* before and after the class expression declarations.

As the following example shows, the emphasis was on preserving the structure of the functions, rather than optimising for size.

```
print_specification(x) ≡ print_module_decl_list(x),

print_module_decl_list : module_decl* → Text
print_module_decl_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ⌢ ⟨⟩ → print_module_decl(a),
        ⟨a⟩ ⌢ b →
            print_module_decl(a) ⌢ ",\n" ⌢
```

Two special functions, RSL_int_to_string and RSL_double_to_string have been supplied under-specified in order to allow conversion from integers/doubles to strings. In the generated C++ code they must simply be removed. The compiler will automatically link calls to these functions to the supplied RSL-libraries where they are defined.

Chapter 5

Scheme diagrams

Contents

5.1 Introduction 16
5.1.1 Introductory example 16
5.1.2 Why Scheme Diagrams? 16
5.1.3 Is RSL object oriented? 17
5.1.4 Structure of chapter 21
5.2 Previous work 22
5.2.1 Literature 22
5.2.2 Discussion 24
5.3 Summary of preliminary thesis 24
5.4 Final Scheme Diagram 25
5.4.1 Static implementation 25
5.4.2 Scheme instantiation 27
5.4.3 Object state 29
5.4.4 Executable specification 30
5.5 Narrative of the Scheme Diagram syntax 32
5.5.1 Diagram 32
5.5.2 Type expressions 33
5.5.3 Scheme 35
5.5.4 Object 38
5.5.5 Association 40
5.5.6 Extend 42
5.5.7 Static implement 44
5.6 Examples 46
5.6.1 Mobile infrastructure 46
5.6.2 Constructed example 49
5.7 Translation: $\mathbf{SD}_\delta \rightarrow \mathbf{RSL}_\alpha$ 50
5.8 Future work 50
5.9 Conclusion 51

5.1 Introduction

5.1.1 Introductory example

First a bootstrapping example of **a** Scheme Diagram for **the** Scheme Diagram is given. It will present an overview of the syntax and well-formedness of the Scheme Diagram. Thus it will give an intuition of what the remainder of this chapter is about and how the formalisation is composed. The diagram is presented in figure 5.1.

In the top left corner the `types` scheme is shown which contains all the types necessary to describe the syntax of the Scheme Diagram. This is extended with some convenient auxiliary functions in `auxiliary`. This is followed by additional six extends each adding the well-formedness for one of the major parts of the Scheme Diagram. For example the `wf_types` scheme primarily contains well-formedness functions for types representing type expressions.

The `wf_model` scheme is the complete description of a well-formed Scheme Diagram. From this point there are two different usages. The first is further extension with tests. The scheme `examples` defines the input for the test cases defined in the scheme `test`.

The types and well-formedness specification have been prepared for translation into C++ and will be used by a CASE tool for drawing Scheme Diagrams. The translation is based around the `imperative` scheme which besides adding an imperative layer also is the interface to the tool.

Before proceeding the different kinds of lines will be explained. Based on the narrative so far it is possible to deduce that lines with a hollow equal-sided triangle at one end denotes the `extend` construct in RSL. The lines with a filled diamond pointing towards the `imperative` scheme represents the declaration of nested objects. Lines with a hollow diamond pointing towards the scheme `transltr` represent parameterisation of the scheme.

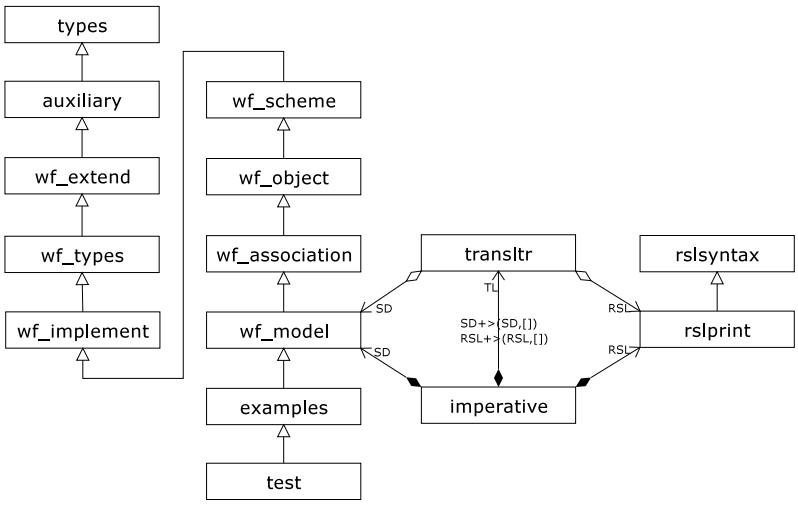

Figure 5.1: Scheme Diagram of the Scheme Diagram specification.

The `rslsyntax` scheme is an abstract syntax in RSL for the RSL syntax. The extending scheme `rslprint` is able to translate the abstract syntax into a concrete RSL specification represented as a text string. The `imperative` scheme declares three nested objects which respectively include: The syntax and well-formedness of the Scheme Diagram, the abstract syntax of RSL and the translator from the Scheme Diagram syntax to the abstract RSL syntax. The latter is a parameterised scheme and the first two objects declared in `imperative` are used as actual parameters.

5.1.2 Why Scheme Diagrams?

A picture says more than a thousand words is a cliché but nonetheless many would say it is true. Like a picture a diagram also conveys its information in two dimensions. It can be inferred that a diagram also says

	Object based criteria									Object oriented criteria									
	1. Object and states	2. Encapsulation	3. Synchronous requests	3. Asynchronous requests	4. Data abstraction	5. Data struct. of objects	6. Object identity	7. Intra-concurrency	8. Inter-concurrency	9. Class as templates	10. Class as collections	11. Inheritance	12. Sub-typing	13. Multi-inheritance/sub-typing	14. Inheritance $\neq$ sub-typing	15. Classes as objects	16. Collection of objects	17. Dynamic/static object instantiation	18. Genericity or parameterization
Object-Z	•	•	•		•	•	•	•	•	•		•	•	•	•		(n)		•
VDM++	•	•	•		•	•	•	•	•	•		•	(•)	•	•		(•)	(•)	
Z++	•	•	•		•	•	•	•	•	•		•	•	•	(n)		•	•	•
RSL	•	•	•		•	•	•	•	•	•		(•)	•	(•)	•			(•)	•

Table 5.1: Comparison table from [20] between object oriented specification languages (modified). Only the languages based on first-order logic and set-theory are included, and RSL has been added. • stands for yes, 'n' for no and parentheses for partial.

a thousand words. This is of course a logical game based on weak assumptions. But it reaches the conclusion that UML users, among others, already have shown to be true in practice:

> [12]: A stated strength of OO modelling notations is their intuitive appeal, reducing the effort required to read and understand the models. ∎

Especially the Class Diagram which displays the static structure of a system has been widely accepted. The diagram is however limited to describing only the structure. On the other hand formal specification languages are very capable of describing an entire system. But they lack the intuitive appeal requiring that one actually has to read the specifications. Obviously the two should be combined. There are several different approaches of doing so. One is to formalise UML as it is, which is done in [13, 28]. Another is to extend an existing formal specification language with object oriented constructs. This has amongst others been done in VDM++ [33].

The Scheme Diagram is a diagram made for RSL with inspiration from the UML Class Diagram. The elements of the diagram have a direct mapping to RSL constructs. Thus the semantics is clear. The inspiration from the Class Diagram is how the boxes and lines in the diagram are depicted. Similar constructs in UML and RSL will be displayed in a similar way.

The example in section 5.1.1 has hopefully demonstrated the usefulness and intuitive appeal of Scheme Diagrams without entirely disclosing the reminder of the chapter.

5.1.3 Is RSL object oriented?

The motivation for creating a new diagram for RSL instead of formalising the UML Class Diagram is based on the fact that RSL is not object oriented. Therefore it is in place to elaborate on why it is not. In [20] a set of criteria are presented for categorising a language as object-based and object-oriented. These criteria are selected by the authors of the paper and are not necessarily widely accepted as the *correct* criteria for classification. In this section the criteria will be used to categorise RSLsince they are the most comprehensive and structured set available. This is summarised in table 5.1 which also includes three languages for comparison.

In the following listing the italic text is quoted from [20, p. 3-5] and the normal text describes the possibilities in RSL. Not all the criteria are clearly defined and they are not elaborated further in the paper. In such cases an interpretation will be mentioned.

Object based criteria

1. *Object: The ability to consider a system as a collection of independent entities which interact and collaborate, i.e. an object gives some services to the other objects and can request some as well; every entity possesses a state which can be modified.*

   A RSL specification is a collection of modules; modules being schemes and objects. A scheme is a named set of models and an object is a specific model within a given set of models.

   The objects of a specification can interact and collaborate but the module dependency must be a tree structure; circular dependencies are not allowed. Mutual interaction can still be achieved using channels.

   A state can optionally be specified for a set of models using variable declarations. [18, chp. 28].

2. *Encapsulation: The ability to hide the state of objects from the outside, the only way to interact with an object is to request on of its services.*

   Identifiers defined within a class expression may be hidden from the outside using the `hide` construct. The identifiers can be type, value, variable, channel, and object declarations. [18, sec. 29.2].

3. *Synchronous/asynchronous: ... Generally, synchronisation requests are achieved by method calling, while message passing is associated with asynchronous requests.*

   Both functions and CSP message passing available in RSL are synchronous. It is possible to model asynchronous message passing using an intermediate buffer process between two communicating processes, but it is not natively available. [18, sec. 38.7]

4. *Data abstraction: The ability to describe abstract data types, as distinct from individual objects; these models are called hybrid in contrast to pure object models, in which only objects can be used as modelling entities.*

   In RSL it is possible to define abstract data types (sorts) which are types. Types and modules are different concepts. [18, sec. 3.3,38.4.1].

5. *Data structures of objects: The ability to describe data structures of objects such as stacks or arrays of objects.*

   In RSL an identifier can be bound to an array of models. In the following example *L* is bound to an array of objects:

   scheme Lists = **class**
   object L[i : ListNo] : **class** ... **end**
   type ListNo = {| n : **Nat** • n < 2|}
   end

   The part postfixed to the object identifier, [i : ListNo], is the index type and thus the size of the array. The binding, *i*, is available to the class expression and can i.e. be used for index dependent instantiation. [18, sec. 32].

6. *Object identity: The notion of a persistent identity for an object. The object identity is unique within the system.*

   Object declarations introduce a persistent identifier for an arbitrary model within a specified class expression, [18, sec. 28.2,38.3]:

   object O : **class** ... **end**

7. *Intra-object concurrency: The ability to express concurrent events inside an object.*

   Two processes may be evaluated in parallel in RSL, using the parallel operator ||, as shown with the `run` function:

channel c : **Unit**
value
run : **Unit** → **in** c **out** c **Unit**
run() ≡ foo() ‖ bar(),

foo : **Unit** → **in** c **Unit**
foo() ≡ c? ; foo(),

bar : **Unit** → **out** c **Unit**
bar() ≡ c!**skip** ; bar()

If parallel processes have write access to variables then they get copies of those variables. This means that there can be no interference between two processes except through channel communication. Sharing a variable is done by wrapping it in a process that changes the value of the variable via channel communication. [18, sec. 24.4]

8. *Inter-object concurrency: The ability to have concurrent progress of objects.*

   Inter object concurrency is supported in RSL using objects. In the following example, objects A and B are parallel processes which may communicate through the channel c of object C.

   object A : **class value** foo : **Unit** → **in** C.c **out** C.c **Unit end**,
   object B : **class value** bar : **Unit** → **in** C.c **out** C.c **Unit end**,
   object C : **class channel** c : **Unit end**

   A better structuring would be to declare *A*, *B*, and *C* as objects within the same class and pass *C* as an actual parameter to *A* and *B*. Now *C* can be hidden in the class and communication between *A* and *B* is secure since *C* cannot be accessed by anything else than *A* and *B*.

Object oriented criteria

9. *Classes as templates: The ability to describe the common aspects of objects and to create (statically or dynamically) instances or objects of the class; each class defines a type which is associated with all instances of the class (intensional description)*

   In RSL schemes are named class expressions which can optionally be manipulated before being used to define one or more objects. Thus they are used as templates. [18, sec. 28.3].

10. *Classes as collections: The ability to describe homogeneous collections of existing objects (extensional description).*

    This criteria is interpreted as the ability to describe commonalities of objects instantiated by different class expressions. In RSL this is only relevant when using parameterised schemes which is the only situation where objects are used as expressions. Formal parameters do not restrict the actual parameters to be instances of a certain class expression. Instead they prescribe requirements that the actual parameters must fulfil such as the presence of a particular type or axioms that must hold. [18, sec. 30.5].

11. *Inheritance: The reuse or modification of an existing class in order to obtain a new one. Three distinct scenarios are considered:*
    (a) Specialization inheritance: Some ingredients may be added.
    (b) Redefinition inheritance: Some services may be redefined.
    (c) Design inheritance: Some ingredients may be removed.

    The `extend` construct in RSL gives the ability to extend one class expression with another; that is, the resulting class expression is the result of extending the scope of one class expression to include the scope of another. It is required that the two class expression are compatible, meaning that it is possible to add new definitions but not to redefine nor remove existing definitions. [18, sec. 28.4]

    Overloading, i.e. value definitions with same name but different maximal signatures, is allowed. But redefinition is not directly supported using extend. It is however still possible by combining extend with

renaming. In the following example the function foo in A is renamed to foo_old in B. Hence there is no function in B with the name foo which means that C may define one.

scheme A = **class**
  value
    foo : **Int** → **Bool**
    foo(i) ≡ i > 5
end,

scheme B = **use** foo_old **for** foo **in** A,

scheme C = **extend** B **with class**
  value
    foo : **Int** → **Bool**
    foo(i) ≡ **true**
end

With this approach the original definition is still in the specification which was probably not the intention. However if scheme A fulfils a contract then scheme C will also fulfil the contract. This is not entirely true since the redefinition of the function could violate the theory but C does statically implement A and hence fulfils the contract statically.

Regarding removal of *ingredients* both `rename` and `hide` could be used, but neither do actually remove declarations.

12. *Sub-typing: The ability to express a hierarchy of types based on the substitutivity principle. Three cases are considered.*
    (a) Weak sub-typing: Only the profiles of the methods are considered.
    (b) Strong sub-typing: The semantics of all the methods are considered.
    (c) Observational sub-typing: Only a subset of the properties (the so-called observable ones) is considered in the used logic.

    The substitutivity principle is an essential element of the RAISE method and the implementation relation. Weak sub-typing as described is similar to static implementation and strong sub-typing is similar to implementation where the theory also is considered. Observational sub-typing is exactly covered by sub-typing expression in RSL. The predicate used is free to observe only a subset of the properties of the super-type. [18, chp. 11]

13. *Multiple inheritance/sub-typing: The ability to construct a class by means of more than one class, or the ability for a type to have more than one super-type.*

    The *extend* construct can be used several times in succession thus achieving multiple inheritance:

    scheme A = **extend** B **with extend** C **with class** ... **end**,
    scheme B = **class** ... **end**,
    scheme C = **class** ... **end**

    This is no different than single extension and the schemes must all be compatible. [18, sec. 28.4,39.3]

    Only one type can be super-type in a subtype expression. The desired effect could be achieved using a Cartesian product or a short record definition as super-type. [18, chp. 11].

14. *Inheritance ≠ sub-typing: The ability to build the inheritance hierarchy distinct from the sub-typing hierarchy.*

    Extension and sub-typing are two different concepts. The *extend* construct is an operator on class expressions and sub-typing on type expressions.

15. *Classes as object: The ability to consider a class definition as an object. Therefore, there is some notion of meta-class (i.e. the class of classes).*

    This concept is not present in RSL.

16. *Collection of objects: The ability to make a collection of heterogeneous interacting objects, and to act on the whole collection.*

    This is not possible in RSL.

17. *Dynamic/static creation of objects:*

    RSL only supports static creation of objects. Dynamic creation could be modelled using object arrays but this would not be true dynamic creation. Dynamic instantiation of parameterised schemes would be problematic since it is unknown until the creation which object(s) should be used as actual parameters.

18. *Genericity or parameterisation of classes:*

    RSL allows schemes to be parameterised with objects. [18, chp. 30]

Discussion

Table 5.1 summarises the evaluation of the criteria for RSL together with the languages of first-order logic and set theory evaluated in [20]. RSL fulfils all the object based criteria except for asynchronous messages which none of the other languages in the table do. Many of the object oriented criteria are supported in RSL, but some essential features are missing in order to be object oriented:

The *extend* construct which provides inheritance in RSL extends the scope of one class expression to include another, requiring that the two are compatible. It is thus not possible to overload a function with the same maximal signature in the extending class expression, without using the trick of renaming the overloaded function to another name as describe in criteria 11.

An important criteria for an language to be object oriented is dynamic creation of objects which RSL does not support. It can be modelled to some degree with object arrays but still poses a problem with parameterised schemes. Another related issue is that RSL types and objects are two different concepts. It is thus not possible for a function to return a reference to an object (pointer). An object type is missing.

An issue which is not clearly captured by the criteria is the constraint that RSL modules must not be recursively dependent. The module dependency must form a tree or a forest of trees. Recursive dependency is possible in object oriented modelling as references to objects are possible.

There is a rationale behind the limitations, which is to keep RSL sound. For example the use of recursion between classes means that the semantics has to be explained in terms of macro expansion - unfold everything into one big context. Then the semantics is not compositional, an we cannot in general talk about the semantics of individual classes. Reasoning also is not compositional. [16]

Another point of view on recursion between modules is whether this is the right way to model. Dijkstra said that the use of *goto* resulted in spaghetti code. Some are of the opinion that recursion between classes gives spaghetti modules.

Although it is possible to model many of the criteria it must not become to cumbersome to do so, since the benefits of object oriented modelling are likely to be less significant. It is concluded that RSL should be categorised as a object based language and not object oriented.

5.1.4 Structure of chapter

Section 5.2 will present a literature study continued from [2]. It will cover related approaches and will be supplemented by a discussion. Section 5.3 will summarise our work on the Scheme Diagram in [2], which is followed by section 5.4 covering the technical advancements in the Scheme Diagram. A complete narrative is presented in section 5.5 which presents all the parts of the Scheme Diagram. It is encouraged to read this section before section 5.4. The narrative is followed by two non-trivial examples of the Scheme Diagram. Both the narrative and the examples should provide a good understanding of how the Scheme Diagram is represented in RSL. The non-trivial issues of translation is covered in section 5.7. The chapter ends with comments on future work for the Scheme Diagram in section 5.8 and a conclusion in section 5.9. For this chapter in general only selected parts are included from the formal specifications. The complete specifications can be found in appendix C.

5.2 Previous work

Related and previous work to the Scheme Diagram will be presented in this section as a literature study. This is followed by a discussion of the overall perspective of the literature study. Our preliminary thesis is summarised in section 5.3.

5.2.1 Literature

[12] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Comput. Stand. Interfaces*, 19(7):325–334, 1998.

> Although this paper was written with UML v1.1 in mind it still seems relevant. In particular it describes UML as the best Object Oriented modelling experience available though it suffers from a lack of precise semantics. The paper presents an overview for the formalisation approach that the *precise UML group* (pUML) is using to give UML a proper semantics and justifies its work by discussing the problem that UML has. In the context of the Scheme Diagram the following observations are interesting:
>
> It is noted that a strength of Object Oriented modelling is the intuitiveness of the diagrams which reduces the efforts to read and understand the models. However due to the lack of proper semantics for Object Oriented notations the understanding for different readers may not be consistent.
>
> The UML architects state, according to the paper, that the precision of syntax and semantics is a major goal and claim to provide a "complete syntax" using meta-models and a mixture of natural language and an adaption of formal techniques.
>
> The paper acknowledges the use of a meta-model for describing the syntax of UML but states that it cannot be used to interpret the semantics of non-trivial UML structures, especially when the semantic meta-model is expressed in a subset of the notation it tries to interpret.
>
> The homepage of *pUML* indicates that they are still active and involved with the definition of UML v2.0:
> `http://www.cs.york.ac.uk/puml/`

[3] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel. Formalization of the Whole-Part Relationship in the Unified Modeling Language. *IEEE Trans. Softw. Eng.*, 29(5):459–470, 2003.

> The paper is concerned with the whole-part relationship in the UML Class Diagram being association of kind Composition or Aggregation. Using UML v1.4 a proposal for a change to the meta-model for UML v2.0 is given.
>
> The problem is that Composition is declared as a subtype of aggregation, but no actual subtyping is present in the meta–model. Additionally well-formedness for multiplicity for the two kinds is missing.
>
> The characteristics of the Whole-Part relationship is analysed and categorised as either primary or secondary. Based on the characteristics, a modification to the UML meta-model is proposed. Instead of using the association relation with a meta-attribute indicating Composition or Aggregation, it introduces a new meta-class representing the Whole-Part relationship. This meta-class has the primary characteristics.
>
> Another two meta-classes are introduced representing the Aggregation and Composition relation. Both are sub-classes of the Whole-Part meta-class and are their difference is how their features are different regarding the secondary characteristics.
>
> These new meta-classes are supplemented with constraints written in the Object Constraint Language (OCL).

[28] J. He, Z. Liu, X. Li, and S. Qin. A Relational Model for Object-Oriented Designs. In *Pro. APLAS'2004, Lecture Notes in Computer Science*, Taiwan, 2004. Springer.

The paper identifies object-oriented programming and formal methods as two important but independent approaches to software engineering in recent years. Moreover *objects* are and will remain an important concept.

Besides the following model-based formalisms: Object-Z, VDM++, Syntropy, and Fusion, additional six object oriented languages are mentioned. But all with limitations in one or more features that categorise object oriented modelling.

The paper presents the semantics for a new object-oriented language (OOL) which includes subtypes, visibility, inheritance, dynamic binding and polymorphism. A calculus based on this model is also presented that supports structural and behavioural refinement of object oriented designs.

[35] Yang Jing, Long Quan, Li Xiaoshan, and Zhiming Liu. A Predicative Semantic Model for Integrating UML Models. In *Proceedings of the 1st International Colloquium on Theoretical Aspects of Computing (ICTAC)*, 2004.

It is mentioned that the majority of the existing formal support to UML-based development focus on formalisation of individual diagrams. Thus the consistency between different diagrams are maintained.

This aim of the paper is a semantic model of UML based on the Object Oriented Language (OOL) introduced in [28]. A syntax for a simple Class Diagram is presented using OOL which include classes with attributes and methods, inheritance and binary associations with multiplicity but not aggregation. A syntax for sequence diagrams is presented where each time line represents an object or multiple objects. Messages between time lines are included. Simple well-formedness rules are presented for the two diagrams.

[33] IFAD. The Rose–VDM++ Link. Technical report, IFAD, Forskerparken 10A, DK-5230 Odense M, 2000. Revised for v6.6.

The report is a manual for the *Rose-VDM++ Link* which is an add-on to Rational Rose 98/2000[1]. Without going into technical details, it describes the architecture of the link. It is divided into three categories: mapping from UML to VDM++, mapping from VDM++ to UML, and synchronising UML and VDM++ models. A short tutorial of Rational Rose is also given.

The mapping rules from VDM++ to UML is presented by showing VDM++ specifications along with the corresponding UML translations. The mapping rules have been defined such that they are injective, thus the reverse mapping from UML to VDM++ is defined at the same time. Constructs that can only be described in one of the two representations are left out of the mapping.

- Instance variables and value definitions are basically the same but the latter is read-only. These are mapped to attributes using the stereo types *«instance variable»* and *«value»* to distinguish the two.
- VDM++ Operations are explicit and functions are implicit. These are mapped to UML operations which only include the signature. Therefore stereotypes are again used to distinguish the two: *«operation»* and *«function»*.
- In VDM++ objects may have relations to objects of other classes using the object reference type declared in the instance variable clause. These are mapped to UML associations and are not shown as attributes. The relations can be bidirectional and recursive. Multiplicity is modelled using sets and ordered sets.
- There is a direct mapping of inheritance.
- In VDM++ operations can be delegated to subclasses using the *subclass responsibility*. This corresponds to declaring operations abstract in UML. This means that the class becomes abstract and marked by displaying the class name in an italic font.

[1] Visual modelling tool for UML.

[20] N. Guelfi, O. Biberstein, D. Buchs, E. Canver, M-C. Gaudel, F. von Henke, and D. Schwier. Comparison of object-oriented formal methods. Technical report, University of Newcastle Upon Tyne, Department of Computing Science, 1997.

> The goal of the document is the classification and comparison of object oriented formalisms for assessing their suitability within the DeVa [10] project. The paper notes that the term "OO formal approach" is used for several different approaches.
>
> The first part of the document describes the classification criteria selected. The criteria are divided into two categories: *Object based* and *object oriented*. The object based criteria primarily describe encapsulation and methods for accessing the encapsulated state. Moreover the objects are organised in a static structure. What is lacking compared to the object oriented criteria are real inheritance and objects as value (pointers).
>
> The main aspects that are missing in object based formalisms compared to object oriented in the presented criteria is that no real inheritance is provided and objects may not be used like values nor may objects be used as arguments to functions.
>
> In total thirteen formalisms are compared divided into four categories of languages:
>
> - First-order logic and set-theory: Object-Z, VDM++, Z++
> - Algebraic: HOSA, TROLL, Maude, AS-IS
> - Class Orientation with Nets: CLOWN, CO, OPN, CO-OPN/2
> - Temporal Logic: TRIO+, OO-LTL

5.2.2 Discussion

The apparently large acceptance in industry and research interest in UML indicates the potential of the project. It has, however, not reached its goal yet as several papers suggest. As [12] states: the semantics of UML is not formally defined and it is also used to define itself. Two different concrete problems in the Class Diagram regarding the semantics of the association relation, are mentioned in [21] and [3] respectively. A consequence of the incomplete semantics is that the interpretation of UML diagrams rely on informal descriptions and intuition.

To rectify this problem several approaches have been tried to combine UML with formal methods. In [13] the Class Diagram is formalised and in [39, 35, 28] a new formal language is developed for object oriented modelling. In [33] a formal language is extended to include object orientation. Common for most approaches is that they only describe a subset of UML. Often a single diagram is formalised or even a subset of a single diagram.

Another noticeable aspect is that the diagrams and features defined by UML are widely being accepted. A consequence is that the formalisations and formal languages are adapted accordingly. In [48] the approach of "UML'ising" formal techniques is presented. This is also the basis for the Scheme Diagram initially presented in the preliminary thesis [2] and further developed in this thesis.

5.3 Summary of preliminary thesis

In [31] a relationship between the UML Class Diagram and RSL is sought through examples and formalisation of the UML Class Diagram. The examples show that there indeed is a relationship and that diagrams can be a supplement to a RSL specification. The formalisation of a part of the UML Class Diagram syntax does also show that this is not a small task. Numerous attempts have been made to formalise subsets of UML but none have been adopted by OMG which maintains the developments of UML.

The Scheme Diagram, first presented in our preliminary thesis [2], is inspired by the work done in [31]. We started with a different approach by "UML'ising" formal methods instead of formalising UML. That is, a new diagram is created, tailored for RSL. It maintains the formal foundation of RSL and is inspired by the UML Class Diagram which already has proven its worth. In the remainder of this section the features of the Scheme Diagram introduced in [2] are presented. This is based partly on [2, sec. 2.10].

Schemes, being named class expressions, are supported by the diagram with type, value, variable, channel and axiom declarations. This is achieved by fully supporting type expressions. Additionally hiding of declarations is supported. Objects are included but must be an instance of a scheme. Extend is introduced as a relation between two schemes. The association relation is introduced which is a precondition for qualification. Additionally static implementation is introduced as a relation.

Regarding the static implementation as a relation in the diagram the underlying formalisation is not sound. It is missing formal parameters and refinement of sorts and the use of qualification is not considered. The state of an object neglects the variables available through qualification. A more profound detail that is missing is scheme instantiation. It is thus not possible to specify which actual parameter should be used for a given formal parameter. As a consequence hereof, object fitting is also missing.

It is chosen in [2] not to support renaming, the **with** operator and class expressions which are not named.

5.4 Final Scheme Diagram

This section will present the larger technical advancements made to the Scheme Diagram since it was first presented in [2]. In particular scheme instantiation and object fitting have been introduced, and the static implementation relation and object state have been improved. Further discussion and elaboration is covered in section 5.4.1, 5.4.2 and 5.4.3. Furthermore the entire RSL specification for the Scheme Diagram has been made translatable. That is, rewritten to the subset of RSL which can be translated into C++. This is covered in section 5.4.4. The purpose of making the model executable is to reuse it in conjunction with the tool presented in part III. A side effect is the possibility of testing the model. Test cases are present in appendix C.4.

In [2] the concept of diagrams is introduced with the purpose of presenting selected parts of a well-formed model. This feature is primarily interesting from a CASE tool point of view and not from a modelling point of view. With this in consideration and in order to simplify the specification, the feature has been removed. It is possible in [2] to specify a list of role-names for a given association which is a shorthand for associations differing only in the role-names. It has been removed for the same reasons.

Although fitting of objects have been included, renaming of schemes is not supported. This is further discussed in section 5.8. The newly introduced **with** operator on class expressions is not supported. It is similar to global objects without qualification but does not add new functionality to RSL; it is simply introduced for convenience.

The to expressions *scheme instantiation* and *instance of* are similar but in fact different. The first stands for the class expression which the scheme represents evaluated in the context of the actual parameters [18, sec. 39.6]. The second is used to state that a given model (object) is in a given set of models represented (class expression) [18, sec. 28.2].

The two terms $client$ and $supplier$ are essential in understanding relations of the diagram. The two terms refer to the participating modules of a relationship, and are also used in the Object Oriented terminology and furthermore used in [19, p. 38]. The *client* is the dependent module and the *supplier* is the providing module.

5.4.1 Static implementation

A narrative of the static implementation relation for the Scheme Diagram is described in section 5.5.7. In order to verify its well-formedness it is necessary to determine whether a $client$ scheme statically implements a $supplier$ scheme: $client\_id \sqsubseteq_s supplier\_id$. The $\preceq_s$ symbol is used for static implement between class expressions and the $\sqsubseteq_s$ symbol between scheme id's. Static implementation is also used in conjunction with object instantiation where actual parameters must statically implement formal parameters. It is described in section 5.4.2.

In the Scheme Diagram only named class expressions are used, hence it is only necessary to determine static implementation between named class expressions: Schemes. This is divided into two major parts. First the signatures of the two schemes are determined. Second it is determined if the signature of the $supplier$ is included in the $client$.

The specification described in this section is based on that described in [2]. In [2] it was, however, far from complete since it only included class expressions, no parameters or nested objects. Additionally determining if the $supplier$ is part of the $client$ was done without considering the context.

The complete formal specification of the static implementation relation in the Scheme Diagram is present i appendix C.1.5.

Signature

The signature of a scheme is the basis for determining static implementation. It denotes an identifier and a maximal class expression. Formal parameters of a scheme are not included in the signature but are dealt with through recursion in the comparison. Normally in RSL nested objects are considered a clause within a class expression. In the Scheme Diagram all objects are relations and are not modelled as part of a scheme. Consequently the signature of nested objects must be treated separately. The signature of the remaining clauses is determined by the following three steps:

1. Make one basic class expression by eliminating extend relations.
2. Rewrite variant, union and short record definitions to sorts and value definitions.
3. Determine the maximal class.

Creating one basic class expression is done by first determining the class expression of the scheme. If the scheme is $client$ of an extend relation then the class expressions of the $supplier$ is merged with the original class expression. If the $supplier$ is $client$ in an extend relation, this is repeated. Any declaration that is hidden and not part of the initial class expression is omitted.

Variant, union and short record declarations are all short hands for a sort definition, one or more value definitions and two or more axioms [18, chp. 12, 15]. The axioms are not relevant for static implementation and are ignored. After rewriting the maximal class can be determined. This is done by finding and substituting all type expressions with their corresponding maximal type expression.

Determining the nested objects of a scheme is done similiar to eliminating extends. The nested associations for the initial scheme are determined. If the scheme is a $client$ in an extend relation then the nested associations of the $supplier$ are added. This is done recursively. If a client of a private nested association is a supplier of the initial scheme, it is omitted.

Comparison

As already stated only schemes are considered in conjunction with static implementation in the Scheme Diagram. Static implementation between two schemes is defined as follows [15, p. 54-55]:

- The number of formal parameters must be the same.
- The formal parameters of the $supplier$ scheme must statically implement the parameters of the $client$ scheme.
- The class expression of the $client$ scheme must statically implement the class expression of the $supplier$ scheme.

Substitutivity is a desired property of the implementation relation [19, sec. 1.6] captured by static implementation. This is also the reason that the static relation for the formal parameters is reversed. If a $client$ scheme strengthens its formal requirements only a subset of the actual parameters accepted by the $supplier$ scheme is accepted. Thus the $client$ will not be able to substitute the $supplier$. It is however allowed for the $client$ to loosen its requirements since it still accepts all the actual parameters of the $supplier$.

In RSL the formal parameters of a scheme is a list. The formal parameter of the $supplier$ must statically implement the formal parameter of the $client$ with the same index. In the Scheme Diagram this ordering is not present since the parameters are drawn as lines connected to the scheme box. In RSL and thus the Scheme Diagram the IDs used for the parameters must be unique and the same IDs must be present in a refinement. Hence this does not pose a problem.

The static implementation relation for class expressions is defined as follows:

[18, sec. 30.5]: $class\_expr_2$ statically implements $class\_expr_1$ if the maximal signature of $class\_expr_1$ is included in the maximal signature of $class\_expr_2$ ∎

Included is a broad definition which need to be elaborated for each of the clauses of a class expression:

type Since variants, unions and short record definitions are rewritten, only sorts and abbreviation definitions are considered. A sort of the $supplier$ is included in the $client$ if there exists a sort or abbreviation with the same name. An abbreviation of the $supplier$ is included in the $client$ if the name and maximal signature are the same.

value A value definition in the $supplier$ is included in the $client$ if there exists a value definition with the same name and maximal signature. Additionally variable and channel accesses of the $client$ must be a subset of those in the $supplier$. [18, sec. 30.6.4]

variable A variable definition of the $supplier$ is included in the $client$ if its maximal signatures are the same [18, sec. 30.6.2].

channel A channel definition of the $supplier$ is included in the $client$ if its maximal signatures are the same [18, sec. 30.6.2].

axiom Axiom definitions do not have signatures and are disregarded [18, sec. 30.5].

object An object definition of the $supplier$ is included in the $client$ if there exists a new object definition with the same type. Additionally the class expression of the new object definition must statically implement the old. If the old is an array then the new must also be an array and the indices must statically implement the old. [18, sec. 30.6.1]

Sorts may be refined into abbreviations which means that the maximal signatures of type, value, variable and channel definitions may change in the refinement. Consider the following example:

```
scheme A1 = class
    type T
    value x : T
end
```

```
scheme A2 = class
    type T = Int
    value x : T
end
```

The scheme *A2* statically implements *A1* but the maximal type of x in *A1* is T and in *A2* it is **Int**. It is thus necessary to consider the context. In [38, sec. 3.2] a signature morphism is described that addresses this problem. We have adapted this approach to the Scheme Diagram. A map from type names to type expressions is generated for each type definition in the $client$ scheme. Before determining if a value definition from the $supplier$ is present in the $client$, all type names in its type expression are substituted by the type expression in the map with the type name as lookup key.

5.4.2 Scheme instantiation

The syntax for the Scheme Diagram in [2] does not support scheme instantiation of parameterised schemes, because it is not possible to specify actual parameters for the formal parameters. The formal parameters of a scheme are all the associations of kind parameter which the scheme is a $client$ of. When instantiating a scheme all the formal parameters must be instantiated with an actual parameter being an object available within the instantiating scheme. Since both the actual and formal parameters are described as associations and thus as relations between two modules, there is no ordering information. That is, the order in which the formal parameters are written in an RSL matters with regards to specifing which actual parameter is used for a given formal parameter. Hence we introduce the following type:

```
type
    ActualParameters = Name ~m~> Name × Fitting,
    Fitting = Name ~m~> Name,
```

The type `ActualParameters` is a map from the name of a formal parameter to the name of an actual parameter together with fitting information of the actual parameter. The type `Fitting` is a map from "old" names to "new" names.

Renaming of class expressions is not supported by the Scheme Diagram, but fitting of objects is included. The rationale is that generic parameterised schemes loose their potential without object fitting. Fitting is used in conjunction with objects as actual parameters. The relationsship between the actual and formal parameters are as follows: The class expression of the actual parameter must statically implement the class expression of the formal parameter. If fitting of the actual parameter is included then the class expression of the actual parameter must statically implement the class expression of the formal after the class expression of the formal parameter has been renamed with the fitting: [18, sec. 30.5, 40.5]

object AP : ce_{ap},
scheme FP = ce_{fp}

AP{xx **for** yy} $\sqsubseteq_s$ FP $\equiv$ ce_{ap} $\preceq_s$ **use** xx **for** yy **in** ce_{fp}

Thus fitting requires renaming, but it is only required once just before instantiation. Thus the argument is that complete rename support is more complicated for less functionality. See section 5.8 for a discussion about renaming in the Scheme Diagram.

The `ActualParameters` information is relevant when instantiating parameterised schemes which may happen with global objects and associations of kind parameter and nested. When using the extend construct in RSL in conjunction with parameterised schemes, it is required to provide actual parameters to the $supplier$. In the Scheme Diagram an extending scheme will inherit the formal parameters of the supplier and use these as actual parameter when instantiating the supplier. Since the same name and class expression are used it is not necessary to add the `ActualParameters` to the `Extend` type in the Scheme Diagram. See section 5.5.6.

In order to instantiate a global object with a parameterised scheme other global objects must be used as actual parameters. It should also be possible to fit the global objects which are used as actual parameters thus the type `ActualParameters` is added to the type `Object`.

type
  Object ::
    instance_of : Name
    actual_parameters : ActualParameters
    state : State

The set of names of the formal parameters of the scheme which the object is an instance of must thus equal the domain of `ActualParameters`. This would mean that each formal parameter has an actual parameter.

Instantiating a parameterised scheme as a global object requires other global objects as actual parameters. In the Scheme Diagram objects are only available through associations, hence it becomes necessary to allow associations of kind Global between two global objects. This does, however, introduce an inconsistency since neither the object nor the association can be added to the diagram before the other is present. The object requires the association in order to use the supplier as actual parameter. The association requires the object to be present otherwise the client end will be dangling. We chose to accept this transition from well-formed diagram to non well-formed and back. The argument is that this is normal for development including that of RSL specifications written in text. It is not well-formed until the user is finished typing (this is of course an insufficient precondition for well-formedness).

scheme ELEM : **class type** Elem **end**
scheme ITEM : **class type** Item **end**
scheme B : **class type** Boat **end**
scheme STACK(e : ELEM) : **class** ... **end**
scheme QUEUE(i : ITEM) : **class** ... **end**

object b : B
object s : STACK(b{Elem **for** Boat})
object q : QUEUE(b{Item **for** Boat})

In the example the object *s* will be an instance of STACK and the actual parameters would be:

$[''\texttt{e}'' \mapsto (''\texttt{b}'', [''\texttt{Boat}'' \mapsto ''\texttt{Elem}''])]$

The second place where object instantiation and fitting are used is with associations of kind Parameter and Nested. Global associations do not denote instantiations since the $supplier$ is a global object, thus no role name is specified and it will not be necessary to include fitting information either. This information is present in the global object. Both parameters and nested objects are scheme instantiations and can thus have formal parameters which must be instantiated with actual parameters. As with global object instantiation the actual parameter is a name of an object which is available within the instantiating context and again optional fitting information is included.

type
Association ::
client : Name
kind : Kind
supplier : Name
rolename : Name
mul : Multiplicity,
Kind ==
Nested(Visibility, ActualParameters) |
Parameter(ActualParameters) |
Global,
Multiplicity == None | Index(binding : Name, mtype : TypeExpr)

The domain of the actual parameters is again the names of the formal parameters of the supplier scheme, hence the domain of the actual parameters and the set of formal parameters names must be equal.

The following functions are relevant to scheme instantiation:

- (*wf_scheme_instantiation*, C.1.6): Determines if the actual parameters statically implements the formal parameters.
- (*wf_actual_parameters*, C.1.6): Check if there is an actual parameter for each formal parameter and that the object names used as actual parameters are actually available.
- (*wf_object*, C.1.6): Checks the well-formedness of global objects which includes a call to *wf_scheme_intantiation*.
- (*wf_association*, C.1.8): Checks the well-formedness of associations of kind Nested and Parameter- which includes a call to *wf_scheme_intantiation*.

5.4.3 Object state

How to depict the state of an object in the Scheme Diagram is inspired by the UML Class Diagram. A Class Diagram in UML can additional to classes also contain objects. Actually an Object Diagram is merely a Class Diagram without classes. An object in the Scheme Diagram must be an instance of a scheme, therefore it is not possible to have a Scheme Diagram only containing objects.

It is recognised in [17, p. 435] that the use of the Object Diagram is limited, but may be used to illustrate a complex state. This is also true for showing the state of an object in the Scheme Diagram. It is not possible to translate the state into RSLsince the values represent a given state in time.

The state of an object in the Scheme Diagram is composed of the variables declared in the scheme which the object is an instance of. The declared variables are however not limited to the immediate scheme but also public variables available through the `Extend` relation and public variables available through nested associations. Only private declared variables from the immediate scheme is visible in the object state. Other private declared variables, through `Extend` or `Association` relations, are not visible to the object and thus not included in its state. See figure 5.2.

The state is represented in RSL as a mapping from a variable to its value. In order to uniquely identify the variables the qualification needed to reach the variable is included. Since value expressions are not included, the state is not translatable and the values are simply modelled as a text giving the user freedom.

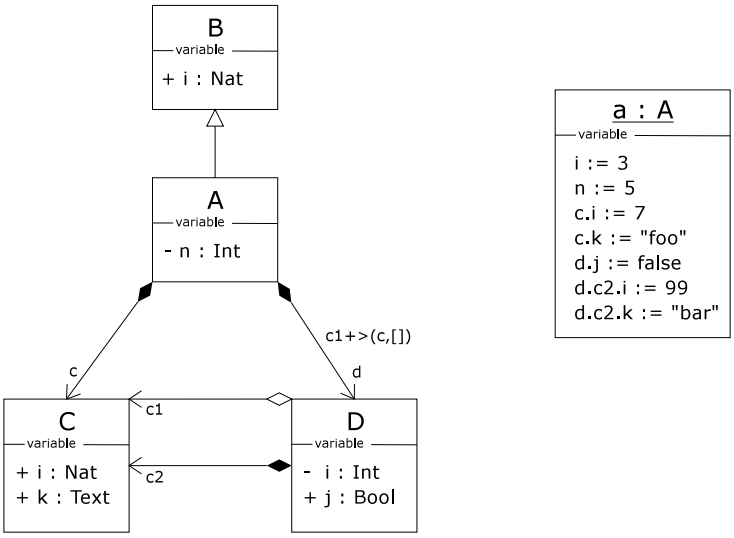

Figure 5.2: Example of an object with a state compartment.

type
State = QualifiedName $\overrightarrow{m}$ Value,
Value = **Text**

It is only possible to depict one object with the same name in a diagram. Thus it is only possible to depict a single state.

Variables may only be shown as part of a state in conjunction with global objects. Object declarations containing variables are considered part of the declaring scheme. Actual parameters are always global objects or nested association, hence parameter objects are not part of the state since their variables is part of the actual parameter.

The following functions are relevant to object state:

- (*wf_object_state*, C.1.6) Checks the well-formedness of the state, which is whether all the variables available is represented in the state.
- (*state_domain*, C.1.6) Determines the set of names with qualification of the variables which constitute the state.

5.4.4 Executable specification

The complete specification for the Scheme Diagram is executable in the sense that it can be translated into C++ using the *rsltc* tool. Only a subset of RSL can be translated for the following two reasons. First, some of the expressions are not possible to translate with a reasonable outcome, such as general for-all quantification (∀). Second, the translator to C++ is simple not finished, e.g. it is capable of translating variant and short record definitions but not union definitions. The specification presented in [2] is not translatable and an effort has been put into rewriting offending expressions. The remainder of this section will describe some of the encounters in the process of making the Scheme Diagram executable. When two RSL texts are listed in the remainder of this section, then the left is from [2] and the right from appendix C.

Unions are used in several places in [2] but as just mentioned, they cannot be translated. Unions are simply a short hand for a variant definitions where all constructors have exactly one type expression, thus it is possible to rewrite unions to variants. This would reduce the readability of the specification. Instead we decided to reconsider the type definition `Model'`. Conceptually the Scheme Diagram is divided into modules and relations which is reflected in the definition in [2]. Instead we chose to model each element of the diagram in a separate map and remove the concept of *diagrams*. This has simplified the model of the Scheme Diagram, e.g. determining if a given name is a scheme. Removing the concept of *diagrams* was also done in order to

simplify the well-formedness in general. Unions are in [2] also used in the type definition of type declarations and type expression which also have been rewritten.

```
type
  Model′ ::
    modules : Name m→ Module
    relations : RID m→ Relation
    diagrams : DID m→ Element-set,
  Module = Scheme | Object,
  Relation = Association | Extend | Implement,
  Element = Name | RID
```

```
type
  Model′ ::
    schemes : Name m→ ClassExpr ↔ replace_schemes
    objects : Name m→ Object ↔ replace_objects
    associations : RID m→ Association ↔ replace_associations
    extends : RID m→ Extend ↔ replace_extends
    implements : RID m→ Implement ↔ replace_implements
```

Another issue for the translation to C++ is curried functions. The `suppliers` function in [2] has a predicate as parameter which determines which relations to consider. The function returns the set of names of modules which are suppliers to the specified module. The predicate could restrict the suppliers to consider associations only. Since it is not possible to specify a predicate it is necessary to create additional two functions which restrict the relations considered. (*suppliers_ass, suppliers_ext*, C.1.2)

```
value
  suppliers :
    T.Model′ × (T.Model′ × T.RID → Bool) →
      T.Name ~→ T.Name-set
  suppliers(mdl, p)(m) ≡
    {supplier_of(mdl, r) | r : T.RID •
      is_relation(mdl, r) ∧ p(mdl, r) ∧
       client_of(mdl, r) = m}
  pre is_scheme(mdl, m) ∨ is_object(mdl, m)
```

```
value
  suppliers : Model′ × Name ~→ Name m→ RID
  suppliers(mdl, s) ≡
    [ supplier_of(mdl, r) ↦ r | r : RID •
        r ∈ relations(mdl) ∧
        client_of(mdl, r) = s ]
  pre s ∈ modules(mdl)
```

The same situation holds for the *associations* functions which, provided a name of a module, determines the available objects. In this case a predicate could select what kind of association should be considered. (*associations*, C.1.2)

A third issue is recursive calls in quantification expressions and comprehended expressions in general. They are not supported by the translator. This is solved by creating a new recursive function and instead of e.g. a quantified expression the new function is called. Consider the case for product type expressions in the following example:

```
wf_type_expr : Model′ × Name × TypeExpr → Bool
wf_type_expr(mdl, m, te) ≡
  m ∈ modules(mdl) ∧
  case te of
    tl_Unit → true,
    tl_Bool → true,
    ...
    BracketedTypeExpr(te′) → wf_type_expr(mdl, m, te′),
    ProductTypeExpr(tel) → wf_product_type_expr(mdl, m, tel)
  end,

wf_product_type_expr : Model′ × Name × TypeExpr* → Bool
wf_product_type_expr(mdl, m, tel) ≡
  tel = ⟨⟩ ∨
  (tel ≠ ⟨⟩ ∧ wf_type_expr(mdl, m, hd tel) ∧
   wf_product_type_expr(mdl, m, tl tel))
```

Here the call to function `wf_product_type_expr` replaces the more simple expression, which besides the recursive call would have been accepted by in the translation to C++.

```
∀ te′ : TypeExpr • te′ ∈ tel ⇒ wf_type_expr(mdl, m, te′)
```

Although some of the elegant notations of RSL may not be used when considering translation, we still think that the specifications has maintained its readability.

5.5 Narrative of the Scheme Diagram syntax

In this section an informal description will be given of the Scheme Diagram and its elements. This is supplemented with the types used to define the diagram and selected functions that will contribute to an in-depth understanding. The complete formal specification for the Scheme Diagram is presented in appendix C. Figure 5.1 in section 5.1.1 presents an overview of the formal specification. A glossary is provided in appendix A.1 with terms used in the Scheme Diagram. The narrative is based on [2, chp. 2] but has been modified and updated.

5.5.1 Diagram

Types

The purpose of the scheme diagram is to give an overview of the modules which are used to describe a system. The diagram basically consists of boxes and lines connecting those boxes. The boxes represent RSL modules being either schemes or objects. In this context it should be mentioned that contrary to a normal RSL specification it is only possible for objects to be instances of schemes; that is, instantiation with an unnamed class expression is not supported. The rationale for doing so is to follow the example of the UML Class Diagram which has classes and objects which to some degree can be compared to RSL schemes and objects respectively. In the UML Class Diagram it is necessary to have a class before it is possible to instantiate an object. Another reason is that it is simpler to keep track of the class expressions in the model if they are named and a scheme is simply a named class expression. Objects can consequently only be instantiated by named class expressions in the Scheme Diagram.

The lines between boxes denote relationships. There are three different relationships included in the Scheme Diagram. They are Association, Extend and Implement. Association is concerned with references to other modules. Extend represent the construct available in RSL with the same name. Implement[2] indicates to the user that a static implementation relation holds between the two schemes. In the Scheme Diagram all relations are binary and directed.

```
type
    Model′ ::
        schemes : Name -m→ ClassExpr ↔ replace_schemes
        objects : Name -m→ Object ↔ replace_objects
        associations : RID -m→ Association ↔ replace_associations
        extends : RID -m→ Extend ↔ replace_extends
        implements : RID -m→ Implement ↔ replace_implements,
    RID = Nat
```

When describing a diagram which must depict RSL it is perhaps a bit misleading when using the word *model* in the formal description since it is an important term in the RSL terminology. A scheme being a set of models and an object a single model. However the name "Model" has been chosen also inspired by UML by having a complete model.

A well-formed Scheme Diagram is represented as a subtype which fulfills all the well-formed conditions described in the following sections. (*wf_model*, C.1.9)

```
type
    Model = {| mdl : Model′ • wf_model(mdl) |}
value
    wf_model : Model′ → Bool
```

Well-formedness

1. All boxes must be either a Scheme or an Object and these must be uniquely identified by their name. (*wf_module_names*, C.1.9)

[2] The name *Implement* is used for the relation in the Scheme Diagram as a shorthand for static implement.

2. All lines must be either be an Association, an Extend or an Implement. All relations must have an unique identification regardless of its kind. (*wf_relation_ids*, C.1.9)
3. Circularity between modules are not allowed in RSL and will hence not be allowed in the Scheme Diagram: There must not be circular relations between modules. It is possible for the static implementation relation to be cyclic, e.g. two identical schemes. In order to completely avoid cyclic structures in the diagram we have chosen to disallow it. (*wf_non_cyclic*, C.1.9)

```
wf_non_cyclic : Model′ → Bool
wf_non_cyclic(mdl) ≡
  ∼ (∃ s : Name •
       s ∈ schemes(mdl) ∧ path(mdl, s, s)),
```

The `path` determines if there exists a directed path between two modules in the model. All the relations in the Scheme Diagram are directed in the sense that one module is the supplier and the other is the client and the direction is from client to supplier. There are no restrictions to the modules which are part of the path except that they are part of the model. (*path*, C.1.9)

```
path : Model′ × Name × Name ~→ Bool
path(mdl, org, dst) ≡
  path(mdl, org, dom suppliers(mdl, org), dst)
pre {org, dst} ⊆ modules(mdl),
```

```
path : Model′ × Name × Name-set × Name ~→ Bool
path(mdl, org, intermediate, dst) ≡
  intermediate ≠ {} ∧
  (dst ∈ intermediate ∨
   let n = hd intermediate in
     path(mdl, n, dom suppliers(mdl, n), dst) ∨
     path(mdl, dst, intermediate \ {n}, dst)
   end)
pre {org, dst} ⊆ modules(mdl)
```

5.5.2 Type expressions

Types

The primary focus of schemes in the Scheme Diagram is on signatures of declarations. In this context type expressions are essential and therefore included in the diagram. It gives the possibility to generate RSL specifications that are syntactically correct. This can be verified by the *rsltc* tool. Secondly it becomes possible to verify the rules of overloading and the static implementation relation between two schemes.

Type expressions are described in detail in [18, chp. 41] and will thus not get a thorough description here. All possible type expressions have been included in the Scheme Diagram, some with limitations. The limitations of type expressions in the Scheme Diagram will be highlighted in the remainder of this section.

```
type
  TypeExpr ==
    tl_Unit |
    tl_Bool |
    tl_Int |
    tl_Nat |
    tl_Real |
    tl_Text |
    tl_Char |
    TypeName(Name, Qualification) |
```

```
    ProductTypeExpr({| tel : TypeExpr* • len tel ≥ 2 |}) |
    BracketedTypeExpr(expr : TypeExpr) |
    FiniteSetTypeExpr(TypeExpr) |
    InfiniteSetTypeExpr(TypeExpr) |
    FiniteListTypeExpr(TypeExpr) |
    InfiniteListTypeExpr(TypeExpr) |
    MapTypeExpr(domain : TypeExpr, range : TypeExpr) |
    FunctionTypeExpr(
        param : TypeExpr,
        arrow : FunctionArrow,
        result : ResultDescr) |
    SubtypeExpr(TypeExpr, restriction : QualifiedName),
FunctionArrow == fa_total | fa_partial,
ResultDescr = AccessDescr* × TypeExpr
```

Type expressions are recursively defined where the basic and terminating elements are type literals and type names. Type literals are predefined types within RSL and type names refer to declared types with optional qualification.

The subtype expression is more complex than the other type expressions. It includes a value expression which is a restriction on the specified super type. The subtype thus contains all the values which satisfies the value expression which is required to be a predicate. Since value expressions are not included in the Scheme Diagram the value expression of the subtype cannot be included. Instead the subtype expression has the name of a predicate value expression which must be declared, as shown to the right:

```
y = {| x : Int • x > 10|}
```

```
y = {| x : Int • wf_x(x)|}
wf_x : Int → Bool
```

The actual value expression is not included in the Scheme Diagram, only the signature. It can be added after the diagram has been translated into RSL. The value declaration which is referenced from the subtype expression must, to be well-formed, take supertype as parameter and return a boolean value.

The function type expression consists of four parts: (1) The type expressions for the parameters (2) Result type expression of the function (3) If the function is partial or total (4) Access description for variables and channels. There are many allowed combinations of how accesses to variables and channels may be specified as part of the function type expression. Note that comprehended access is not included in the Scheme Diagram. It is omitted in order to simplify the Scheme Diagram, since accesses are already a complicated recursivly defined type structure. The possible ways of describing access is thus done by explicitly specifying the name of the variable or channel or by using completed access (**any**).

```
type
    AccessDescr = AccessMode × Access*,
    AccessMode == am_read | am_write | am_in | am_out,
    Access ==
        NameAccess(QualifiedName) |
        EnumeratedAccess(Access*) |
        CompletedAccess(Qualification)
```

With an almost full description of type expressions it is possible to determine the maximal type expression (*maximal_type*, C.1.3). As mentioned in the beginning of the section, this is interesting in conjunction with overloading and static implementation. It allows for well-formedness predicates that will ensure that a translated diagram will be syntactically correct.

Well-formedness

4. Type expressions are represented by a recursive structure, e.g. a Cartesian product is a list of type expressions. A type expression is well-formed if each of its contained type expressions are well-formed (*wf_type_expr*, C.1.3). Type literals, e.g. **Int**, are trivially true. A type expression is not well-formed if one of the following criteria fail:

(a) Sub-type expressions: The super-type must be well-formed and the restriction predicate must exist as a value declaration. The type expression of the value declaration must be a function expression with the super-type as parameter and **Bool** as return type. (*wf_subtype_expr*, C.1.3)

(b) Type name expressions: A type name is a reference to a declared type with optional qualification. The name must exist relatively to the context in which the type name is used. (*wf_typename_expr*, C.1.3)

```
wf_typename_expr : Model′ × Name × TypeExpr → Bool
wf_typename_expr(mdl, m, tn) ≡
    case tn of
        TypeName(n, q) →
            valid_qualification(mdl, m, q) ∧
            let schn = follow_qualification(mdl, m, q) in
                schn ∈ schemes(mdl) ∧
                n ∈ declared_type_names(extend_history(mdl,schn))
            end,
        _ → false
    end,
```

(c) Accesses: In conjunction with functions, variables and channels may be accessed. If the access mode is **in** or **out** then the following names must be names of channels. If the access mode is **read** or **write** then the following names must be names of variables. (*wf_access_descr*, C.1.3)

5.5.3 Scheme

Types

Schemes are depicted in the Scheme Diagram as rectangular boxes with between one and six compartments. The top compartment contains the name of the scheme which also is the unique identifier of the scheme within the Scheme Diagram. The remaining compartments contain the declarations of type, value, variable, channel, and axiom – see section 5.5.5 for the reason why the object clause has been excluded. See figure 5.3 for an example of the how the scheme is depicted. This is very similar to how the class is depicted in the UML Class Diagram.

```
type
    ClassExpr ::
        types : TypeDecls ↔ replace_types
        values : ValueDecls ↔ replace_values
        variables : VariableDecls ↔ replace_variables
        channels : ChannelDecls ↔ replace_channels
        axioms : AxiomDecls ↔ replace_axioms
```

As the space available to depict the scheme is sparse and it must give an overview, the information presented must be limited to fit these criteria. In the UML Class Diagram this has been done by only displaying the signature of both attributes and operations; that is, information regarding the type of the attributes and the types of parameters and result of functions. The same approach is chosen for the Scheme Diagram. Thus only including the signature of declarations. All compartments need not be shown. The top compartment is mandatory, the remaining compartments are optional.

An important concept in object oriented modelling is visibility. It denotes the level of visibility of elements within an object to other objects. RSL has corresponding possibilities through hiding; all declarations are visible to other modules unless they are explicitly hidden.

```
type
    Visibility == Private | Public
```

The visibility concept with public and private declarations is thus included in the Scheme Diagram. It is depicted in the scheme diagram to the left of each declaration with a plus sign for public and a minus sign for private.

```
scheme STACK = hide el in
  class
    type Elem
    variable el : Elem*
    value
      push : Elem → write el Unit,
      pop : Unit → write el Unit,
      top : Unit → read el Elem,
      empty : Unit → write el Unit,
      is_empty : Unit → read el Bool
  end
```

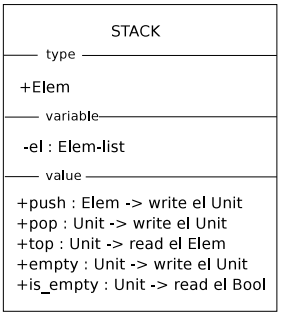

Figure 5.3: An example of a scheme in the Scheme Diagram.

Above is an example of a RSL specification of a stack. The corresponding depiction in figure 5.3 is how the scheme is depicted in a Scheme Diagram. Notice the hiding of the variable `el`. Empty compartments are not shown.

The first compartment holds the type declarations with corresponding type expressions. All the possible type declarations have been included: sort definitions, abbreviation definitions, and variant definitions. The two variant shorthands have also been included: union and short record definitions.

```
type
  TypeDecls = (TypeDecl × Visibility)*,
  TypeDecl ==
    SortDef(Name) |
    AbbreviationDef(Name, TypeExpr) |
    VariantDef(Name, {| vdl : Variant* • len vdl ≥ 1 |}) |
    UnionDef(Name, {| nwl : NameOrWildcard* • len nwl ≥ 2 |}) |
    ShortRecordDef(Name, ComponentKind*),
  NameOrWildcard == udName(qname : QualifiedName) | udWildcard
```

Sorts are the simplest form of type declaration consisting only of a name. Likewise are abbreviations simple consisting of a name and a type expression. Variant declarations are somewhat more complex since it is a shorthand for a sort definition, value function definitions, and some axioms. In conjunction with variant and short record declaration it is possible to specify destructors and reconstructors. It is however only allowed to use reconstructors if there exists a corresponding destructor. This will be part of the well-formedness conditions and not be a concern when expanding the respective type declarations to value function definitions. The formal representation of the union and short record declarations do not differ significantly compared to the variant declaration and are also based on the same type declarations.

In the Scheme Diagram only the name and signature of a value declaration are shown. This is the same for value definitions and function definitions both implicitly and explicitly. It is possible, due to overloading, to have two value declarations with the same name as long as the maximal signature is different. The declarations for variables and channels are similar to that of value declarations. Both declarations consist of a unique name and a type expression.

```
type
  ValueDecls = (ValueDecl × Visibility)*,
  ValueDecl ::
    vdname : Name ↔ replace_vdname
    vdte : TypeExpr ↔ replace_vdte
```

```
type
  VariableDecls = (VariableDecl × Visibility)*,
  VariableDecl ::
    vdname : Name ↔ replace_vdname
    vdte : TypeExpr ↔ replace_vdte
```

```
type
    ChannelDecls = (ChannelDecl × Visibility)*,
    ChannelDecl ::
        cdname : Name ↔ replace_cdname
        cdte : TypeExpr ↔ replace_cdte
```

In a RSL specification the axiom declarations can have an optional name associated. It does not have any effect on the axiom but can provide an intuitive meaning to the axiom if named properly. Additionally it makes references to the axiom more clear from e.g. documentation. For the same reasons and because value expressions are not included, only the name of the axiom is included in the Scheme Diagram thus making it a requirement that every axiom has a name. Axioms are the only declarations which do not have visibility in the Scheme Diagram.

```
type
    AxiomDecls = AxiomDecl*,
    AxiomDecl :: adname : Name ↔ replace_adname
```

Well-formedness

5. The names of the declarations in a class expression must adhere to the following requirements. (*wf_class_expr*, C.1.4)
   - Names within type declarations must be unique.
   - Names within value declarations must be unique unless the maximal signature is different (overloading).
   - Names of variable declarations must be unique.
   - The set of value names and the set of variable names must be disjoint.
   - Names within channel declarations must be unique.

   The functions that check the names of value declarations is included as example. Two different indices are selected from the list of value declaration. Either the names must be different or the maximal type at the selected indices. It is necessary to use indices instead of two value declaration being part of the list. In the latter case the same two will represent the same value declaration thus being equal and the predicate will return false.

```
wf_value_overloading :
    Model′ × Name × ValueDecl* → Bool
wf_value_overloading(mdl, n, valdl) ≡
    (∀ i : Nat •
        i ∈ inds valdl ⇒
            (∀ j : Nat •
                j ∈ inds valdl ∧ i ≠ j ⇒
                    vdname(valdl(i)) ≠ vdname(valdl(j)) ∨
                    maximal_type(mdl, ⟨⟩, n, vdte(valdl(i))) ≠
                        maximal_type(
                            mdl, ⟨⟩, n, vdte(valdl(j)))))
```

6. The type expressions used in the declarations must be well-formed as specified in section 5.5.2. This includes the type, value, variable and channel clauses of class expressions. Axiom and object declarations do not have type expressions. (*wf_scheme_decl_expr*, C.1.4)

   Value, variable and channel declarations are simple in their type structure, consisting of a name and a type expression. Therefore they are also simple to check. Type declarations are more complex, the well-formedness function for union definitions is included as example:

```
wf_union_def :
    Model′ × Name × NameOrWildcard* → Bool
wf_union_def(mdl, n, nwl) ≡
    (∀ i : Int •
```

```
i ∈ inds nwl ⇒
    case nwl(i) of
        udName((n′, q′)) →
            valid_qualification(mdl, n, q′) ∧
            n′ ∈
                declared_type_names(
                    schemes(mdl)(
                        follow_qualification(mdl, n, q′))),
        udWildcard → true
    end),
```

The following must hold for each entry of the list of the right hand side of a `UnionDef`: if the entry is a name, then the name must be a name of a type declaration, possibly with a qualification.

5.5.4 Object

Types

An object is a single model out of the set which a class expression represents. In RSL an object is either instance of a class expression or a scheme. As the Scheme Diagram only include schemes, an object must be an instance of a scheme.

```
type
    Object ::
        instance_of : Name
        actual_parameters : ActualParameters
        state : State,
    State = QualifiedName m→ Value,
    Value = Text
```

The object is depicted as a rectangle with between one and three compartments. The top compartment contains the name of the object and the name of the scheme which it is an instance of, separated by a colon. Additionally the text in the top compartment is underlined to emphasise that it is an object. See figure 5.4. The remaining two compartments are optional and hold actual parameters and the variable state respectively.

```
object INTSTACK : STACK(OI),

object OI : INTEGER,
scheme INTEGER = class
    type Elem = Int
end,

scheme STACK(E : ELEM) =
    hide el in class
    variable el : E.Elem*
    value
        push : E.Elem → write el Unit,
        pop : Unit → write el Unit,
        top : Unit → read el E.Elem,
        empty : Unit → write el Unit,
        is_empty : Unit → read el Bool
end,
scheme ELEM = class type Elem end
```

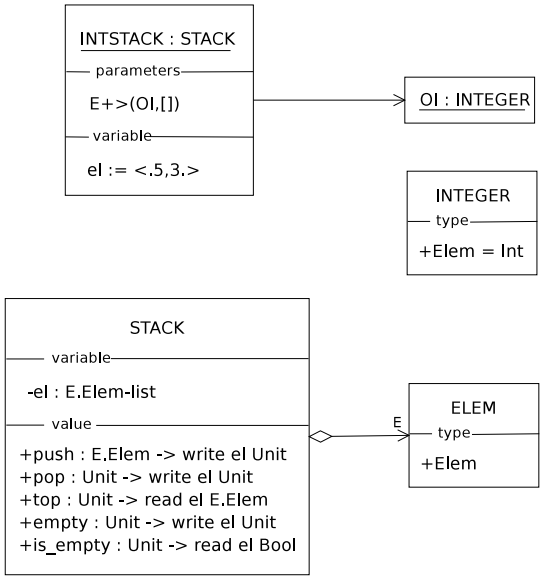

Figure 5.4: Example of objects and schemes in the Scheme Diagram. The lines are introduced in section 5.5.5.

The state of an object is composed of variables and corresponding values. The variables which are part of the state are: All variables of the scheme which the object is an instance of and all public variables which can be reached through nested qualification. Variables available through global or parameter associations are part of another objects state. There are no constraints on the values of the state. Since value expressions are not part of the diagram, it is not possible to specify well-formedness. See section 5.4.3 for a more detailed description of the object state.

The name of the scheme which the object is an instance of, is written in the top compartment after the colon. If the scheme is parameterised then it is necessary to specify the actual parameters. This is done in the *parameters* compartment. For each formal parameter there must be an actual parameter with optional fitting. This is described in detail in section 5.4.2.

Figure 5.4 illustrates the usage of objects. It builds upon the example presented in figure 5.3 but the $STACK$ is now a parameterised scheme. Two global objects are declared in the example: OI and $INTSTACK$. The first is an instance of scheme $INTEGER$ which is neither parameterised nor does it have variables. Therefore only one compartment is shown with the name of the object. The $INTSTACK$ object is an instance of the parameterised scheme $STACK$. It uses the OI object as actual parameter. The lines between the boxes are explained in section 5.5.5.

Well-formedness

7. The state of an object is well-formed if there for all variables available from the object is a corresponding key in the state map (*wf_object_state*, C.1.6). Available variables are:
   - All variables declared in the scheme which the object is an instance of.
   - All public variables inherited by the scheme through the extend relation.
   - All public variables which can be reached via qualification (association).

   Nothing is required about the value of a given variable. It is represented by a text string and the user is free to enter anything. If the value should be verified then it would be necessary to include value expressions which is omitted in the Scheme Diagram.

8. An object must be an instance of a scheme which is part of the model. If the scheme is parameterised then it must be a valid scheme instantiation. (*wf_scheme_instantiation*, C.1.6)

   All the actual parameters must statically implement the formal parameters.

```
wf_scheme_instantiation :
    Model′ × (Name ⇸ Association) × Name ×
    ActualParameters ⟶
        Bool
wf_scheme_instantiation(mdl, avail_objm, supplier, apm) ≡
    let fpm = associations_param(mdl, supplier) in
        wf_actual_parameters(mdl, dom avail_objm, fpm, apm) ∧
        (∀ fp : Name •
            fp ∈ dom fpm ⇒
                let (apn, ap_fit) = apm(fp) in
                    static_implement(
                            mdl,
                            (scheme_name(
                                    mdl, supplier(avail_objm(apn))),
                             ap_fit), supplier(fpm(fp)))
                end)
    end,
```

   Each of the formal parameters must be assigned an actual parameter and the actual parameter must be an object available in the context. (*wf_actual_parameters*, C.1.6)

5.5.5 Association

Types

The association relation is introduced in the diagram to emphasise the use of qualification in the RSL model; that is, when a module makes a reference to another module. Qualification is used in three situations in RSL:

- when a globally declared object is referenced from other modules,
- when parameterised schemes are used and
- when objects are nested within another module.

The name "Association" is borrowed from the UML terminology and has been chosen due to the close similarity with the use of qualification in RSL. The Association is one of the most complex relationships in UML from a modelling point of view, and all the features are not included in the Scheme Diagram counterpart. One reason for using the same name for the RSL relationship is to emphasise similarity to UML.

The Association in the Scheme Diagram is a binary relationship between two modules. It is depicted in the diagram as a solid line between the participating modules. The accompanying ornaments are described below. In the UML Class Diagram an association is a precondition for a link. A *link* is a relationship between objects and can be considered an instantiation of an association. In the Scheme Diagram an association is a precondition for qualification, there is no similiar concept of instantiation of associations. However objects are declared in conjunction with associations of kind parameter and nested, which is mentioned below.

```
type
    Association ::
        client : Name
        kind : Kind
        supplier : Name
        rolename : Name
        mul : Multiplicity,
    Kind ==
        Nested(Visibility, ActualParameters) |
        Parameter(ActualParameters) |
        Global,
    Multiplicity == None | Index(binding : Name, mtype : TypeExpr)
```

The association relationship in the Scheme Diagram is uni-directional unlike its UML counterpart which also can be bi-directional. Thus navigability is introduced to indicate in which direction the association can be traversed. Since it in the Scheme Diagram is only possible to traverse the association relation from the client to the supplier the *navigability arrow* must always be at the supplier end. Hence the navigability arrow is a mandatory part of how the association relation is depicted in the diagram and its purpose is to show which of the participants that are $client$ and $supplier$ respectively. The navigability arrow is depicted as an open arrowhead at the supplier end of the solid line representing the association.

It is possible to specify aggregation for a binary association in the UML Class Diagram which introduces a whole-part relationship between the participants of the relation. A similar notion is introduced for the Association in the Scheme Diagram by using an optional ornament at the client end of the association specifying the *kind*. The kind indicates which of the three possible situations of qualification that is used. The three kinds and their relation to UML are shown in the table:

Kind	Ornament	UML
Global	None	Association
Parameter	Hollow diamond	Shareable aggregation
Nested	Filled diamond	Composite aggregation

The association kind in the Scheme Diagram can to some degree be perceived to have the same meaning as aggregation does in the UML Class Diagram. A globally available object in the Scheme Diagram means that it can be referenced by all other declared modules. When using either parameterised schemes or nested

```
scheme A = class ... end,
scheme B(ba : A) = class ... end,
scheme C = class
    object
        ca : A,
        b[ x : Int ] : B(ca)
    ...
end
```

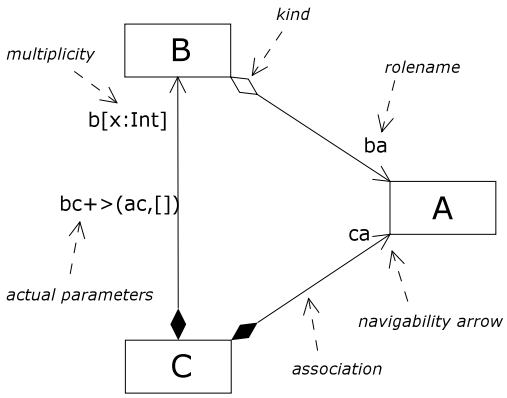

Figure 5.5: Scheme Diagram with associations of kind Parameter and Nested between schemes.

modules there is an introduction of the whole-part relationship mentioned in conjunction with UML. The use of parameters in RSL is thus similar to sharing an object (it should be noted that it is not the same as passing a pointer to the object, which is not possible). Embedded objects are declared and instantiated by the client.

Rolenames are the names used by the client for the instantiated object. These are used with associations of kind parameter and embedded since the supplier of global associations must be an object and thus already has a name. Rolenames are names of the objects which are instantiated with the supplier as the named class expression. The rolenames must therefore be placed at the supplier end of the association when shown in the diagram. Rolenames are not used when the supplier is a global object.

In conjunction with embedded objects and parameterised schemes it is possible to specify arrays of objects. In the scheme diagram this will go under the name multiplicity inspired by the Class Diagram. Multiplicity applies to the supplier and must therefore be placed at the supplier end of the association.

Being able to determine the rolenames (and thereby objects) that can be referenced from a given module, it is possible to see if a given qualification is valid relative to the specified module. Together with validation of qualification it is possible to find the module which is reached by traversing the associations.

Unlike associations of kind Parameter and Nested the Global kind does not have a visible translation into RSL. It could seem superfluous in the diagram, but it is not from a modelling point of view. In general it is required to explicitly state all dependencies in the Scheme Diagram, where a dependency basically is a relation. This is used to determine if there is a cyclic path between the relations and specify dependencies during translation to RSL as required by the *rsltc* tool. Associations are preconditions for qualification and global objects can be used just as parameters and nested objects. Thus the global association provides consistency from a diagramatic point of view.

Well-formedness

9. The multiplicity of an association is well-formed if the type expression of the multiplicity is well-formed. (*wf_multiplicity*, C.1.8).
10. An association is a relationship between two modules. If the kind is Parameter or Nested then it must be between two schemes and the rolename must be nonempty. If the kind is Global then the supplier of the relationship must be an object and the rolename must be empty. The client can either be a scheme or an object. (*wf_kind*, C.1.8).
11. If the supplier of an association of kind Nested or Parameter is the name of a parameterised scheme, then the scheme instantiation must be well-formed. There must be an actual parameter for each formal parameter and the actual parameters must statically implement the formal parameters. (*wf_association*, C.1.8)
12. The rolenames of all the associations in which the scheme directly or indirectly is a client of must be unique. Indirectly is when associations are inherited through the extend relationship. That is all available objects must have unique names in the context of the scheme. (*wf_unique_rolenames*, C.1.7).

```
wf_unique_rolenames : Model′ → Bool
wf_unique_rolenames(mdl) ≡
    (∀ n : Name •
        n ∈ schemes(mdl) ⇒
            let
                ridl = rid_set2list(extend_relations(mdl, n)),
                rnl =
                    ⟨rolename(associations(mdl)(r)) |
                        r in ridl⟩
            in
                len rnl = card elems rnl
            end),
```

The (*extend_relations.rsl*, C.1.7) function returns the set of unique association id's which are unique even if the rolenames of the associations used are not. This is done recursively if the scheme is a child in a extend relationship. The unique id's are used to create a list of rolenames used by the associations represented by the id's. The list of rolenames is used to determine if the rolenames are unique.

5.5.6 Extend

Types

The UML generalisation relationship is a taxonomic relationship between two classes. The client of the relationship inherits the attributes and operations of the supplier class but also the association relationships which the supplier class participates in. The inheritance of association relationships is particularly interesting since it, in our view, improves the readability of diagrams by reducing the number of lines that have to be drawn.

The extend construct in RSL provides a somewhat similar relationship where the client scheme inherits all declarations which are not hidden. The scope of the $client$ class expression is extended to the $supplier$ class expression. In section 5.5.5 a relationship similar to the association relationship in the UML Class Diagram is introduced for the Scheme Diagram with the same name. It is thus obvious that the $client$ of an extend relationship in the Scheme Diagram inherits associations which the $supplier$ participates in. Since the association relation covers nested objects, parameters and global objects it is necessary to see how each of them comply with the extend relationship.

Globally declared objects are available to all modules in the model as long as cyclic module dependencies are avoided. In the Scheme Diagram it is necessary for modules that use global objects to indicate so. This is done by drawing an association of kind global from the $client$ module to the global object. In figure 5.6 a Scheme Diagram is shown with a combination of a global association and an extend relation. As discussed in section 5.5.5 associations of kind Global is primary included due to modelling issues. They are not directly visible in the corresponding RSL specification. Although not visible the association of kind Global is also inherited and used in the Scheme Diagram when determining the available objects of a given scheme and when determining module dependencies.

scheme A = **class** ... **end**,
object OA : A,
scheme B = **class** ... **end**,
scheme C = **extend** B **with class** ... **end**

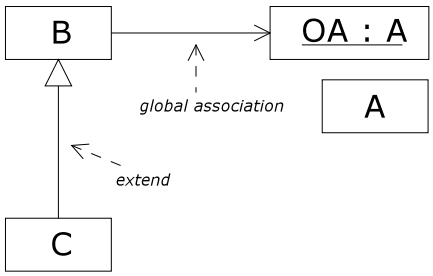

Figure 5.6: Global association combined with the extend relation.

In RSL nested objects are declarations, which means that an extending class expression will include all objects declared in its supplier. See figure 5.7. In the Scheme Diagram it has been chosen to model nested objects as

association relations. For both the nested association and the extend relation there is a direct mapping to RSL which implicitly supports the inheritance of the nested association relation.

scheme A = **class** ... **end**
scheme B = **class object** OA : A ... **end**
scheme C = **extend** B **with class** ... **end**

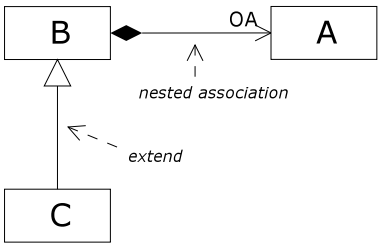

Figure 5.7: Nested association combined with the extend relation.

Parameterised schemes are somewhat more complex than the other two situations, since the $supplier$ scheme must be instantiated. Remember that a scheme instantiation is also a scheme [18, sec. 39.6]. The actual parameters must statically implement the formal. Hence there is no requirement that an actual parameter given is an object of the same scheme as the formal parameter. In order to provide an actual parameter the $client$ scheme must also have a parameters. It is possible to use global objects, but it is not possible to use nested objects. In the Scheme Diagram parameters are inherited, hence a $client$ will have all the parameters of the $supplier$ with the same names and class expressions. See figure 5.8. It is to some degree a limitation of the extend operator, but in our opinion it improves the readability of the diagram by reducing the number of lines drawn. It is still possible for the $client$ to add its own parameters as long as the names are different from the parameters of the supplier.

scheme A = **class** ... **end**,
scheme B(OA : A) = **class** ... **end**,
scheme C(OA : A) = **extend** B(OA) **with**
class ... **end**

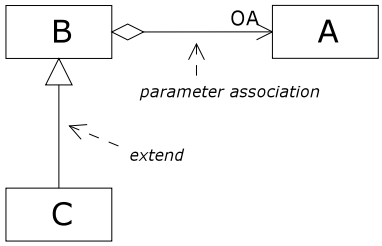

Figure 5.8: Parameter association combined with the extend relation.

Determining the available objects for a given module is an important aspect of the Scheme Diagram. It corresponds to finding all associations in which the module is a client, including inherited associations. For a given module name the `associations` function produces a map from rolenames to the associations with the given rolename and the module as client. If an association is of kind Global then the rolename of the association must be empty since the object already has a name. In that case the name of the global object is used. If the module is a scheme and it is $client$ in an extend relation then this is repeated for the supplier. Objects cannot participate in an extend relation hence no recursive calls will be made. Notice that the `associations` function in the first let expression is a destructor in the `Model′` type and not a recursive call. It gives the map all associations in the model with a unique relation identifier in the domain.

```
associations : Model′ × Name → (Name m→ Association)
associations(mdl, n) ≡
    let am = associations(mdl) in
        [ case kind(am(rid)) of
            Global → supplier(am(rid)),
            _ → rolename(am(rid))
          end ↦ am(rid) |
          rid : RID •
            rid ∈ dom am ∧ n ∈ modules(mdl) ∧
            n = client(am(rid)) ]
    end †
    let ss = suppliers_ext(mdl, n) in
```

```
        if ss = [ ] then [ ] else associations(mdl, hd ss) end
    end,
```

The scheme diagram only allows for a *client* scheme to extend one supplier. Multiple inheritance is omitted to simplify the model of the Scheme Diagram. As discussed in section 5.1.3 item 13, RSL does support this to some degree.

A more complex example is presented in section 5.6.2. It includes both association and extend relations, and parameterised schemes where the formal parameters themselves are parameterised.

Well-formedness

13. The relationship must be between two distinct schemes that are part of the model. (*wf_extend*, C.1.7)
14. A scheme may at most be *client* of one extend relation. There is no limitation on the number of *supplier* roles a scheme can have. (*wf_no_of_extends*, C.1.7)

```
wf_no_of_extends : Model′ → Bool
wf_no_of_extends(mdl) ≡
    (∀ n : Name •
        n ∈ schemes(mdl) ⇒
            let
                sup_ext =
                    {rid |
                        rid : RID •
                            rid ∈ extends(mdl) ∧
                            client_of(mdl, rid) = n}
            in
                card sup_ext ≤ 1
            end)
```

15. If a scheme is *client* in an extend relation then the class expression of the *client* scheme concatenated with its *supplier* (and its *supplier* etc.) must be a well-formed class expression. (*wf_schemes*, C.1.4)

5.5.7 Static implement

Types

The implementation relation is important in the RAISE development method. It is used to determine whether a module is a correct development of a previous module. A brief description of the implementation in RSL is as follows: a scheme *A* implements a scheme *B* when *A* statically implements *B* and the theory of *B* holds in *A*. The theory holds in *A* if the axioms of *B* hold in the context of *A*. Proving if the theory holds is a large and complex task and will not be considered. The remaining part of the implementation relation, when disregarding the theory, is static implementation. Static implementation can be determined considering only signatures, being names and type expressions of the various declarations. Notice that in the RSL specification for the Scheme Diagram the name `Implement` is used for the relation as a shorthand for static implement.

```
type
    Implement ::
        client : Name
        supplier : Name
```

The static implementation relation is depicted in figure 5.9 and 5.10 as a dashed line with an open arrow head. The two ends of the relation are denoted *client* and *supplier*. The *supplier* is the end with the supplier. The interpretation of the relation is that the *client* must statically implement the *supplier*. In the Scheme Diagram, static implementation is a relation between two schemes.

```
scheme S = class
    type T
    value x : T, y : T
    axiom x ≠ y
end,
scheme S1 = class
    type T = Int
    value x : T = 1, y : T = 2
end,
scheme S2 = class
    value x : Int = 1, y : Int = 2
end
scheme S3 = class
    type T = Int
    value x : T = 1, y : T = 1
end
scheme S4 = class
    type T, U
    value x : T, y : U
end
scheme S5 = class
    type T = Int, U = Int
    value x : T = 1, y : U = 2
end
```

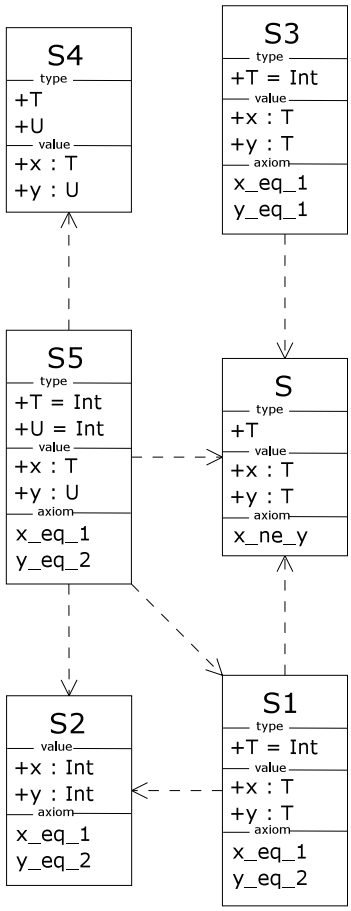

Figure 5.9: Scheme Diagram of the example presented in [18, sec. 30.6].

An in depth and technical discussion on static implementation is given in section 5.4.1. The remainder of this section will describe the relation more loosely and focus on its usage in the Scheme Diagram. The relation is composed of two parts, which consider the declarations of the two schemes and their association relations respectively. A consequence of static implementation is that the $client$ module can substitute the $supplier$.

When considering the declarations, a scheme *A* statically implements a scheme *B* when the maximal signature of *B* is included in *A*. The maximal signature of a scheme is the maximal signature for each of its declarations excluding axioms which do not have a signature. Variant, union, and short record declarations must first be expanded to sorts and value functions. Figure 5.9 shows the example presented in [18, sec. 30.6]. The value declaration of the form: **value** x : **Int** = 1, is not possible to represent in the Scheme Diagram since it is a short hand for both a value and axiom definition. Instead the names of the corresponding axioms are included in the diagram. Notice that $S3$ only statically implements S, since the theory of S does not hold in $S3$. Another observation is that it is not possible to indicate that the relation does not hold, which is also shown in the example. E.g. $S2$ does not statically implement S.

The second part of static implementation in the Scheme Diagram is concerned with associations of kind parameter and nested. The example presented in figure 5.10 illustrates the relationship that must hold when using associations. If scheme D statically implements A then E must statically implement B and C must statically implement F. Notice that the relationship is reversed for formal parameters. This is due to the substitutivity principle. Thus the formal parameter p of D has fewer requirements than the formal parameter of A meaning that any actual parameter that can be given to A can also be given to D.

The Static Implementation relation in the Scheme Diagram is s meta-relationship. It says something about the two schemes it connects, but it does not have a mapping into RSL. It is included in the Scheme Diagram because of its importance. In our oppinion the diagram is well-suited for the relation. Since it is a meta-relationship it may be circular, but will not be in general. It is not allowed in the Scheme Diagram in order to avoid circular dependencies.

scheme B = **class** ... **end**,
scheme C = **class** ... **end**,
scheme A(p : C) =
  class
    object n : B
    ...
  end,

scheme E = **class** ... **end**,
scheme F = **class** ... **end**,
scheme D(p : F) =
  class
    object n : E
    ...
  end

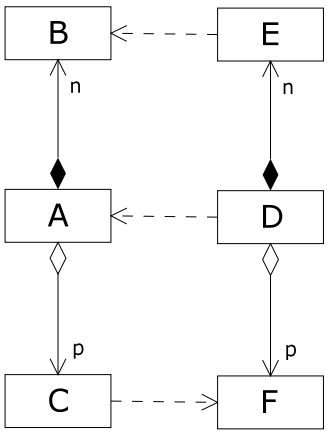

Figure 5.10: The static implementation relation and parameterised schemes.

Well-formedness

16. The *client* and *supplier* must both be schemes and *client* must statically implement the *supplier*. Static implementation is further described in section 5.4.1.

wf_implement : Model′ × RID → **Bool**
wf_implement(mdl, rid) ≡
  rid ∈ implements(mdl) ∧
  let i = implements(mdl)(rid) **in**
    {client(i), supplier(i)} ⊆ **dom** schemes(mdl) ∧
    static_implement(mdl, client(i), supplier(i))
  end

5.6 Examples

Two non-trivial examples are presented in this section. The main focus of the examples is to demonstrate the use of Extend and Association relations. The rationale is that the use of relations is the main element which separates the Scheme Diagram from a normal RSL text.

5.6.1 Mobile infrastructure

In this section the RSL specification from [1] is used to demonstrate the Scheme Diagram. The specification is about the mobility of users/devices within a wireless infrastructure. Only the structure of the specification is interesting in conjunction with a diagrammatic presentation and not the actual purpose. This example is particular interesting since the specification is relatively large. Furthermore it is created independently of the Scheme Diagram. It will thus demonstrate if and how the diagram can supplement an existing specification. The last part of this section lists the relevant parts of the specification, the body of the modules is omitted.

The existing RAISE tool is already capable of generating visual output, called VCG. See figure 5.11. The diagram displays the dependency between modules in the specification. The directed lines are transitive. For example, the module `MSS` is directly dependent on the module `RESREQBAG`, this is achieved through the module `TASK` and thus not shown. The diagram is created by the *rsltc* tool using the `-g` option. It parses the given RSL file and generates input for the Visualisation of Computer Graphs (VCG) tool in a `.vcg` file. An advantage of the VCG is that the layout of the diagram is done automatically. A disadvantage is that is cannot be used to create only to depict existing RSL specifications.

The corresponding Scheme Diagram is shown in figure 5.12. When compared with figure 5.11 it is natural to see, that the two diagrams contain exactly the same number of boxes, one for each module. The difference lies in the lines between the boxes, where additional information is added.

In the RSL specification the **with** ... **in** operator is used, which includes the objects given as parameter in the scope of a class expression. The usage of the **with** operator is primary for convenience, since qualification to the objects can be omitted. Otherwise it corresponds to globally declared objects. In the specification it is used in nearly all modules except for ELEM and RESREQBAG. The Scheme Diagram does not support the **with** operator, it could however have been replaced by global associations from to the object T from the modules: SYS, MHs, MH, MSSs, MSS, TASK, HOARD, RASSIGN and RESOURCES. Notice that RESREQBAG[3] and ELEM do not use the global object. The remaining three modules would inherit the relationship from SYS.

The extra global associations have not been added to the diagram in figure 5.12, since they would certainly degrade the readability. As mentioned in section 5.5.5 the global association is mainly present in the diagram from a modelling point of view. The example show that from a diagrammatic point of view the global association relations is questionable. The global association is thus still an open issue.

scheme TESTING = **with** T **in extend** I_MOB **with class** ... **end**
scheme I_MOB = **with** T **in extend** MOBICHART **with class** ... **end**
scheme MOBICHART = **with** T **in extend** SYS **with class** ... **end**

scheme SYS = **with** T **in class**
  object
    TS : TASK,
    RS : RESOURCES,
    M : MSSs(RS, TS),
    H : MHs(RS, TS)
  ...
end

scheme MHs(R : RESOURCES, TSK : TASK) = **with** T **in class**
  object HS : MH(R, TSK)
  ...
end

scheme MH(R : RESOURCES, TSK : TASK) =
  with T **in class object** RBAG : RESREQBAG(T{R_kind **for** Elem}) ... **end**

scheme MSSs(R : RESOURCES, TSK : TASK) = **with** T **in class**
  object MS : MSS(R, TSK)
  ...
end

scheme MSS(R : RESOURCES, TSK : TASK) = **with** T **in class**
  object RBAG : RESREQBAG(T{R_kind **for** Elem})
  ...
end

scheme TASK = **with** T **in class**
  object
    RB : RESREQBAG(T{R_kind **for** Elem}),
    ASIGN : RASSIGN,
    HL : HOARD
  ...
end

scheme RASSIGN = **with** T **in class** ... **end**
scheme RESREQBAG(E : ELEM) = **with** T **in class** ... **end**

[3] Although it is included in the specification it is not actually used. Hence the dependency can be removed.

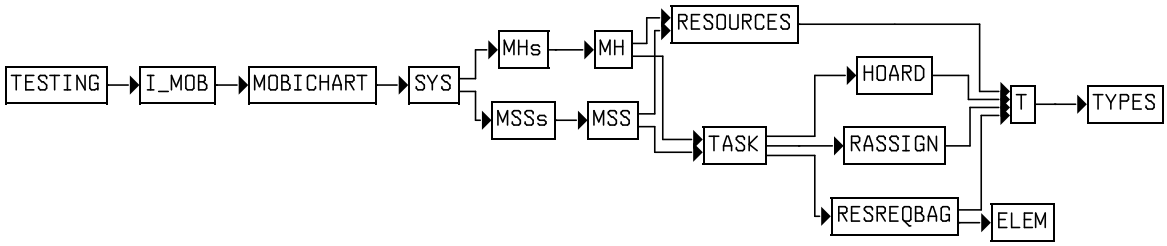

Figure 5.11: VCG chart of the mobile infrastructure specification.

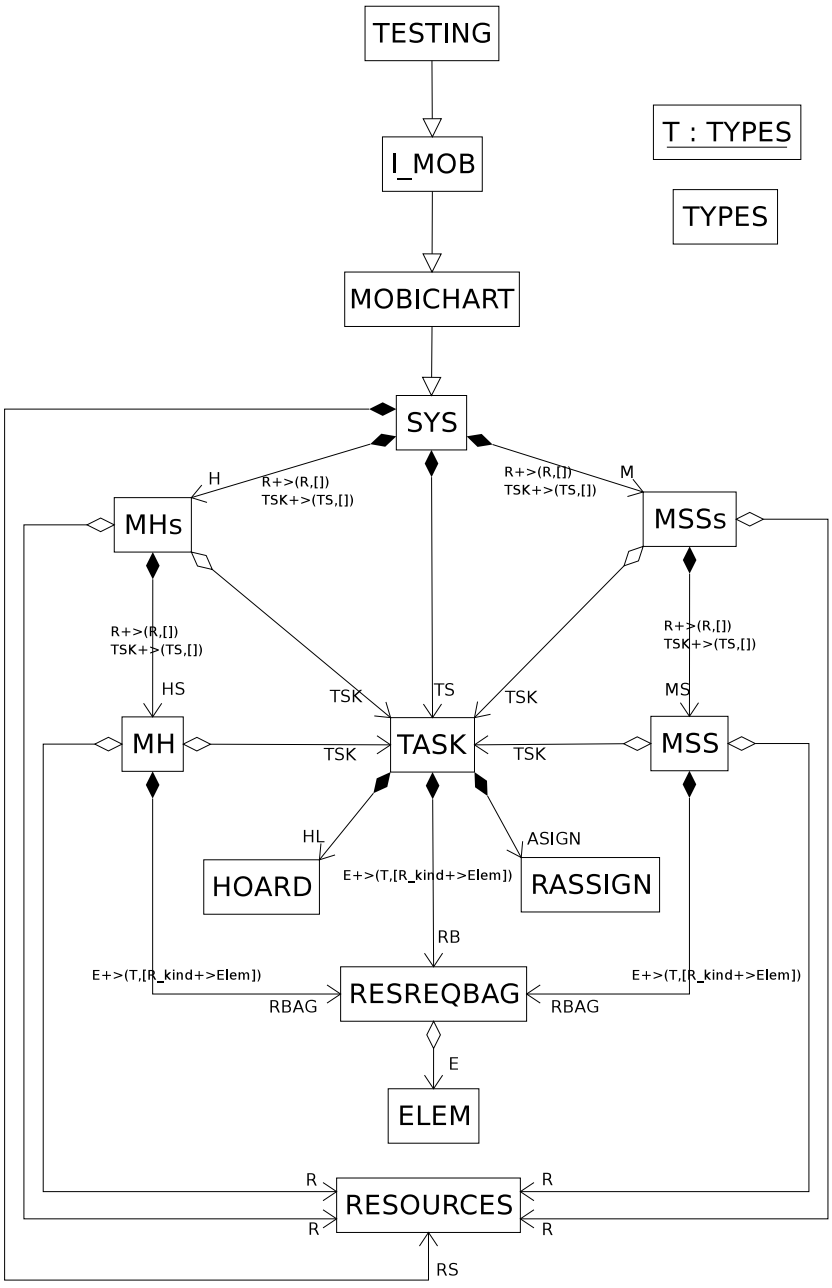

Figure 5.12: Scheme Diagram of the mobile infrastructure specification.

```
scheme D = class ... end,
scheme E = class ... end,
scheme F = class ... end,
scheme G(f2 : F) = class ... end,
scheme C = class
    object d2 : D
    ...
end,
scheme A(f1 : F, e : E) = class
    object d1 : D
    ...
end,
scheme B(e : E, f1 : F, ge : G(f1), cp : C) =
    extend A(f1, e) with class
    object gp : G(f1), ce : C
    ...
end,
scheme H(cp : C, ge : G(f1), f1 : F, e : E) =
    extend B(e, f1, ge, cp) with class
    ...
end
```

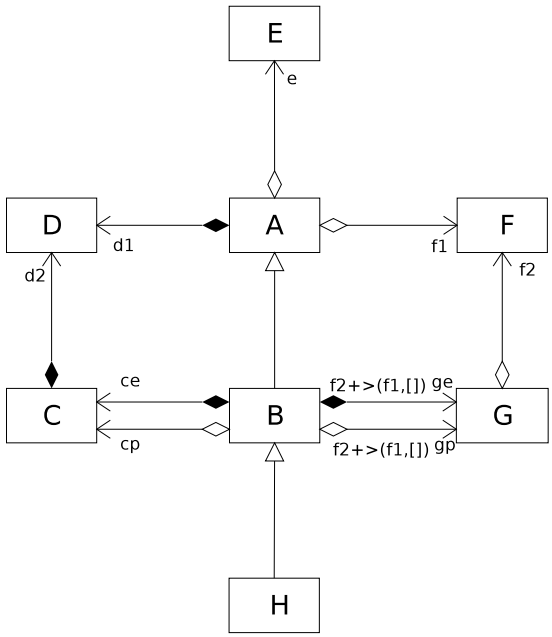

Figure 5.13: Combination of associations of kind Parameter and Nested with the extend relation.

```
scheme ELEM = class ... end
scheme RESOURCES = with T in class ... end

scheme TYPES = class ... end
object T : TYPES
```

5.6.2 Constructed example

The Scheme Diagram presented in figure 5.13 is a constructed example. The main purpose is to show scheme instantiation in conjunction with the association and extend relations. The usage of formal parameters are particular important to show since they are implicitly inherited. This is not the case for a normal RSL. The RSL text presented together with figure 5.13 is actually auto generated by the tool presented in part III after drawing the diagram in the tool.

The nested associations in the diagram have a clear and simple mapping to RSL. Their presence are only visible in the schemes with the declarations, and are available for a client through the extend operator in RSL. This is the case for all declarations in a class expression.

As described in section 5.5.5 there is no ordering of the formal parameters in the diagram. As a consequence the type `ActualParameters` is introduced which maps the actual parameters to the formal. Hence the ordering is not needed. When using the **extend** operator on parameterised schemes in RSL, scheme instantiation must take place. Consider scheme `A` which has two formal parameter. It is extended by scheme `B` which inherits the two schemes and another two formal parameters are added. Notice that the translated ordering actually differs. Remember that the RSL is auto generated. The ordering is only used for pairing the actual and formal parameters. Therefore the translation is correct and will syntax check. Another interesting situation is the formal parmater `gp` of `B` which itself has a formal parameter. In this case an actual parameter must be provided to the scheme instantiation.

The scheme `H` is also noticable. Although it visually only participates as a client in one relation, it actually has four formal parameters and three nested objects.

5.7 Translation: $\mathbf{SD}_\delta \rightarrow \mathbf{RSL}_\alpha$

The Scheme Diagram is made for RSL and represents a subset of RSL. Most of the constructs in the diagram have a direct mapping to RSL; and vice versa. The complete formal specification describing the transformational semantics of the Scheme Diagram is presented in appendix C.2. That is, the semantics of the diagram are described as a mapping to RSL which already has a well defined semantics. The abstract RSL syntax described in section 4 is used as target for the translation. In this section the less obvious aspects of the translation specification are commented.

A goal of the translation is that the result will pass the syntax check by the RSL type checker *rsltc*. In the Scheme Diagram there is enough information to achieve this, in particular because type expressions are included. The type expressions make it possible to add signatures for types, values, variables and channels. Axioms do not have a signature and are basically a value expression of type boolean with an optional name. In order to translate axioms into RSL it is necessary to add value expressions, which in the translation are chosen to be **true**. (*transltr_AxiomDecl*, C.2)

The *rsltc* tool requires that each module is placed in its own file with the same name as the module and the *.rsl* extension. It is also required that the context is specified, that is, dependencies to other modules. Determining these dependencies in the Scheme Diagram is fairly easy since all modules are names and dependencies between them are the relations present in the model. Thus the context for a given scheme is the set of supplier names for all relations in which the client is the given scheme. (*transltr_context*, C.2)

As mentioned in previous sections the order in which the formal parameters are written in a RSL specification is important in conjunction with scheme instantiation. Since there is no ordering of the formal parameters in the Scheme Diagram it is necessary to create one during translation. In general in the diagram a module dependency is always explicitly stated in the form of an association or extend relation. Since module dependency must not be recursive the structure of module dependency is either a tree or a forest of trees. The problem of the ordering of parameters is solved in the order in which the modules are translated: Schemes are translated before objects. A scheme must first be translated if it either has no dependencies or all of its dependencies already have been translated. Hence the leafs of the tree(s) are translated first. Consequently when translating a scheme instantiation, the order of the parameters have already been decided. (*leaf_scheme, next_scheme*, C.2)

The extend construct in RSL is an operator on two class expressions giving one class expression being the combination of the two. Formal parameters are part of a scheme declaration and not class expressions, hence they are not included in the extension like object declarations. In the UML Class Diagram all associations are inherited. Although this is an object oriented characteristic it is still practical in the diagram since it reduces the number of relations needed to be drawn in the diagram. This was discussed in section 5.5.6. When extending a parameterised scheme in RSL it is necessary to specify the actual parameters for that scheme. All formal parameters are inherited with the same name and class expression in the Scheme Diagram, thus there exists an actual parameter with the same name as the formal. (*transltr_Extend*, C.2)

Parts of the Scheme Diagram are not translatable and are simply ignored. The static implementation relation is meta information about two schemes. Loosely speaking it is a factual statement about the two schemes. Associations of kind Global indicate usage of a global declared object and in principle it is not translated. It is used to determine the context which is required by the *rsltc* tool.

The state of an object is not translatable either since it shows the state at some moment in time. It does not correspond to the initial value of a variable, since this should be included in the scheme.

5.8 Future work

The introduction of fitting required the renaming of class expressions. Thus part of the renaming functionality is already implemented. Object fitting is introduced since it adds considerable value to the use of modules and parameterised schemes. Without fitting generic parameterised schemes would be difficult to develop. Renaming would not contribute similar value to the diagram if added, its usage is less obvious. It should, however, still be considered for addition to the diagram. Since the renaming is an operator on class expressions it should be available in conjunction with the extend relation and with the scheme.

In the Scheme Diagram static implementation is a relation between two schemes. An interesting development would be to make it a relation between two diagrams. It could thus be used to automatically verify that one diagram is a correct development of a former.

It is possible to display the state of an object. It could be considered to expand this notion to show the development of the state. This could be done by having several boxes in the diagram representing the same object. This is currently not possible since names of objects must be unique. Furthermore it would cause redundant information since any given actual parameters must be the same if the object names are the same.

Section 5.7 presents the translation from the Scheme Diagram to RSL. In order for the Scheme Diagram to be truly useful it must be possible to translate existing RSL texts to the diagram. Since only a subset of RSL is included in the diagram some consideration must be given to the reverse translation, e.g. value expressions could simply be ignored. More interesting is if requirement that all class expressions must be named in the Scheme Diagramcan be fulfilled. Unnamed class expressions can in fact be rewritten to schemes during translation without altering the meaning of the specification.

There are a few details that should be corrected. The use of *Name* in the Scheme Diagram does not have the same meaning as in RSL. In RSL its is either an id or an operator, in the Scheme Diagram it only represents an id. The rolename of the *Association* type should be part of the *Parameter* and *Nested* kind, like *ActualParameters*. It would simplify well-formedness and give a clearer understanding when reading the specification. Hiding in the Scheme Diagram only uses the name of the declarations. If overloading is used, then it is necessary to add the type expression.

In the specification of the Scheme Diagram there are the following known issues:

- In (*wf_typename_expr*, C.1.3) only public type declarations are considered. If the qualification of the type name is empty then the private type declarations of the initial scheme must also be included.
- When determining the available association of a given scheme the visibility is not considered. (*associations*, C.1.2)
- Static implementation and signature morphing does not support qualification. Thus a type name that includes qualification will always be its own maximal type. (*gen_sig_map*, C.1.5)
- Signature morphing is only applied to value and variable declarations. It should also be applied to type and channel delcarations. (*static_implement_types and static_implement_channels*, C.1.5)
- Multiplicity is missing in global objects. Additionally it is only possible to specify one binding and not a list of bindings.
- It must be checked that the specified predicate in a subtype expression has the supertype as parameter and a boolean return type. (*wf_subtype_expr*, C.1.3)

5.9 Conclusion

The new Scheme Diagram has been presented in this chapter. The purpose of the diagram is to present an overview of a RSL specification with focus on the structure and relationship between the modules. For this purpose the selected subset, which the diagram represents, seems well balanced. This is apparent since a well-formed Scheme Diagram always will pass the *rsltc* type checker after translation to RSL.

The examples in section 5.6 do reveal situations where the readability of the diagram is reduced. In particular when a global object is used for common types by all other modules. This results in a diagram with a growing number of lines which consequently may cross each other. Another situation which gives an unwanted number of lines is the use of parameterised schemes, where the formal parameters themselves are parameterised, etc.

The schemes of the Scheme Diagram allow for the signature of the declaration to be included. The size of the boxes do however tend to grow rapidly. The is partly due to the long signature that applicative specifications produce, but also by the number of declarations. This is also the reason why it is not required that the modules show their content in the diagram but only their name. Hence the focus is kept on the relationships.

Despite the fact that the Scheme Diagram is not ideal in all situations and that RSL is not object oriented, we are of the opinion that the diagram is a positive addition to RSL. In particular because it emphasises the use of modules.

Chapter 6

Live Sequence Charts

Contents

6.1 Before we start . . . 54
6.1.1 Introductory example . . . 54
6.1.2 Why LSCs? . . . 54
6.1.3 Structure of Chapter . . . 55
6.2 Structured narrative of LSC . . . 56
6.2.1 Background . . . 56
6.2.2 Events . . . 57
6.2.3 Timing . . . 58
6.2.4 Cuts/States . . . 59
6.3 Previous work . . . 60
6.3.1 A bevy of LSC related papers . . . 60
6.3.2 Summary of preliminary thesis . . . 65
6.4 The LSC subset chosen: RSC . . . 66
6.4.1 Collections . . . 66
6.4.2 Charts . . . 66
6.4.3 Instances/Locations . . . 66
6.4.4 Subcharts . . . 67
6.4.5 Events . . . 67
6.4.6 Timing . . . 69
6.5 Formal description of RSC . . . 70
6.5.1 Types . . . 70
6.5.2 Well-formedness conditions . . . 71
6.5.3 Semantics for one chart . . . 75
6.6 Example: RSC RSL specification . . . 78
6.7 Translation: $\mathbf{RSC}_\delta \rightarrow \mathbf{RSL}_\alpha$ . . . 82
6.7.1 An applicative RSL model of RSCs . . . 82
6.7.2 RSL CSP and LSCs . . . 85
6.7.3 RSL CSP approach . . . 86
6.7.4 A pure CSP approach . . . 87
6.8 Example: Applicative RSC . . . 89
6.8.1 RSCs . . . 89
6.8.2 Specification . . . 89

6.8.3 Complete System 91
6.9 Future work 92
6.10 Conclusion 92

6.1 Before we start

6.1.1 Introductory example

Since Live Sequence Charts is a graphical notation, we start with a more thorough example than given in the appetiser in the introductory part. It should give the reader a better idea of how LSCs look like and work.

Figure 6.1 is an example about how a PC might be started. It has a universal mainchart, recognised by the fully drawn borderline on the lower box. A universal mainchart means that the behaviour described by this chart is always mandatory for the system being modelled. In contrast, a dotted borderline on the lower chart represents an existential chart, denoting behaviour that minimum once must be exhibited by the system. This resembles MSCs (Message Sequence Charts) whose weak semantics are not clearly defined in terms of what a MSC actually specifies, e.g. [4].

The top hexagon is a prechart, a precondition which must be fulfilled before the behaviour in the underlying mainchart must be exhibited. The precondition here is that the instance *User* sends a message to *Controller* named *Push* with the parameter *on*. Thus for example describing the push of a button. *User* and *Controller* are instances which may denote specific objects or processes, depending on the system treated. Instances communicate via messages which have names and optionally arguments.

The next message is sent from the *Controller* instance to the *PC* instance. The *PC* then performs a local action called *Bootstrap*, indicated by the box. It is further unspecified. The next hexagon is a condition, stating "CPUtemp > 50". This denotes that the condition must be fulfilled in order to proceed beyond it.

It is then specified that the *Controller* sends a message *Beep* and *Throttledown*. The dots along the *Controller* instance denote a coregion, meaning that the order in which those two events happen is non-deterministic.

The upper part of the vertical lines of the instances are fully drawn. This means that the progress along the instances is mandatory, the instances **must** proceed. The lower part is dotted, meaning optional behaviour. It includes a subchart that specifies that the *Controller* **may** choose to emit up to 3 more beeps depending on the condition. The iteration of up to 3 times is denoted by the number in the upper left corner.

6.1.2 Why LSCs?

LSCs is a quite new scenario-based graphical requirements notation first proposed in [7] to extend MSCs [47, 34]. The paper identifies several weaknesses in MSC which is a widely used notation to capture requirements for parallel systems. It is industry accepted and is much in use in the telecommunications industry for example, and the graphical and textual syntax has been standardised by ITU-T [34]. A variant of MSCs is also known as Sequence Diagrams in UML [17]. LSCs introduce a new feature of hot and cold elements which distinguish mandatory and optional behaviour, thus adding liveness [2].

LSCs have been chosen as an interesting graphical notation with some recent research carried out regarding the syntax and semantics, see section 6.3. It was interesting to see if LSCs could be successfully formalised for the use with, and incorporated in the RSL-tool-set. This should be done in order to obtain the possibility of specifying inter-object behaviour graphically and then translate it to RSL.

The focus has been on the integration with RSL rather than the formalisation of LSCs as in [41], which gave rise to several limitations compared to the original LSCs. The initial idea was that the parallel nature of LSCs was obvious to model using the CSP [30] constructs in RSL.

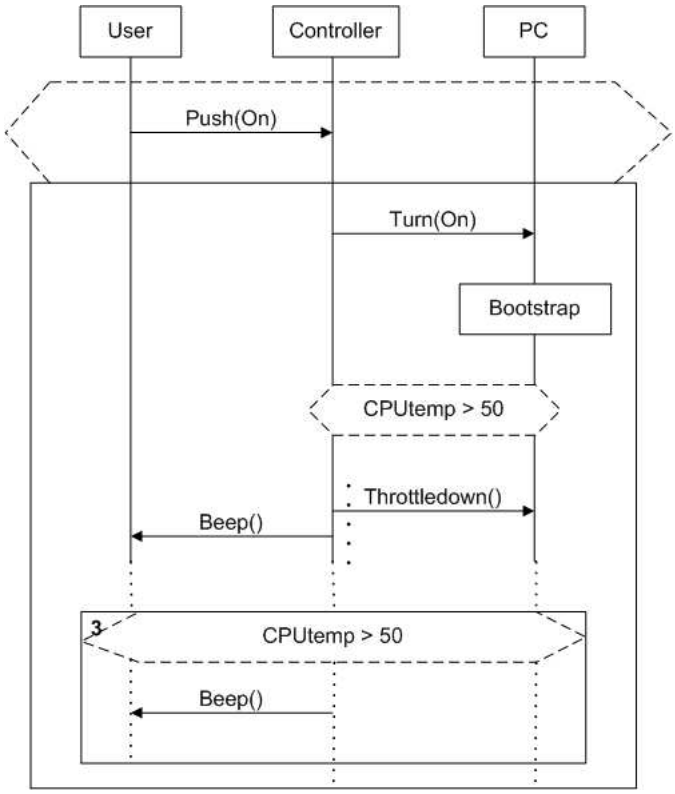

Figure 6.1: Example of a LSC with various constructs.

6.1.3 Structure of Chapter

LSCs are introduced in a structured narrative in section 6.2. LSCs are described with their syntax and semantics in detail. The use of LSCs is also explained.

This is followed by section 6.3 about the previous work that has been used for this part of the thesis. The findings that were relevant are presented in order to give a background on the choices that were made.

Section 6.4 describes the subset of LSC constructs that were chosen and the reasons for these choices. This is also presented in context of the previous work that is described.

Then a formal description of our version of LSCs is presented in RSL in section 6.5. It is called RSCs for **RSL Sequence Charts**. The syntax with well-formedness conditions is given. After that trace semantics for one chart is presented.

Then an example of RSC is given in section 6.6. Section 6.7 presents the translation of RSCs to an equivalent RSL specification, the goal of this work. This is followed by an example of the applicative use in section 6.8.

Finally a section about the future work is given and the results will be concluded and discussed.

6.2 Structured narrative of LSC

This chapter introduces the syntax and semantics of LSCs informally. The basic form of LSCs is presented with [7] as basis. It was the initial paper on LSCs. The aim is to build an understanding of LSCs and their constructs in order to prepare for the next, more technical sections. This section is partly based on the preliminary thesis [2]. If the reader is acquainted with LSCs this section may be skipped.

6.2.1 Background

MSCs are used in the early stages of requirements engineering. They can for example be seen as a manifestation of use cases in UML. They thus represent an inherent informal existential view of a behavioural model of a system. They are convenient for describing sample scenarios in an inter-object oriented fashion.

In the design phase there is a need for a more rigorous approach in order to fully specify what a system should do. For this, often state-machine languages are used [7], like state-charts [22, 23]. They fully specify the behaviour of a system in an intra-object fashion and are usually executable.

The ultimate goal of the authors of [7] is to investigate the two-way relationship between these two views of behavioural description. As a solution they propose LSCs with the goal of enough semantical rigour to bridge the gab. LSCs are suitable for the later stages of requirements when a universal view of the system is required. The idea is to capture enough information about a system using LSCs in order to allow linking between the descriptive view of LSCs and the constructive view of state-machine languages. LSCs as such can be used to specify constraints on the partial order of events that take place in a system.

In [26] this idea is taken even further. A new method of development is proposed where executable LSCs directly lead to the implementation. Thus effectively using the requirements to directly specify the system, omitting the design phase. More on this in section 6.3.

The prechart and mainchart of a LSC are syntactically similar. They are charts that consist of one or more instances that are drawn as vertical lines as presented in figure 6.1. They may be models of processes or objects and semantically consist of a list of locations. Locations have a temperature: hot or cold. A location has an event attached and if the location is hot it denotes that the event must be performed. If it is cold, it may be performed. A restriction is that after a cold location in a chart all locations are cold. Locations are strictly ordered in time along the instance on which they are defined.

In order to enhance the expressiveness, LSCs include the ability to specify subcharts as shown in figure 6.2. Iteration is possible since subcharts have a multiplicity. This enables a subchart to specify that the enclosed behaviour must be exhibited several times. This feature is not present in MSCs. Cold locations which are the optional last locations in a subchart may be followed by hot locations in the super-chart, as shown in figure 6.2.

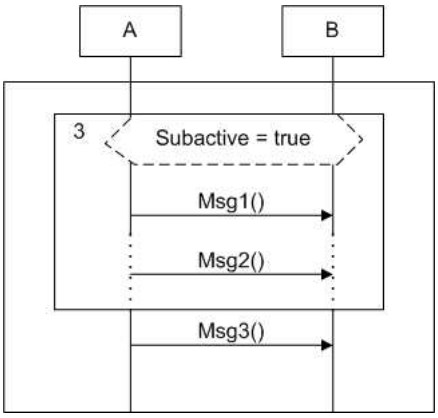

Figure 6.2: Example of a LSC with subchart and hot/cold locations: First the predicate is evaluated. If it is true Msg1 ***must*** *be sent. After that Msg2* ***may*** *be sent, as it is specified on cold locations (as denoted by the dotted lines). This is repeated 2 more times if the predicate still holds, due to the multiplicity of 3. Iteration continues even if Msg2 is not sent. Finally Msg3 must be performed.*

A style that resembles imperative programming is also possible by constructs that give an while-do loop. This can be done by using a subchart using a cold condition, see figure 6.3. The subchart is performed. If the condition is true, the subchart is repeated. This is repeated until the condition evaluates to false due to the unbounded multiplicity. Also if-then-else (see figure 6.3) and for-do and similar loops are possible. This makes LSCs highly expressive with regards to flow of control and may ease the use of them since the above features are highly intuitive.

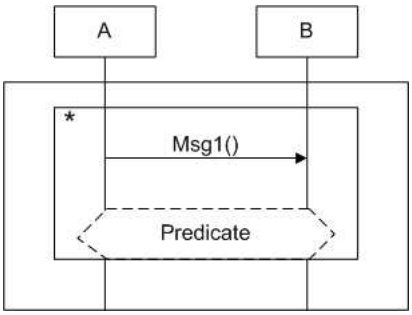

Figure 6.3: Example of a do-while construct using a subchart with a predicate and a multiplicity of asterisk (*), denoting infinite repetition.

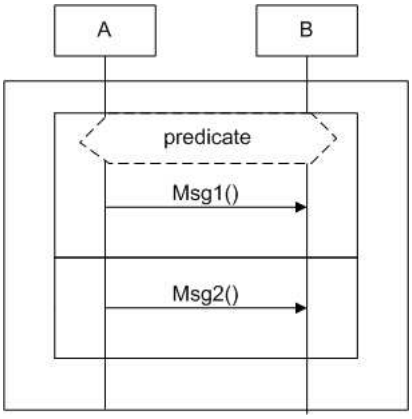

Figure 6.4: Example of an if-then-else construct using a specialised subchart and a predicate. If the predicate is true, the first part of the subchart is performed, if not, the lower part. The two subcharts must be adjacent. If there is a gap, they denote two separate subcharts.

A finite number of instances may be created by other instances as shown in figure 6.5.

To each LSC there is attached a set of visible events and visible variables. The visible events are the events that the LSC constrains. All the events that are shown in a mainchart are visible. In a separate box it is possible to show events that are visible but may not be performed. They are called forbidden events. Visible variables are the set of variables a mainchart uses. They can also contain variables that are not used and are not allowed to be changed by other charts while the LSC is active. They are not displayed graphically.

6.2.2 Events

Events are the basic elements of LSCs. They denote the actual events that happen in the system. As such they may also be shared among several LSCs. This is a basic concept: if a LSC event is present in two separate LSCs it denotes the same actual event. This automatically links the described behaviour of LSCs together.

Messages/Coregions

Messages are one of the central elements of LSCs as they specify how instances communicate. They are hot or cold: Hot messages are messages that must be sent and must be received by the intended receiver in order

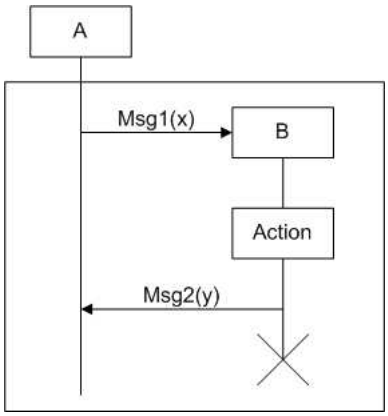

Figure 6.5: Example where instance A creates instance B. This is denoted by a message from A to the instance box of B. B does only exist while this mainchart is active. Therefore it is annotated with a cross at the end of the time line, denoting destruction.

to satisfy a LSC. A cold message must only be sent, there is no requirement for reception. Hot messages are denoted with solid lines, cold messages with a dotted line.

Messages are synchronous or asynchronous, meaning that a synchronous message must be received immediately by the receiver. A asynchronous message may be delayed an unspecified amount of time. Synchronous messages are denoted with solid arrowheads, asynchronous messages are denoted with hollow arrowheads. Messages have a name and possibly some parameters [26].

A special event called coregion is introduced in order to specify some message inputs/outputs that may happen in an arbitrary order. The coregion is strictly for message events only. An example of a coregion can be found in figure 6.1 where it is denoted with dots along the vertical line of the instance named Controller.

Conditions

Conditions as shown in figure 6.1 are also hot (denoted with solid line hexagon) or cold (denoted with dotted line hexagon) with very different semantical meaning. Conditions are evaluated on the basis of visible variables. If a condition is true, the remainder of the chart may proceed.

If a false cold condition is encountered the complete chart exits successfully, meaning that the system fulfils the chart. If the cold condition is located in a subchart, only the innermost subchart the condition is located in is exited, continuing right after it. Semantically this resembles a *goto* statement.

If a false hot condition is encountered, this means there is an inconsistency in the model as a false hot condition is never allowed. It is an assertion of a given statement.

Actions

Action labels are supplied in order to denote local computations that may alter arbitrary locally visible variables on an instance to arbitrary values. An example of an action, "Bootstrap", may be seen in figure 6.1.

6.2.3 Timing

Time is mentioned shortly in [7] with local timing constructs. There is later work on time in [25] which introduces a global clock and ticks. As there is no general accepted notation or semantics for timing constructs, it is not presented here. See section 6.3.

6.2.4 Cuts/States

As LSCs are meant to be executable in [26] the notion of **cut** is introduced. A cut denotes a set of positions of instances in a chart. See figure 6.6. A cut corresponds to a program counter, identifying where to a LSC has progressed. In order to capture all the information for execution, an actual **state** for a chart is needed, see figures 6.6, 6.7 and 6.8. The state records information about the program counter and about the state of coregions. Since coregions are considered one event comprising of several message events it is necessary to record which events in a coregion have been performed and which have not.

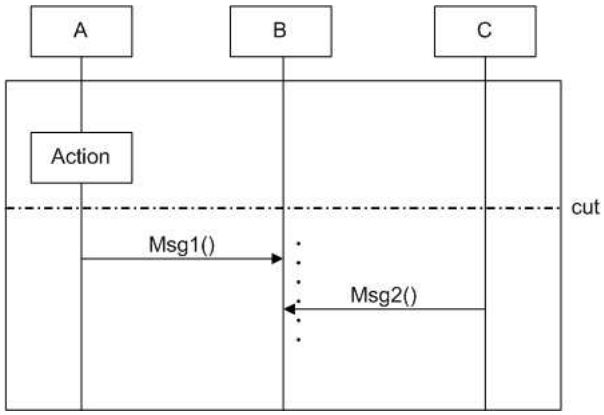

Figure 6.6: Example of a cut: Instance A has performed the local action and can send Msg1. Instance C can send Msg2. Instance B can either receive Msg 1 or Msg2 due to the coregion. The cut is denoted by the dotted line.

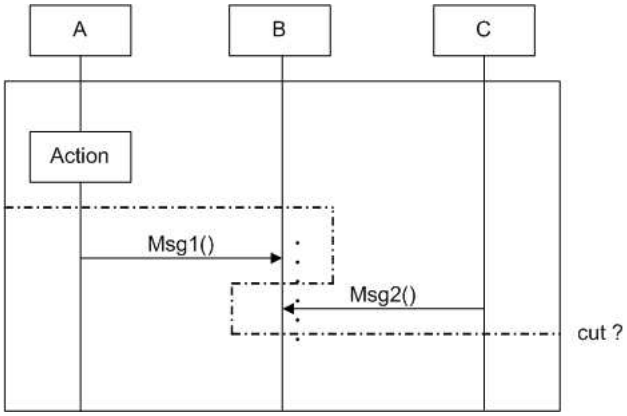

Figure 6.7: Msg2 has been sent. This new information would yield a cut like the dotted line. This is however not quite correct, as the cut intersects instance B several times. B is semantically still at the beginning of the coregion location, as the complete coregion event has not been performed.

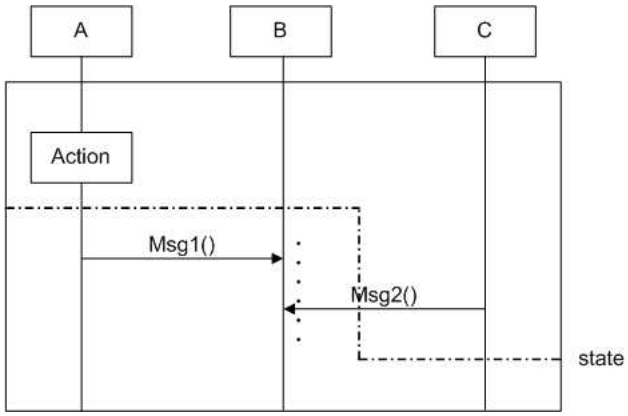

Figure 6.8: The correct interpretation is that instance B is still at the coregion beginning and C has sent Msg2. This cannot be displayed graphically. But semantically it is done by including auxiliary information about the state of B, namely that Msg2 already has been received. This only leaves the possibility of receiving Msg1. The resulting information is called a state instead of a cut.

6.3 Previous work

In the following, previous research used in this thesis is presented. The first section is about previous research papers regarding LSCs and other useful papers. The second shortly introduces the work and results of our preliminary thesis with regards to LSCs.

6.3.1 A bevy of LSC related papers

This section introduces the papers that set the stage for our interpretation of LSCs. Main aspects are shortly introduced and limitations/extensions are commented on. The following papers are emphasised since they have been studied extensively.

The section is also included since some of these papers are heavily referenced. Especially when justifying the choices made regarding the subset of LSC that will be dealt with. Thus, it is not necessary to read the papers themselves. The references are ordered chronologically.

[7] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.

> The initial proposal of LSCs. It introduces LSCs as an extension to MSCs in order to cope with the weaknesses of the latter. It uses program-like pseudo-code to introduce the semantics. Furthermore it uses skeleton-automata (state transition system) to explain the intra workings of LSCs, for example between precharts and maincharts. Only a sketch of a possible formalisation has been provided.
>
> The main goal by Damm and Harel is to introduce liveness to a notation like MSCs, recognising that there are fundamental unanswered questions regarding MSCs. For example: do they describe all behaviours of a system? Or maybe just some behaviour?
>
> To solve this issue, hot and cold elements are introduced, which denote mandatory and optional behaviour respectively. Furthermore the notion of subcharts is introduced which creates interesting possibilities of creating iterative structures e.g. do-while, see section 6.2. It also introduces the concept of forbidden scenarios which describe behaviour that a system may not exhibit. The paper mainly focuses on the key aspects of the approach. Therefore it omits specific issues regarding instance creation/destruction and the use of timers. With regards to MSCs it is also noted that there exist several problematic semantic issues, e.g. what happens if a condition evaluates to false?
>
> Instances introduce a weak partial ordering on events. The partial ordering stems from the coregions that do not impose an ordering on message events, thus introducing non-determinism.

Besides the ordering on instances there is no ordering between instances other than the ordering introduced by messages, conditions etc. Effectively parallelism is introduced. However the system is treated as being discrete in order to ease the semantics. We follow that approach.

Timers are briefly shown with a notation taken from state-charts.

[24] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *CIAA '00: Revised Papers from the 5th International Conference on Implementation and Application of Automata*, pages 1–33. Springer-Verlag, 2001.

The paper is a follow-up on [7]. It is noted that Sequence Diagrams are the manifestation of use cases. If these could be synthesised these could lead directly to implementation via state based object systems. This means that there is a need to go from the inter-object oriented approach in the requirements phase to the intra-object state machine approach in the design phase. The idea is to synthesise a collection of LSCs as they convey enough information about the system in contrast to Sequence Diagrams. It is defined that a specification using LSCs is consistent iff it is satisfiable by a state based object system (finite state machines or state charts).

It is noted that if-then-else constructs in [7] are defined with two separate subcharts. This is altered in order to ensure that a condition is only evaluated once, as depending variables may change between two evaluations of the condition. This is also noted later in our approach.

Several constraints are imposed on the approach. Firstly only universal charts are used since existential charts do not convey enough information about the model in order to allow synthesis. Existential charts are seen as informal sketches early in the development phase which may be refined to universal charts later.

Subcharts are only allowed if they cover all instances, thus effectively enforcing a synchronisation among all instances. This allows for easy recursion when synthesising a state based system, as it can be done using a transition from a state to itself, where a state may consist of sub-states.

The paper only treats synchronous messages, but gives no arguments why. A guess is the idea is to use StateMate, which is a validation/generation tool for Statecharts from the same company D. Harel is affiliated with [32]. StateMate can only handle synchronous communication.

Also it basically only sketches a solution for a fundamental problem, namely how to avoid built-in contradictions in universal charts. As discussed in [45] distributed reactive systems are in general hard to synthesise. It is also assumed that the activation of the same chart cannot overlap, which is a rather restrictive approach. This for example severely limits the possibilities of handling several identical requests in parallel. Nothing is said about timing.

[36] Jochen Klose and Hartmut Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–527. Springer-Verlag, 2001.

The paper tries to address the issue of the only sketched formalisation of LSCs. A Timed Büchi Automata is derived from the LSC via unwinding the structure of it for the sake of model-checking via an extension to already existing verification tools. They also introduce a form of timing annotations which sets lower and upper bounds on events. Time has been treated as discrete as they base their time model on a system exhibiting time-discrete behaviour.

They do not follow the syntax for annotations introduced by [7], since they simply use it as a guard on the resulting automata. A notion of simultaneous regions is introduced in order to allow parallel events in contrast to the discrete semantics of LSCs. This is the only place where timing annotations may occur, effectively limiting the usefulness of these. Furthermore only synchronous messages are considered, since StateMate is used to check the automata. A major limitation is also that only a single LSCs is treated, so no formalisation of the inter workings between several charts is given. This is a severe constraint, as this leaves open the fundamental aspect of stating several requirements in several charts, a point also made in [24].

[6] Yves Bontemps and Patrick Heymans. Turning High-Level Live Sequence Charts into Automata. Technical report, Univ. of Namur - Computer Science Dept, March 2002. `http://www.info.fundp.ac.be/~ybo`.

> The paper focuses on creating an equivalent of High-level MSC [42] using LSCs in order to create a more formally defined interaction between several charts. However it limits the LSC to using conditions and messages only.
>
> They extend LSCs with compositional operators and define their semantics in terms of regular-language traces. From these they build corresponding Büchi Automata (as in [36]) which accept the traces of the language expressed by LSCs. By using appropriate existing algorithms the resulting automata are checked for consistency, refinement etc. As the other approaches using automata, the state-space explosion problem, due to the complexity of LSCs by synthesising automata has not been addressed.
>
> They clarify that messages not shown on LSCs are abstracted away but not disallowed, which also is the case in our later specification in RSL. Furthermore they introduce invariants which may (or must) hold during two points in time. The latter two additions have not been addressed by us. Furthermore the article does not cover time specifically on LSCs as only conditions and messages where allowed.

[25] David Harel and Rami Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, page 193. IEEE Computer Society, 2002.

> The paper presents an extension to LSCs with timing constructs in order to specify behavioural requirements of time-intensive systems, acknowledging that reactive systems often must react and refer to time. Time is primarily modelled by extending LSCs with a single clock object. For explicit timing demands a "Tick" has been introduced, which relates to an external clock object, i.e. enabling LSCs to be activated after e.g. 50 "ticks". The synchrony hypothesis is assumed, meaning that the actual events do not take time but time passes between events. This seems as a reasonable abstraction in the context.
>
> Generally the addition of time is very complex and in this case does not at all adhere to the initial syntax of [7], where no semantics where presented. The annotations are quite complex and allow for some structures that are not possible with our chosen subset of LSCs.

[44] Rami Marelly, David Harel, and Hillel Kugler. Specifying and executing requirements: the play-in/play-out approach. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 84–85. ACM Press, 2002. Previously: Technical Report MCS01-15, Mathematics & Computer Science, Weizmann Institute Of Sience.

> The paper is about a quite alternative approach to using LSCs. The main idea is to use a novel approach called play-in/play-out in order to specify a system. The corresponding tool, called play-engine, is only described on a high level. For further explanation, see [43, 26].

[43] Rami Marelly, David Harel, and Hillel Kugler. Multiple instances and symbolic variables in executable sequence charts. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–100. ACM Press, 2002.

> This paper was presented at the same conference as [44] and delves into the actual semantics of the executable semantics of LSCs. Only an overview and the main principles of the execution mechanism are presented. It is noted that the entry of a subchart actually is a synchronisation barrier, not allowing an instance to proceed until all the participating instances have reached the subchart. This applies to conditions as well, since the condition must only be evaluated once.
>
> It is clarified that events not occuring in a chart may happen without constraining/violating the chart. Symbolic execution is described as being possible with a slight modification to the partial

order on instances. Furthermore an extension regarding the instances is introduced: an instance may denote a group of objects rather than one concrete object. Consistency checks are not integrated in the tool, thus LSCs might lead to violations under execution.

[26] David Harel and Rami Marelly. *Come, Let's Play - Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag Berlin Heidelberg, 2003.

The book may be seen as a compilation of the previous papers of the authors about LSCs and the "play-in/play-out" approach. Play-in is done via letting users show the desired behaviour of a system via a mock GUI. The behaviour is then captured via LSCs while "playing". The recorded LSCs are then used to "play-out" the behaviour, i.e. interacting with the user based on the requirements given via the previously recorded LSCs. This does not equal to a simple replay, but an interaction based on the constraints imposed by the LSCs.

By recording enough LSCs it is suggested that this approach could be used to build an entire system (though simple ones), completely skipping the design phase [24] and going directly from requirements to implementation. It is noted that this approach is much more natural for users allowing them directly to specify how a system should react. Timing annotations are taken from [25] and the synchrony hypothesis is therefore assumed.

As the book states itself it does not dwell on the details of the language constructs, the methodology nor the tool. This of course makes it somewhat difficult to asses the choices made. Especially the absence of rigorously defined semantics makes it hard to make use of it. Example: A very interesting note on messages states that there are semantic problems with temperatures on locations and messages. They allow message sending and reception locations to be of different temperatures, thus allowing up to $2^3 = 8$ different temperature combinations since messages themselves have a temperature. Several of these are contradictory, see example in figure 6.13.

The play-engine, as the implemented tool is called, enforces an maximal policy, that is that everything that can be performed, will be performed. As it is executable it has to perform some choices, which the semantics allow. In relation to cold locations this policy is applied by performing the events on these locations if they do not violate hot events, called hot cuts. This might be a reasonable assumption, however it is not documented very well. Furthermore the issue of multiple activation of the same chart as done in [43] has not been addressed.

The book also treats other issues, e.g. smart play-out, which is an addition to the play-out approach. This is however not interesting for our work as we do not cover these issues.

[42] Christian Krog Madsen. Study of Graphical and Temporal Specification Techniques. Technical report, DTU, Nov 2003.

The paper is a preliminary thesis [41]. Three types of diagrams, namely Petri Nets, Message Sequence Charts and State Charts are described using structured English, examples and formalisation in RSL. Especially the MSC part is interesting as an example is given where it is modelled using CSP in RSL. However it must be noted that throughout the example only simple message sending is included, no conditions. So the modelling amounts to quite trivial CSP constructs.

[41] Christian Krog Madsen. Integration of Specification Techniques. Master's thesis, DTU, 2003.

The thesis describes the graphical notation of State Charts and LSCs. A process algebra is defined for a subset of LSCs and for expressing communication behaviours of RSL specifications. The algebra is then used to give an algebraic semantics to the subset of LSC. The main drawback is that only message (including coregion) and condition events are included.

The subset of LSC is then related to a subset of RSL noting that there are problematic issues with LSC which need to be addressed. Only synchronous messages are considered, as RSLs CSP only allows this kind of messages. An appendix is given which lists some of the problems with LSCs as they are proposed in the previous papers.

[51] Tao Wang, Abhik Roychoudhury, Roland H. C. Yap, and S. C. Choudhary. Symbolic execution of behavioral requirements. In *PADL*, pages 178–192, 2004.

The paper describes an approach to add real symbolic execution of LSCs using Constraint Logic Programming (CLP). This enables the execution of several LSCs, which only differ in values, as one. Furthermore it gives the possibility to simulate scenarios with an unbounded number of processes.

It is noted that [43] does **not** allow symbolic execution, even though the charts are depicted with variables, as these are instantiated during execution, thus restricting the potential behaviour of a LSC. This is also undesired since the charts are instantiated with all possible values, possibly resulting in a large number of charts.

Time is also handled differently than in [25]. As described previously the Play-Engine introduces a global clock which ticks (automatically or user invoked in simulation), thus making the time annotations tests as whether they hold or not. In the paper they treat the annotations as constraints and constrain the time to an allowed interval, see figure 6.9. Only universal charts are considered.

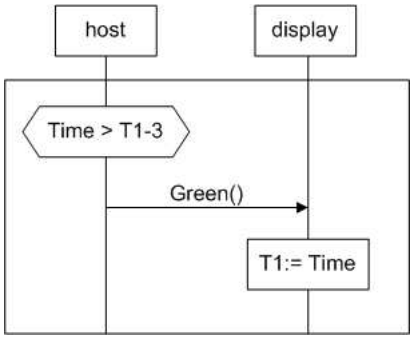

Figure 6.9: Example from [51] where the play-engine will fail: When the variable T1 is encountered it must be instantiated, which is not the case. The approach described in [51] adds this as a constraint. When the message then has been sent and the assignment is encountered, it can be checked that the constraint is satisfied.

[49] Jun Sun and Jin Song Dong. Model checking live sequence charts. not published, 2004. Paper written in conjunction with Ph.D. Thesis at School of Computing, National University of Singapore, (sunj, dongjs)@comp.nus.edu.sg.

The paper tries to exploit the close semantic resemblance between LSCs and CSPs traces and failures. The idea is to transform the former to the latter in order exploit the available tool-support for the latter. Inconsistency (see figure **??**) can be checked for by using FDR[1]. The approach is based on the skeleton-automata in [7]. A separate RUN process is included in order to specify that not visible events are not constrained.

The approach also limits messages to synchronous messages. Furthermore conditions are restricted to one instance only. This could potentially lead to different results if two instances are to evaluate the same condition, as evaluation times may differ. Furthermore all locations are cold, as CSP only specifies prefix-close languages. However solutions to some of these shortcomings are sketched.

[40] Zhiming Liu, Xiaoshan Li, Jing Liu, and He Jifeng. Integrating and refining UML models. Technical report, UNU-IIST, March 2004. Report No. 295.

The report looks at UML multi view modelling and tries to give a formal solutions to the inherent model consistency and model integration problems. It is noted that the majority of the research in this area concentrates on formalising a single diagram. They note that there is little work on refinement of UML models. As this report is on UML there are no direct implications for

[1] Commercial model checker for CSP, [11]

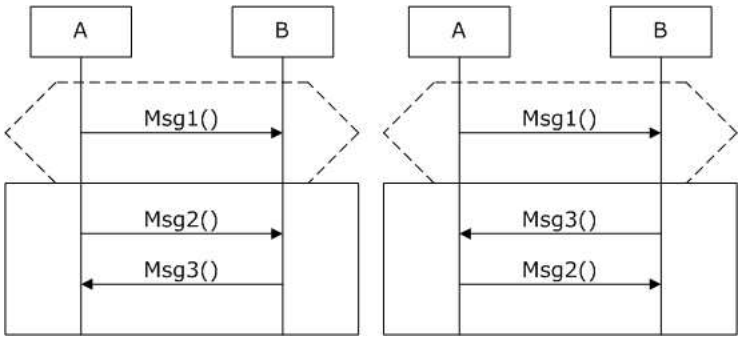

Figure 6.10: An example of inconsistency. Both main charts are activated by the message Msg1. The left LSC then specifies with hot locations that Msg2 must happen before Msg3. The right specifies that Msg3 must be sent before Msg2. Clearly a contradiction.

our treatment of LSC. It is an attempt to formalise the inherently informal UML whereas our approach is more in the direction of adding graphical notations to RSL.

The approach is to model a system with a system model consisting of a Class Diagram, several Sequence Diagrams, several State Machines and a system constraint. It then uses OOL, which is an object-oriented language, to specify the system. They do not consider parallelism and they adopt the run-to-complete semantics of UML state-machines. This would not allow for a state machine to be interrupted by some external event, as is the case by a false hot condition in LSCs. In an attempt to lessen the complexity of the task many of the more advanced features of the diagrams have been omitted.

[50] Jun Sun and Jin Song Dong. Synthesizing distributed processes from scenario-based specification. not published, 2004. Paper written in conjunction with Ph.D. Thesis at School of Computing, National University of Singapore, (sunj, dongjs)@comp.nus.edu.sg.

The paper is a followup on [49] and is still in draft. It gives a more complete overview of LSCs and CSP. It steps through the construction of equivalent CSP processes of LSCs. One special operator in CSP worth of mentioning, is the interrupt operator. It gives the possibility of interrupting a CSP process on a specified event, which is crucial when constructing the CSP equivalent of conditions, where instances may abort or stop. An example of a synthesised CSP process is given.

6.3.2 Summary of preliminary thesis

[2] Steffen Andersen and Steffen Holmslykke. From UML to RSL – and back again!, July 2004.

In our preliminary thesis we examined LSCs with the goal of creating an equivalent RSL specification in mind. The initial idea was to use the CSP constructs of RSL in order to specify the concurrent nature of LSCs. All the work has to some extend been used in this thesis. However almost all of it has been rewritten and augmented as our understanding of LSCs grew. Minor details from the preliminary thesis have not been included, so the interested reader is encouraged to read it. The development of the LSC related specifications can be viewed as an iterative process during the preliminary thesis and thesis period.

In the preliminary thesis LSCs and their features are introduced. An initial syntax of a LSC is given. This was the basis for the syntax in this thesis. Based on the syntax well-formedness conditions are presented. Finally an initial trace semantics for charts is presented. It is initial as it gave us a feel if we were on the right track. The specifications where not translatable.

The presentation of LSCs has to some extend been reused from the preliminary thesis. We felt it necessary to develop it further in order to better explain LSCs to the reader. Especially the general concepts of using LSCs were not described in enough detail.

6.4 The LSC subset chosen: RSC

In this section it is explained in more detail than the preliminary thesis [2] which subset/version of LSC that is chosen to be modelled in order to set the stage for the RSL types, well-formedness conditions and semantics. In order to avoid confusion, the subset/version of LSC is called RSC, an abbreviation for RSL Sequence Charts. It is written as subset/version since there is no general accepted version of LSC. Nearly every paper has its own restrictions and extensions compared to the initial version introduced in [7], which was also apparent in section 6.3. This resembles the introduction of State Charts, where it took a long time to reach an agreed upon syntax/semantics. Still today there are many. The most popular however is the one found in StateMate [27].

The discussion here is focused on the syntax and semantics in an informal way in order to give an explanation for the choices made in the following section. The types and semantics will be treated formally with RSL in section 6.5 along with more specific comments. They are not presented here as the types also depend on the previous section.

6.4.1 Collections

The basic idea of using LSCs is to specify requirements of a system by describing the behaviour that is mandatory and optional. Since a single chart only has limited expressive power it would make little sense only to look at single LSCs [24] as done in [36]. Our aim is therefore to model several LSCs. We will call that a collection. That name is chosen rather than the normally used specification in order to avoid confusion with a RSL specification. Precharts are crucial since they may activate the corresponding mainchart by events taken place on other events, see figure 6.11.

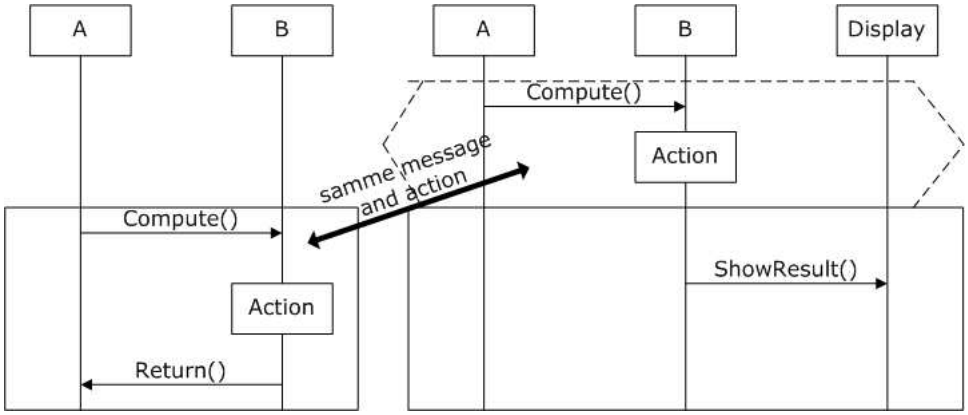

Figure 6.11: The left chart specifies some behaviour that the system must conform to. The first two events are also present in the prechart of the left chart. When they happen, the right mainchart is activated. Therefore the event Return() and *ShowResult() must be performed after Compute() and Action. Therefore the left LSC "activates" the right.*

6.4.2 Charts

We will only consider universal charts. They are much more expressive compared to existential charts which resemble MSCs weak semantics. As we want to specify behaviour that must happen, we only model universal charts in this formal specification context [36].

In precharts we do not consider a separate activation condition as introduced in [7], as this can be modelled by using a condition in the prechart.

6.4.3 Instances/Locations

From a modelling point of view, an instance is an abstraction. It can represent a specific object, a collection of objects etc. The RSC should provide enough freedom to model all those things, since we in reality are interested in the inter-object communication, whatever the object is.

We only consider hot locations in the semantics in the next section 6.5 in contrary to [26] where mostly cold locations are used. As hot locations denote mandatory behaviour, we think the force of LSC lies with these. We mainly want to use scenarios where we specify what the system **must** do, not what it may do. In [26] the Play-Engine is also primarily used with cold locations. This is not a problem as the Play-Engine actually executes the LSCs rather than using them as a requirement. This implicitly gives liveness since the play-engine always chooses to perform events on cold locations if they do not violate hot elements due to its maximal policy. However if all the locations are cold on a collection of LSCs, a system may fulfil the requirements by simply doing nothing, which is not desirable. Therefore hot locations give a more expressive applicative RSL description of a RSC (see section 6.7.1) as the specified events must happen.

Even though [26] omitted instance creation we felt it necessary to include it. The reason is that it should be possible to create an instance that is unaffected by the behaviour of other LSCs. Dynamic creation is not possible, as the created instances must be known beforehand. This is also a consequence of the graphical nature of RSCs. An example of creation can be seen in figure 6.5.

6.4.4 Subcharts

Regarding subcharts we could have considered simple subcharts that must span all instances in a LSC as in [24]. However this severely limits the expressibility in the chart. Therefore a subchart may span arbitrary instances, including a single one, which may be necessary for iteration over a local action.

However we do not allow unbounded iteration as this would pose a big problem in creating our operational trace function, which must return a finite result, see section 6.5.3. This is clearly easier to model using automata as in [6] with circular transitions.

For the sake of simplicity we have chosen not to create a separate if-then-else construct [24] with two concatenated subcharts and only one condition evaluation. The construct must be modelled with two separate subcharts with two complementary conditions. This potentially gives rise to two different evaluations of the conditions with the implication that none or both of the subcharts may be evaluated.

6.4.5 Events

Messages/Coregions

Messages are the fundamental building blocks of LSCs, and as such they come in several variants as explained in section 6.2. We started out by only modelling hot, synchronous messages since our aim was to implement messages using RSLs CSP constructs. Cold messages would mean that messages could be lost. Since messages are synchronous in CSP it would be necessary to create an intermediate process which nondeterministically would chose whether to pass on the message or delay it indefinitely (same limitation as in [49]). Thus we do not allow a situation like in figure 6.12.

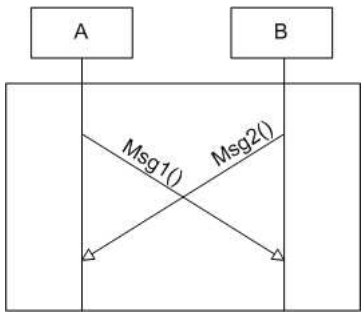

Figure 6.12: Example of message overtaking. Both messages are asynchronous, which can be seen as their arrowheads are hollow. This means they can be sent before they are received, allowing the above construct.

Another argument for hot messages combined with hot locations are the semantic problems arising as described in [26]. See figure 6.13.

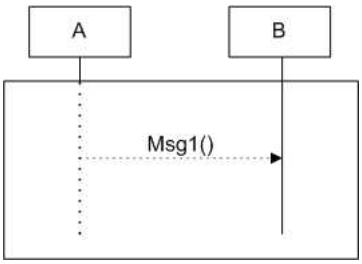

Figure 6.13: Example of a problem with the notion of hot/cold: Instance A has only cold locations, thus specifying that the events may *happen. The message is cold, denoting when it is being sent, it must not be received. However instance B is specified with cold locations, which denote that the events* must *happen. Now, must the message be sent or not? Must it be received or not?*

Regarding asynchronous messages an intermediate process could be created that delays the passing on for a nondeterministically chosen amount of time, thus demanding modelling of time, see below. We anticipated that such an process always could be added later to a specification in order to allow these kinds of messages. We chose to simplify the well-formedness conditions and semantics by only modelling synchronous messages. Especially the well-formedness condition that the transitive closure of the partial order of messages and conditions must be acyclic would become somewhat more complicated. A result of this simplification is that messages now have the semantics of synchronisation barriers between two instances, which of course limits the expressibility.

Messages in LSCs also allow for parameters, effectively functioning as value passing. In the RSC context message arguments are omitted as we use variables. More on the handling of this in section 6.7.1.

Coregions are modelled as proposed, allowing the sending/reception of messages in a coregion to occur in a non-determined sequence.

Conditions

Conditions are allowed to span several instances as in [7]. Others have constrained them to one instance, e.g. [49]. LSCs are all about inter-object communication and are thus suitable for modelling distributed systems. However in distributed systems all instances would not have access to the predicate/value that determines the outcome of the condition [41]. Only one execution is desired. If every instance evaluated the condition they could reach different results since time is not ordered among instances.

We allow shared conditions which means the system would require some sort of synchronisation communication. A condition may then be viewed as an abstraction of that in order to simplify the graphical representation with unnecessary messages.

It is noted that a hot condition is quite similar to an assertion in a program, it must be true at the given moment in order for a system to be an correct implementation of the model.

Conditions are abstracted into simple names as it must be possible for the software engineer to exactly specify the predicate in RSL.

Actions

Actions are in [7] only specified as updating arbitrary variables with arbitrary values. In [26] these are completely omitted as the idea is to completely specify the behaviour of the system including variables, thus not needing an action event. We kept the action event in order to allow for local computations on an instance that may update variables. Especially in RSL this is useful when refining an specification, where a local action may be under-specified to begin with.

Forbidden events

We do not consider forbidden events which were introduced in [26]. Our main priority was to handle events on LSCs that must happen.

6.4.6 Timing

Time and timing in RSC was one of the main concerns regarding RSC. The initial syntax completely lacked semantics and not until [25] time was treated in more detail. Unfortunately (albeit naturally) the semantics are quite complex and the included syntax is fundamentally different as the one initially indicated in [7] on which we based our initial syntax in the preliminary thesis [2]. The initial type of timing constructs we worked on only allowed for timing annotations local to an instance, which is not very useful when modelling a distributed system [41]. Having studied the subject more thoroughly and examined what was possible with *Timed RSL* [52] we chose not to model time and timing annotations. *Timed RSL* has a single timing construct, called wait, which semantically only gives the possibility to introduce delay. The semantic of *wait x* is: It will wait at *wait* for at most *x* time units. This not enough for modelling the constructs possible in [25] which introduces global time and ticks.

6.5 Formal description of RSC

In the following the syntax, well-formedness conditions and semantics for RSC are introduced. The specifications are to be found in appendix D. They are based on the work in our preliminary thesis and have been augmented and rewritten in this thesis. One large difference is that they were made translatable, see section refsec:sd-making-model-exec. Some of the following text is based on the preliminary thesis [2].

One of the advantages of making the specifications translatable was the possibility for tests. Extensive tests have been included in appendix D.4 including comments.

As the initial plan was to create an executable RSC collection it was also started to work on types, well-formedness conditions and semantics for these. These where not needed in the resulting applicative RSC. However the interested reader can find the (unfinished) specification in the appendix D.3. It gives an idea of how an executable collection of RSCs might look like. It is presented with comments.

6.5.1 Types

The following types define the syntax of a single RSC chart. The types are reworked from the preliminary thesis [2]. Sorts like names of messages are refined as texts, allowing for translation.

A RSC has a name, prechart, mainchart and a set of created instances. A RSC is modelled consisting of a pre- and mainchart since their use is fundamentally different, even though their syntax is the same. Creations must be the instance names of the instances that are created in the main chart.

type
RSC′ ::
name : RSC_Name
prechart : Chart
mainchart : Chart
creations : Inst_Name-**set**,

A chart is a map from instance names to a list of locations in order to ease look up of locations. We will also use the index on the list in order to model the state in the semantics.

type
Chart = Inst_Name $\xrightarrow[m]{}$ Location*,
Location :: temp : HotCold event : Event,

An event may be one of the following: action, message input/output, condition, coregion or subchart begin/end. Coregion and subcharts have been included as events in order to simplify the specifications.

An action event has a name and an id. It is further unspecified as we want to specify it in RSL later on.

An input event has an id and an source. The message name is given in the corresponding, uniquely identifiable message output event.

An output event has an id, a name and a destination. No parameters are included as this is later done via variables in section 6.7. The temperature is not included since we only consider hot messages in the semantics.

A condition event has a name, an id, a temperature and a list of instance names. The list denotes which instances share the condition.

A coregion event has a location list. It must be the sublist of events on the instance that contains the message events that may happen in random order.

A subchart event has a name, an id, a list of instance names, a multiplicity and a location list. The instance names denote the instances among which the subchart is shared. The multiplicity denotes the maximum number of repetitions of the chart.The location-list is the part of the instance's locations that the subchart encloses.

A endsubchart event has an id. It is used to denote the end of a subchart with the given id.

A stop event is included as the last event on an instance.*/

type
Event ==
mk_ActionEvent(Action_Name, aid : ID) |
mk_InputEvent(inmsgid : ID, isender : Address) |
mk_OutputEvent(
outmsgid : ID,
outpid : Msg_Name,
dest : Address) |
mk_ConditionEvent(
conname : Cond_Name,
cid : ID,
cetemp : HotCold,
ceshare : Inst_Name-**set**) |
mk_CoregionEvent(crlocl : Location*) |
mk_Subchart(
scname : Subchart_Name,
scid : ID,
scshare : Inst_Name-**set**,
mult : Multiplicity,
sclocl : Location*) |
mk_EndSubchart(escid : ID) |
StopEvent,

Temperature may be hot or cold.

type
HotCold == Hot | Cold,

A multiplicity is a positive number.

type
Multiplicity = {| n : **Nat** • n>0 |},

An address is Environment or an instance name.

type
Address == Environment | mk_Address(name : Inst_Name),

Identifiers are texts or integers.

type
Action_Name = **Text**,
Cond_Name = **Text**,
Inst_Name = **Text**,
RSC_Name = **Text**,
ID = **Int**,
Msg_Name = **Text**,
Subchart_Name = **Text**

6.5.2 Well-formedness conditions

The type `RSC` has been made a subtype of `RSC'` that is well-formed. A predicate has been defined in order to check all the well-formedness conditions, 18 in total. Since the specification is rather large it has been included in Appendix D.1.2. The conditions include comments on the purpose and argumentation why they are included. As an example, condition 3 is included here. It is one of the most important ones as it prevents deadlocks. It was also quite complicated since it was made translatable. All the conditions have been completely separated. This eases the understanding and gives the possibility of testing each one separately.

The following is a list of all the conditions with a short description:

1. (*wf_ids_unique*, D.1.2) IDs of events are unique in order to identify them.
2. (*wf_message_match*, D.1.2) Messages are paired correctly or stem from environment in order to ensure correct message passing.
3. (*wf_mess_cond_acyclic*, D.1.2) The undirected connection graph is acyclic in order to avoid deadlock.
4. (*wf_condition_share*, D.1.2) Conditions are consistent across instances that share the specific conditions.
5. (*wf_subchart_locations*, D.1.2) Locations in subcharts are locations on the instance in order to ease semantics and other well-formedness conditions.
6. (*wf_subchart_ordered*, D.1.2) All subcharts are ordered correctly in order to avoid deadlock For example they may not overtake each other across instances.
7. (*wf_subchart_coherent*, D.1.2) Subcharts are coherent in order to ensure correct subcharts.
8. (*wf_subchart_end*, D.1.2) A subchart has an endsubchart token in order to identify its end.
9. (*wf_subchart_conditions*, D.1.2) Conditions in subcharts are contained in order to ensure correct subcharts.
10. (*wf_subchart_messages*, D.1.2) Messages in subcharts are completely contained inside the subchart in order to ensure correct subcharts. A message may for example not be sent but not be received in a subchart.
11. (*wf_subchart_subchart*, D.1.2) Subcharts within subcharts are completely contained inside the superchart in order to ensure correct subcharts.
12. (*wf_coregion_locations*, D.1.2) Locations in coregions are locations on the instance in order to ease semantics and other well-formedness conditions.
13. (*wf_coregion_messages*, D.1.2) All events in coregions are messages, as coregions may only consist of messages.
14. (*wf_cold_subchart*, D.1.2) All locations after a cold location in subcharts are cold in order to ensure correct ordering of hot and cold messages.
15. (*wf_cold_mainchart*, D.1.2) All locations after a cold location in a mainchart are cold in order to ensure correct ordering of hot and cold messages.
16. (*wf_last*, D.1.2) The last event is a stop event in order to ensure correct termination of a chart.
17. (*wf_creation*, D.1.2) A created instance may not be present in prechart, as it is obviously not created until sometime in the mainchart.
18. (*wf_prechart_condition*, D.1.2) Conditions in prechart must be hot, since a cold condition could mean an unwanted successful exit from the prechart.

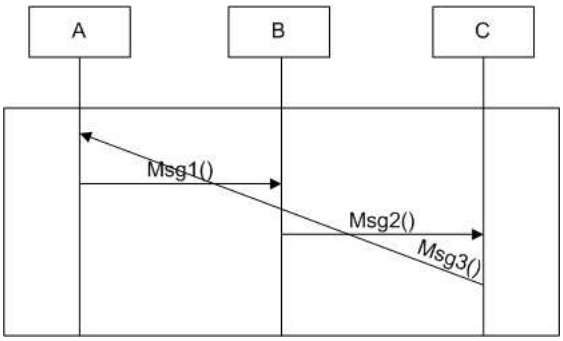

Figure 6.14: Example of a non well-formed RSC that deadlocks. No instance can proceed, as no event is enabled as all messages are hot and synchronous.

The following well-formedness condition is to ensure that no deadlocks occur. This is done by checking that the transitive closure of the bidirectional connection graph is acyclic. This property includes that a message

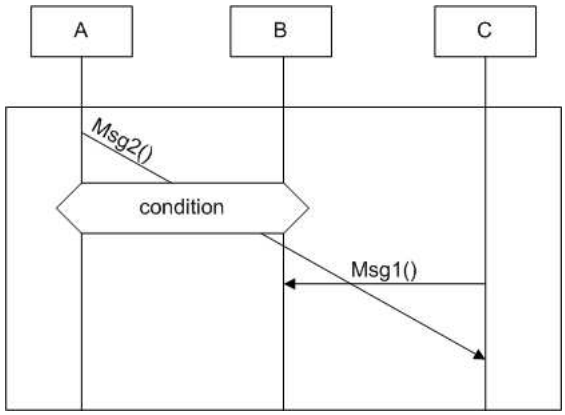

Figure 6.15: Another example of a RSC that deadlocks. This time with a condition. Again, no event is enabled as the connection graph is cyclic.

output is not causally dependent on its corresponding message input, directly and indirectly through other messages, see figure 6.14. However this property is not enough since messages are synchronous and thus will deadlock if the destination is not ready to accept the input. An example can be seen in figure 6.15, where the directed connection graph is not cyclic and the bidirectional is. Messages are a synchronisation barrier, thus the order introduced is given by the synchronisation points on the instances. As condition events also represent synchronisation points they need also be considered. The order is established by creating the tuples of IDs that happen in order, i.e. the tuples AB which denotes that the event with ID A happens before the event with ID B (on an instance). Now by ensuring that this order is not cyclic, the desired property is achieved. Even though subcharts are synchronisation points as well, they must not be considered, since messages, conditions etc. in subcharts are contained within the subchart and can thus not introduce a cyclic order due to the other well-formedness conditions. If the specification was not to be translated, the above condition could be specified very shortly. The following demonstrates this. The `orders` are all the orders introduced in the chart. For example if the message with ID *1* happens before a condition with ID *64*, the order *(1,64)* is included. It is then checked that there does not exist a sequence of these orders that is cyclic:

type
ID = **Int**,
Order :: id1 : ID id2 : ID

value
wf_mess_cond_acyclic : Chart → **Bool**
wf_mess_cond_acyclic(chart) ≡
  let orders = po_instances(chart) **in**
  ∼(∃ ol : Order* •
    (**elems** ol ⊆ orders) ∧
    (∀ i : **Nat** • i > 0 ∧ i < **len** ol ⇒ id2(ol(i)) = id1(ol(i + 1))) ∧
    id1(ol(1)) = id2(ol(**len** ol)))
  end,

The translatable version: For all given orders on a chart this functions checks weather the undirected connection graph is acyclic or not by trying to traverse the IDs reachable from each order present. Sorting is actually not necessary, but done in order to speed up the check in the resulting C++ code.

value
wf_mess_cond_acyclic : Chart → **Bool**
wf_mess_cond_acyclic(chart) ≡
  let
    orders = po_instances(chart),
    orderl = insertionSort(set_to_list(orders), ⟨⟩)
  in
      ∀ o : Order • o ∈ **elems** orderl ⇒

```
            acyclic({id1(o), id2(o)}, id2(o), orderl)
    end,
\RSLatex
```

Finds the order **of** messages and conditions on instances **of** a chart.

```
\RSLatex
po_instances : Chart → Order-set
po_instances(chart) ≡
    if chart = [ ] then {}
    else let i = hd chart in po_instance(chart(i), {}) ∪ po_instances(chart \ {i}) end
    end,
```

`po_instance` finds the order of messages and conditions on an instance (i.e. Location-list). It creates all the orders introduced by the instance by creating tuples of ID, e.g. AB which denotes that the message or condition with ID A happens before the message or condition with ID B. It calls itself recursively in order to treat each Location depending on its event. It maintains a set of ID's (idset) which holds the IDs that happen before the location at the beginning of the current Location-list. Messages and condition events have their ID extracted and an Order-set is created by using the extracted ID and the preceeding ids given by idset. It proceeds with the rest of the list and adds the extracted ids to idset, which now holds the ID's of messages and conditions that have happened. Coregionevents do not introduce ordering among themselves, only in relation to events happening before and after the coregion. All the ids of messages in coregion are extracted. Since a coregionevent spans more than one location all the locations of the coregionevent are removed in order to proceed. All other events are not relevant for ordering. Not even subcharts, since there are wf-conditions that guarantee that messages and conditions contained in subcharts do solely occur inside them.

```
po_instance : Location* × ID-set → Order-set
po_instance(locl, idset) ≡
    if locl = ⟨⟩ then {}
    else
     case event(hd locl) of
        mk_InputEvent(inmsgid, _) →
            append(idset, {inmsgid})
            ∪
            po_instance(tl locl, idset ∪ {inmsgid}),
        mk_OutputEvent(outmsgid, _, _, _, _) →
            append(idset, {outmsgid})
            ∪
            po_instance(tl locl, idset ∪ {outmsgid}),
        mk_ConditionEvent(_, cid, _, _) →
            append(idset, {cid})
            ∪
            po_instance(tl locl, idset ∪ {cid}),
        mk_CoregionEvent(clocl) →
            let cids = coregion_ids(clocl), newlocl = reduce_list((len clocl) + 1, locl) in
                append(idset, cids) ∪ po_instance(newlocl, idset ∪ cids)
            end,
        _ → po_instance(tl locl, idset)
     end
    end,
```

Given a set of already traversed IDs, the current last traversed ID and a list of orders, this function checks whether the connection graph is acyclic or not. This is done by finding the next possible orders given the current last traversed ID.

```
acyclic : Int-set × Int × Order* → Bool
acyclic(seenIDs, lastid, orderl) ≡
    let fitting = extractfitting(lastid, orderl) in acyclic_fit(seenIDs, fitting, orderl) end,
```

Given a set of already traversed ID's, a list of possible next orders and a list of all orders, this function checks weather the connection graph is acyclic or not.

acyclic_fit : **Int-set** × Order* × Order* → **Bool**
acyclic_fit(seenIDs, fitting, orderl) ≡
  case fitting **of**
    ⟨⟩ → **true**,
    ⟨a⟩ ⌢ ⟨⟩ → id2(a) ∉ seenIDs
      ∧ acyclic(seenIDs ∪ {id2(a)}, id2(a), orderl),
    ⟨a⟩ ⌢ b →
      id2(a) ∉ seenIDs
        ∧ acyclic(seenIDs ∪ {id2(a)}, id2(a), orderl)
        ∧ acyclic_fit(seenIDs, b, orderl)
  end,

Extracts the orders of a list (as a list) with a given id as id1.

extractfitting : **Int** × Order* → Order*
extractfitting(id, orderl) ≡
  case orderl **of**
    ⟨⟩ → ⟨⟩,
    ⟨a⟩ ⌢ ⟨⟩ → **if** id1(a) = id **then** ⟨a⟩ **else** ⟨⟩ **end**,
    ⟨a⟩ ⌢ b →
      if id1(a) = id **then** ⟨a⟩ ⌢ extractfitting(id, b)
      else extractfitting(id, b) **end**
  end,

6.5.3 Semantics for one chart

The semantics for RSCs is the set of all possible traces of a RSC. A trace is a list of possible states a RSC can be in during an execution. This list then describes one complete run of the RSC as it is performed. The states resemble the states described in 6.2.

As the semantics must be translatable to C++ we had to be very implementation oriented. The semantics adhere to the informal description presented in section 6.4, so it will not be discussed further here.

The semantics are defined for one chart, which means it is valid for both a prechart and mainchart. It consists of two main parts. The first examines a charts current state in order to find the set of events that may be performed. These events are called enabled events [26]. Based on the enabled events a set of all the possible traces of a RSC is computed by diverging every trace when several enabled events are present.

As the instances are modelled as lists it was decided to use the index of the current event in order to denote the state. Thus the position info is a pointer which points to the next event that must happen. As we are not talking about cuts but states, the state must also give extra information in case of a coregion. Most of the RSL specification is given in appendix D.2.

The following are the most important top level functions:

`semantics` gives all the possible traces for a given chart.

semantics : Chart → Traces
semantics(chart) ≡ eval(chart, {⟨initialize_chart(chart)⟩}),

`eval` recursively evaluates traces until no new are found.

eval : Chart × Traces → Traces
eval(chart, ts) ≡
  let ts′ = eval_traces(chart, ts) **in**
    if ts′ = ts **then** ts **else** eval(chart, ts′) **end**
  end,

`eval_traces` evaluates traces. There is a need for an extra function as recursively defined quantified expressions are not in the subset of RSL that is translatable to C++.

```
eval_traces : Chart × Traces → Traces
eval_traces(chart, ts) ≡
    --converting -set-set to -set
    convert_tss2s({eval_trace(chart, t) | t : Trace • t ∈ ts}),
```

`eval_trace` evaluates a trace by finding the next possible states given the current state, and creating the resulting new traces.

```
eval_trace : Chart × Trace ~→ Traces
eval_trace(chart, t) ≡
    let evalss = eval_state(chart, hd t) in
        if evalss = {} ∨ (∀ ns : State • ns ∈ evalss ⇒ ns =
        hd t ∨ (if len t > 1 then hd t = (hd (tl t)) else false end))
        then {t}
        else
            if (∃ ns : State • ns ∈ evalss ∧ ns = [ ])
            then
                {⟨ns⟩ ⌢ t | ns : State • ns ∈ evalss ∧ ns ≠ hd t ∧ ns ≠ [ ]} ∪
                {⟨hd t⟩ ⌢ t}
            else {⟨ns⟩ ⌢ t | ns : State • ns ∈ evalss ∧ ns ≠ hd t}
            end
        end
    end
    pre t ≠ ⟨⟩ ,
```

`eval_state`: given a chart and a state the enabled events are found and the corresponding next states are found.

```
eval_state : Chart × State → State-set
eval_state(chart, curState) ≡
    {step_event(chart, x, curState) | x : EnabledEvent • x ∈ get_enabled_events(chart, curState)},
```

`get_enabled_events` finds the enabled events in a given state.

```
get_enabled_events : Chart × State → EnabledEvent-set
get_enabled_events(chart, curState) ≡
    let inames = dom curState in
        convert_ees2es(
                {get_enabled_event(chart, curState, iname) | iname : Inst_Name • iname ∈ inames})
    end,
```

Same as above, splitted for convenience.

```
get_enabled_event : Chart × State × Inst_Name → EnabledEvent-set
get_enabled_event(chart, curState, iname) ≡
    let pi = curState(iname), curEvent = eventlist(chart(iname))(pointer(pi)) in
        get_enabled_event_cur(chart, curState, iname, curEvent)
    end,
```

`get_enabled_event_cur` gets the enabled events depending on the current event.

```
get_enabled_event_cur : Chart × State × Inst_Name × Event → EnabledEvent-set
get_enabled_event_cur(chart, curState, iname, curEvent) ≡
    case curEvent of
    mk_InputEvent(inmsgid, inaddr) → {get_ee_inputevent(chart,
    curState, iname, inmsgid, inaddr)},
    mk_OutputEvent(outmsgid, temp, outpid, outpar, outaddr) →
```

{get_ee_outputevent(chart, curState, iname, outmsgid, outaddr)},
mk_ConditionEvent(conname, cid, cetemp, share) →
get_ee_con(chart, curState, iname, curEvent),
mk_CoregionEvent(locl) → get_ee_coregion(chart, curState, iname, locl),
mk_Subchart(scname, scid, inames, mult, sclocl) →
{get_ee_subchart(chart, curState, curEvent)},
mk_EndSubchart(scid) → {get_ee_endsc(chart, curState, scid)},
StopEvent → {EnabledStopped},
mk_ActionEvent(aname,id) → {EnabledAction(iname,aname, id)}
end,

6.6 Example: RSC RSL specification

Figure 6.16 is a constructed example of RSC. It does not represent a specific scenario, but is created in a way to accommodate all the possible RSC constructs in one chart without getting too big. The equivalent RSL specification of the RSC is presented. This is followed with examples of how the semantics for the mainchart work when executed.

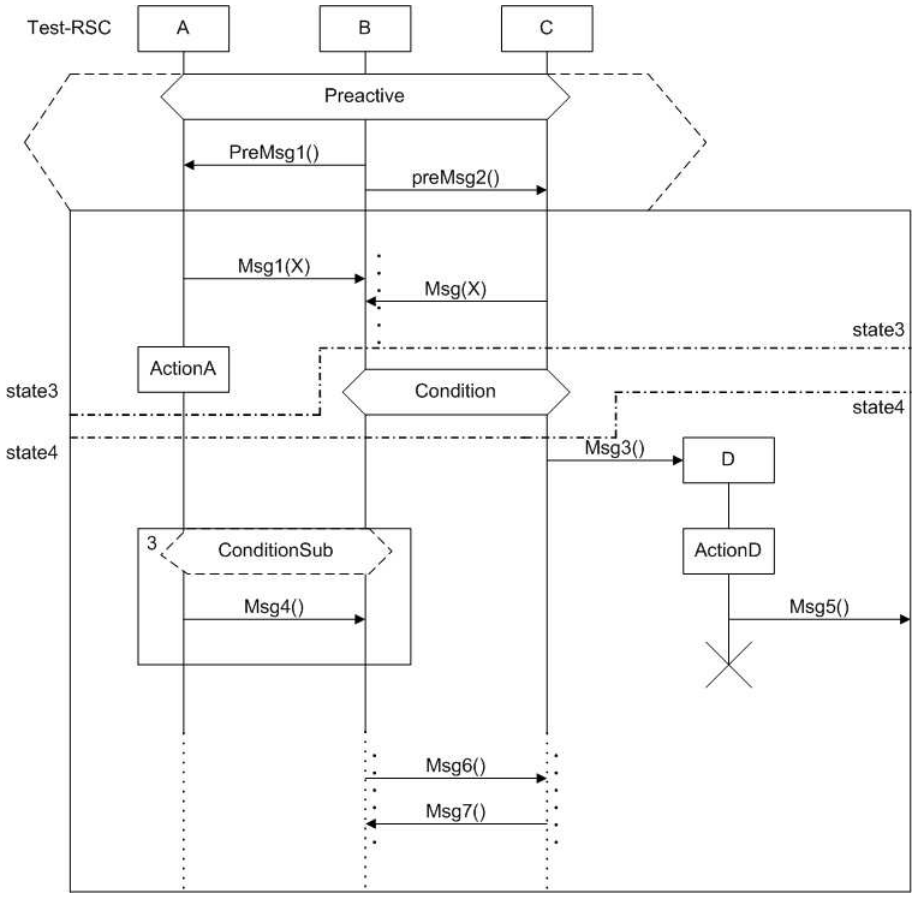

Figure 6.16: Example of a RSC with all the possible constructs.

The following is the equivalent RSC specification in RSL including a small part of the execution using the functions in the semantics given in D.2. The compiled test binaries can be found on the companion CD, see chapter E.

All the events are defined.

value
pre_m1out : Event = mk_OutputEvent(01, "PreMsg1", mk_Address("A")),
pre_m1in : Event = mk_InputEvent(01, mk_Address("B")),
pre_m2out : Event = mk_OutputEvent(02, "PreMsg2", mk_Address("C")),
pre_m2in : Event = mk_InputEvent(02, mk_Address("B")),
pre_cond : Event = mk_ConditionEvent("Preactive", 03, Hot, {"A", "B", "C"}),
main_m1out : Event = mk_OutputEvent(1, "Msg1", mk_Address("B")),
main_m1in : Event = mk_InputEvent(1, mk_Address("A")),
main_m2out : Event = mk_OutputEvent(2, "Msg2", mk_Address("B")),
main_m2in : Event = mk_InputEvent(2, mk_Address("C")),
main_m3out : Event = mk_OutputEvent(3, "Msg3", mk_Address("D")),
main_m3in : Event = mk_InputEvent(3, mk_Address("C")),
main_m4out : Event = mk_OutputEvent(4, "Msg4", mk_Address("B")),
main_m4in : Event = mk_InputEvent(4, mk_Address("A")),
main_m5out : Event = mk_OutputEvent(5, "Msg5", Environment),

```
main_m6out : Event = mk_OutputEvent(6, "Msg6", mk_Address("C")),
main_m6in : Event = mk_InputEvent(6, mk_Address("B")),
main_m7out : Event = mk_OutputEvent(7, "Msg7", mk_Address("B")),
main_m7in : Event = mk_InputEvent(7, mk_Address("C")),
main_actA : Event = mk_ActionEvent("ActionA", 11),
main_actD : Event = mk_ActionEvent("ActionD", 12),
main_cr1 : Event = mk_CoregionEvent(⟨mk_Location(Hot, main_m1in),
    mk_Location(Hot, main_m2in)⟩),
main_cr2 : Event = mk_CoregionEvent(⟨mk_Location(Cold, main_m6out),
    mk_Location(Cold, main_m7in)⟩),
main_cr3 : Event = mk_CoregionEvent(⟨mk_Location(Cold, main_m6in),
    mk_Location(Cold, main_m7out)⟩),
main_condmain : Event = mk_ConditionEvent("Condition", 21, Hot, {"B","C"}),
main_condsub : Event = mk_ConditionEvent("ConditionSub", 22, Hot, {"A", "B"}),
main_scla : Location* = ⟨mk_Location(Hot, main_condsub),
    mk_Location(Cold, main_m4out)⟩,
main_sclb : Location* = ⟨mk_Location(Hot, main_condsub),
    mk_Location(Cold, main_m4in)⟩,
main_sca : Event = mk_Subchart("Subchart", 31, {"A", "B"}, 3, main_scla),
main_scb : Event = mk_Subchart("Subchart", 31, {"A", "B"}, 2, main_sclb),
main_scstop : Event = mk_EndSubchart(31),
```

Defining main chart instances (location lists).

```
main_insta : Location* =
    ⟨mk_Location(Hot, main_m1out),
        mk_Location(Hot, main_actA),
        mk_Location(Hot, main_sca)⟩
        ⌢ main_scla ⌢
    ⟨mk_Location(Hot, main_scstop),
        mk_Location(Hot, StopEvent)⟩,
main_instb : Location* =
    ⟨mk_Location(Hot, main_cr1),
        mk_Location(Hot, main_m1in),
        mk_Location(Hot, main_m2in),
        mk_Location(Hot, main_condmain),
        mk_Location(Hot, main_scb)⟩
        ⌢ main_sclb ⌢
    ⟨mk_Location(Hot, main_scstop),
        mk_Location(Cold, main_cr2),
        mk_Location(Cold, main_m6out),
        mk_Location(Cold, main_m7in),
        mk_Location(Cold, StopEvent)⟩,
main_instc : Location* =
    ⟨mk_Location(Hot, main_m2out),
        mk_Location(Hot, main_condmain),
        mk_Location(Hot, main_m3out),
        mk_Location(Cold, main_cr3),
        mk_Location(Cold, main_m6in),
        mk_Location(Cold, main_m7out),
        mk_Location(Cold, StopEvent)⟩,
main_instd : Location* =
    ⟨mk_Location(Hot, main_m3in),
        mk_Location(Hot, main_actD),
        mk_Location(Hot, main_m5out),
        mk_Location(Hot, StopEvent)⟩,
```

Defining prechart instances.

pre_insta : Location* =
⟨mk_Location(Hot, pre_cond),
mk_Location(Hot, pre_m1in),
mk_Location(Hot, StopEvent)⟩,
pre_instb : Location* =
⟨mk_Location(Hot, pre_cond),
mk_Location(Hot, pre_m1out),
mk_Location(Hot, pre_m2out),
mk_Location(Hot, StopEvent)⟩,
pre_instc : Location* =
⟨mk_Location(Hot, pre_cond),
mk_Location(Hot, pre_m2in),
mk_Location(Hot, StopEvent)⟩,

Defining charts.

mainch : Chart = [″A″ ↦ main_insta,
″B″ ↦ main_instb,
″C″ ↦ main_instc,
″D″ ↦ main_instd],
prech : Chart = [″A″ ↦ pre_insta,
″B″ ↦ pre_instb,
″C″ ↦ pre_instc],

Defining RSC subtype.

testrsc : RSC′ = mk_RSC′(″Test-RSC″, prech, mainch, {″D″}),

Start state and initial enabled events. Provided in order to shorten the paramter list.

state0 : State = initialize_chart(wfl_mainch),
ee0 : EnabledEvent-**set** = get_enabled_events(wfl_mainch, state0)

As the specification is translatable it is possible to create test cases. They all return true when run. The following assumes that the mainchart is activated and it is executed.

Checking if the RSC is wellformed.

test_case
[wfl_RSC_is_wellformed_____]
wf_RSC(testrsc),

The following returns the state after 3 events have been performed. The state corresponds to the state3 drawn in figure 6.16. The state is made more human readable by only showing the program pointers on the instances. The numbers thus denote at which location the next possible event is located on each instance. The ordering is instance A,B,C,D. Instance D is waiting at position 1, waiting for its "activation" message.

test_case
[state_3_steps_____]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(3, state0, ee0),**true**))) =
⟨3,4,2,1⟩,

The following shows which events are enabled in the above state. Currently only the condition is enabled.

test_case
[enabled_events_3_steps_____]
ads(test_machine(wfl_mainch, mk_gStep(3,state0,ee0),**true**)) =
{NotEnabled,EnabledCondition({″B″,″C″},″Cond1″,12)},

This is the state after the condition has been performed, state4 in the figure.

test_case
[state_4_steps_____]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(4, state0, ee0),**true**))) =
⟨3,5,3,1⟩,

Now there are 2 possible events, the entering of the subchart and sending of Msg3.

test_case
[enabled_events_4_steps_____]
ads(test_machine(wfl_mainch, mk_gStep(4,state0,ee0),**true**)) =
{EnabledMessage({″C″},{″D″},81),EnabledEnterSubchart({″A″,″B″},13,2),NotEnabled}

6.7 Translation: $\text{RSC}_\delta \rightarrow \text{RSL}_\alpha$

This section presents a RSC equivalent specification in RSL that allows refinement of messages, actions, and conditions. The RSC described in section 6.4 is the subset of LSCs for which there is specified syntax and well-formedness conditions.

RSL provides several different ways of modelling. LSCs/RSCs have a close conceptual relation to processes and message passing. Therefore it is intuitive to think of the CSP included in RSL. This was also our initial approach of modelling a RSC. But we found that the CSP operators available in RSL are not sufficient to make a general model. This initial approach is discussed in section 6.7.2.

The intended purpose of a collection of LSCs is to specify the requirements of a system. Thus the equivalent RSL specification must achieve the same. Besides the fact that the CSP part of RSL is simplified we also think that using CSP processes and channels makes the specification design oriented. The goal in [26] is the executability of LSCs, thus working towards a design specification. Our goal is to make the RSCs as expressive as possible with a high degree of freedom for the software engineer. Therefore we chose to use an applicative style which captures the requirements expressed using RSCs/LSCs. This is discussed in section 6.7.1.

We also present the pure CSP approach of [50] in the last section.

6.7.1 An applicative RSL model of RSCs

In section 6.5.3 the semantics for a RSC is given as traces of states. The purpose is to visualise the possible ways of progressing a RSC, disregarding the actual meaning of the events. The requirements in the RSL specification must express the constraints imposed by the collection of RSCs. Hence it is necessary to include the semantics of the events in the charts. The semantics presented in this section describe a set of models where each model represent a trace of events.

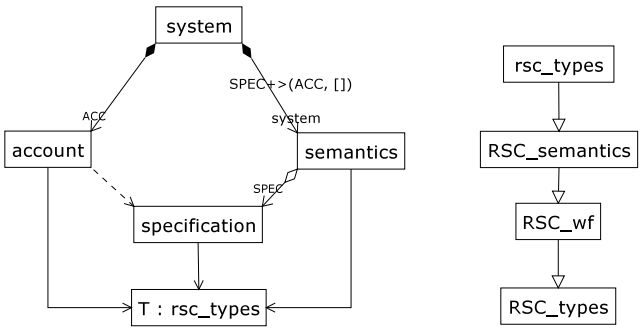

Figure 6.17: Scheme Diagram of the account example RSC specification.

The structure of the specification is shown in figure 6.17. A *system* consists of two parts:

1. A set of RSC specifications and the functions which are used in the RSCs.
2. A generic scheme which given the information in item 1 as parameter, returns the set of models, each containing a valid trace of the system.

In figure 6.17 the *account* scheme contains the set of RSCs and functions in RSL. It must implement the *specification* since it denotes the formal arguments for the *semantics* scheme. An account example with diagrams and explicitly specified functions is given in 6.8. The *semantics* and *rsc_types* schemes are discussed further in the following sections.

The schemes from the semantics in section 6.5.3 are also present in figure 6.17. The RSC specified by the user must be well-formed, thus we reuse the syntax and well-formedness conditions. Furthermore there are two functions ((*get_enabled_events, step_event*, D.6) from the *RSC_semantics* which still are applicable. The functions are used for determining enabled events and the next states in a RSC.

The full RSL specification for figure 6.17 is available in appendix D.6. It is augmented with a lot of comments regarding how the specification works in relation to RSC and is worth reading for the interested reader.

Events versus functions

There are three kinds of events that can take place in a RSC: a message, action or condition. In a RSC it is possible to prescribe the events that can/must happen but not the effect of an event. This can be done in RSL. But first the events must be related to RSL. We have chosen to map events to functions. In order for functions to have an effect we also introduce variables, see section 6.7.1.

```
type
  SysEvent′ ==
    mk_SysAction(instance : Inst_Name, action : Text) |
    mk_SysMsg(src : Inst_Name-set, dst : Inst_Name-set, method : Text) |
    mk_SysCondition(shared_by : Inst_Name-set, cond : Text),
```

A condition event is a predicate which is given the names of the instances it spans and variables as parameter. An action event is a local computation, thus given the name of the instance on which it is placed and variables as parameters. Message events can be considered a function invocation at the destination instance initiated by the source instance. Hence the source instance, destination instance and variables are given as parameters.

```
type
  Condition = Inst_Name-set × Variables → Bool,
  Action = Inst_Name × Variables → Variables,
  Message = Inst_Name-set × Inst_Name-set × Variables → Variables
```

The RSC is given to the *semantics* as part of the actual parameter of the scheme. The three events are in the syntax labelled with the names of the events/functions as a **Text** type. In order to use use this label and make the function calls generic, three maps from function names to functions are introduced.

```
value
  actions : Text m→ T.Action,
  conditions : Text m→ T.Condition,
  messages : Text m→ T.Message
```

Applying a function must be preceded by a map lookup, but makes it possible to apply a function given its name as a text string. The three maps are part of the formal parameters for the *semantics* scheme.

Variables

In order for the functions to have an effect it is necessary to have variables. A global variable state is introduced which maps variable names to values. All functions when applied are given the variables as parameters thus all have the possibility to read variables. Both messages and actions are events that can have an effect, therefore they must return a map of variables. A condition is a predicate thus its return type is a boolean.

```
type
  Variables = Text m→ Value,
  Value == Boolean(Bool) | Integer(Int) | String(Text)
```

Since the variable state is modelled as a map, in addition to changing the value of a variable it also permits the functions to create, delete, and change the type of a variable. It is however required that the functions handle situations where required variables are missing or of the wrong type.

Since the variable state is global all functions have access to all variables. It is however still possible to model local variables using generic functions or predicates. According to the well-formedness of the RSC the instances all have unique names. For actions the name of the instance where the action must be performed is given as parameter. Accessing a variable is preceded by a map lookup using the variable name. Thus a local variable can be preceded by the instance name, eg: `Account.balance`. Whether local or global variables are used are up to the specifier of the functions.

Traces

The *semantics* scheme contains two under-specified value declarations being a system trace (*st*) and an initial variable state. The system trace is constrained by two axioms which are true if the system trace fulfils the RSCs part of the actual parameter. There are no constraints on the initial variable state. Therefore its presence corresponds to an $\exists$ quantification. Thus each model in the set of models from the *semantics* scheme, represents a valid trace and initial variable state combination.

type
SysTrace = SysEvent$^{\omega}$

value
st : T.SysTrace,
v : T.Variables,
vl : T.Variables* = make_var_trace(st, v)

Although several different RSCs can be activated at the same time, it is simpler to verify one RSC with the system trace at a time. The entire system trace is available, but it is also necessary to know the variable state. Only messages and actions may alter the state of the variables thus it is possible to compute a variable state trace that corresponds to the system trace. Thus it is assumed that the variables trace is a consequence of the system trace and the initial variable state.

The system trace is not restricted to events that are part of the RSCs which constitute the system. When evaluating a chart these events matter. In a prechart such events are ignored, as the prechart is only concerned with the events that are specified (visible). In a mainchart these events may not alter the variables which are visible to the chart. This would diminish the semantic expression of conditions. If non visible events where allowed to change the visible variables, the following could happen: A condition is evaluated to true and the mainchart is about to proceed. Now an invisible event alters the variable which forms the basis for the condition so that it would return false. Then the enabled event proceeds event though the condition would evaluate to false. This is not the intention of using conditions and therefore the described limitation.

axiom
[valid_trace_no_prechart]
$\forall$ rsc : T.RSC •
rsc $\in$ SPEC.rscs $\wedge$ T.prechart(rsc) = [ ] $\Rightarrow$ valid_trace_nopre(rsc, st, vl),
[valid_trace_with_prechart]
$\forall$ rsc : T.RSC •
rsc $\in$ SPEC.rscs $\wedge$ T.prechart(rsc) $\neq$ [ ] $\Rightarrow$ valid_trace(rsc, st, vl)

Discussion

Although the RSC diagram is not designed with refinement in mind this option is possible when translating it to RSL. The original applicative `valid_trace` function describes the requirements. This function can be refined using e.g. CSP as previous mentioned. Thus a door has been opened which makes it possible to continue the transition to a design specification. The following is a discussion about some of the features of the current specification.

There is a difference in the semantics of a RSC with and without a prechart. When a prechart is present it controls when the mainchart is active. If there is no prechart it is implicitly true and the mainchart is automatically active. This means the behaviour specified in the mainchart must always hold. This can however pose a problem when we look at activation. If multiple activation is allowed this poses a problem. Example: a mainchart with two distinct events will always result in the empty set of models. Assume the first event happens. After this event the mainchart is again activated, requiring that the first event happens again, see figure 6.18 But this contradicts the first activation which requires the second event to happen. This is possible since both incarnations have the same visible events. We therefore require that the mainchart of a RSC without a prechart must be completed before it is activated again.

It is another case when considering RSCs with prechart, where we allow simultaneous activation. This may seem inconsistent compared to the constraint mentioned above with RSCs without prechart. However the

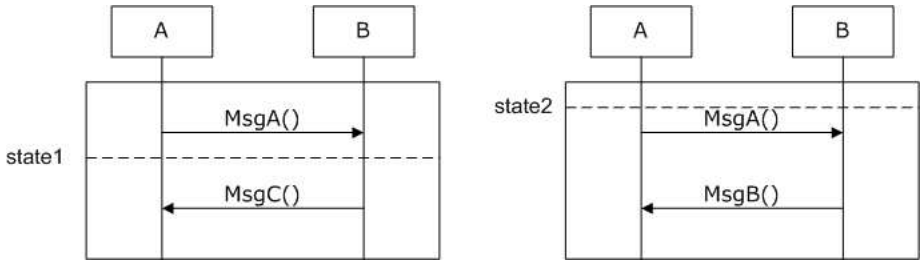

Figure 6.18: Illustrating problem if multiple activation of LSCs without prechart is allowed. MsgA has been sent. The left incarnation of the RSC requires that MsgB is sent. The right (last) incarnation requires MsgA to be sent.

expressiveness is enhanced greatly when allowing simultaneous activation, as can be seen in figure XX (include figure from playbook, page 154). Other approaches, as for example [24] have more severe constraints. Due to the state-based nature they do not even allow simultaneous activation of precharts, thus potentially missing activations of maincharts.

In our framework it is possible to specify inconsistent RSC. An example is given in figure 6.10. Thus the RSL specification will yield an empty model. As it is outside the scope of this project to check inconsistency, it is up to the software engineer to ensure consistency in the RSC collections used.

One of our design decisions was not to consider forbidden events. This is quite easy to change in the current specification by augmenting/altering the function (*visible_events*, D.2) by returning the events that are wished to be forbidden as well. Forbidden scenarios ([26]) which specify unwanted behaviour, are possible. This is simply done by specifying the unwanted behaviour in a prechart that leads to a false hot condition in the mainchart. This ensures that the system may never exhibit the behaviour specified in the prechart.

A more delicate subject is about conditions. As discussed in 6.4 we have chosen to include conditions that can span several instances. As conditions are predicates that need to read variables, it is unclear where these variables are read. As of now it is up to the software engineer to specify this correctly. As a starting point all variables are global, thus avoiding this problem. But RSCs are very suitable for distributed systems where this assumption does not hold, thus requiring refinement by the software engineer.

Cold locations have also been omitted due to the problems described earlier. A possible solution in this applicative context could be the following: if the events specified in the cold part of a chart happen, they must conform to the partial order induced by the chart.

Regarding messages to and from the environment it is chosen to model a destination/source as a set of instance names. If it is empty it denotes the environment. If not it may only contain one instance name that uniquely identifies the destination/source.

The current specification also implies a finite run of a system. However it would only require minor changes in order to allow infinite runs. It was chosen not to do so, as the semantics for one chart are finite as well. Subcharts may not be repeated indefinitely, as the initial goal was to use CSP in RSL. Thus it would be obvious to correct this issue in both the chart semantics and the RSL specification.

6.7.2 RSL CSP and LSCs

The initial approach of using CSP in RSL was initiated by trying to create examples in order to explore the possibilities. Two unfinished examples can be found in D.5. The idea was to construct a RSC process as a parallel composition of processes, one for each instance. Then specify the message passing as specified by the RSC equivalents. A main problem quickly emerged regarding the semantics of hot and cold conditions. Especially hot conditions were hard to specify as they required some sort of interrupt. This stems from the fact that a false hot condition is not allowed, and thus must stop all instances, regardless of their current state. This problem is easily solved using solely CSP, as described in 6.7.4.

Another major problem was the simultaneous activation of several charts. Individual RSCs only constrain some of the events. But in order to specify a complete system the sum of the constraints must be expressed. In order to solve this problem, the resulting RSL specification had to be a sum of all the RSCs. Thus needing some

sort of synthesis as done in [24] in order to capture all the behaviour. It would amount to a RSL equivalent of one large state machine. This would become severely complicated and is out of the scope of this project.

A further undesired property in this approach is that the resulting concurrent specification is rather undesirable with regards to refinement. The usual approach in RSL specification is to start with a applicative specification and through refinement end up with a more design oriented concurrent specification [19]. Starting with a concurrent specification would also make justification diffucult.

The approach described in section 6.7.4 was very interesting considering our own initial goal of using CSP in RSL. However it became evident that we severely lacked the equivalent of an interrupt operator. Semantically the operators we had at our disposal could not be used to model LSCs.

6.7.3 RSL CSP approach

The following subsections describe the thoughts we initially made regarding the construction of a CSP RSL equivalent of LSCs. As it is not used in our final specifications, it may be skipped.

Communication

Communication is done via channels. There should be a dedicated channel for each message to be sent in the LSC in order to lower the risk of deadlock caused by sending several messages on the same channel.

LSCs are modelled as separate parallel processes. In order to synchronise at certain events (i.e. conditions and subcharts) separate processes are created for each of those events. These processes wait for synchronisation signals from the relevant instances and thereafter return a signal with appropriate information on how to proceed. The number of processes may therefore be rather large. But since the specification of them has been made generic it is quite easy to get an overview of all the processes by looking at the instantiated process objects and their parameters.

All communication is done using channels between processes. Schemes have been specified in order to ease the instantiation of channels with correct types.

Synchronisation

Condition processes synchronise the involved instances and use a predicate (*test_condition*) to check whether the condition is true or not. This also has the implication that the condition is only evaluated once, as it should be.

Several scenarios are possible depending on the evaluation and temperature of the condition. If the condition is true, normal LSC execution is resumed. If it is a hot condition that evaluates to false, the LSC behaviour is not satisfied, thus resulting in **stop**. Another interpretation of the semantics of a hot condition in a implemented system, would be to introduce a while loop. This should be repeated until the condition is true. This stems from the fact that a false hot condition is not allowed. On the other hand this could easily lead to a dead locked system.

Interrupt is modelled using an interrupt channel that all instances can send to. The interrupt channel is connected to an interrupt process that distributes the interrupt to all instances, which are to stop. As the system is discrete only one interrupt will be processed.

If it is a cold condition the progression depends on whether the condition is inside a subchart. If it is, the current subchart is exited (all instances involved) and the LSC is resumed right after the subchart end. If the condition is not in a subchart, the main chart is successfully exited and the system may exhibit arbitrary behaviour, **chaos**.

Subchart processes also synchronise the involved instances. Their main function is to determine whether a subchart is to be repeat at the end of it. For that use a variable has been introduced as a counter to maintain track of how often the subchart has been run.

Coregions

Coregion execution is modelled in another function. The reason for this is that a coregion event only consists of message events. It is then (external) nondeterministically determined in which order the events are carried out.

Liveness

When translating the LSC from its diagrammatic notation to RSL the properties of the diagram should still hold. One property is liveness which arguably still holds with the suggested mapping to RSL. Liveness can be achieved through construction; that is, the mapping will result in a specification which has the liveness property.

$\forall$ lsc : LSC • wf_syntax(lsc) $\Rightarrow$ wf_liveness(transltr(lsc))

Communication in a LSC diagram which fulfils the described LSC syntax is always between two processes. At every location of an instance which sends a message there exists a corresponding process and location which is willing to receive the message, and vice versa.

A possible way of showing this formally is considering the next possible locations of a given instance together with the remaining locations and argue that the first location will eventually take place and then proceed with the remainder of the list. Eventually the list will be empty and the instance will have completed successful.

As discussed in section 6.4 we have not introduced timing constructs. However this should be possible given the specification. It would be possible to use Timed RSL and define timing constraints with limited expressibility in relation to certain events. This would also make it possible to create constraints that are global and not only local to an instance. We have not pursued this issue further.

6.7.4 A pure CSP approach

The following approach is a more detailed description of [49] and [50].

CSPs are constructed based on LSCs. This is done by mimicking the states of the equivalent skeleton automaton of a LSC. In charts without a prechart (basic charts) two processes are defined: terminated and aborted to mimic the states of a LSC that has terminated (successful run) or aborted (unsuccessful exit due to a false hot condition). One LSC is defined as a process that is the parallel composition of the processes of the instances. It is noted that operators like if-then-else have their exact images in CSP. If-then-else can be created as an external choice of two guarded processes.

The handling of hot and cold conditions is elegantly done using CSPs interrupt operator [30], which allows for a semantically sound interrupt of a chart or a complete system. The interrupt operator reacts on a special specified event. When for example one instance encounters a false hot condition, this event is emitted and all other instances are interrupted.

The prechart and mainchart are considered two separate processes and are synchronised using a special event. When the prechart finishes the system proceeds as the main chart.

The complete system is then specified as the parallel composition of all the processes of the individual LSCs.

Precharts are monitored continuously in order to see if the main chart must be activated. For each new event a new prechart process is forked to check if the prechart behaviour is exhibited. If unexpected events are encountered, the precharts are discarded. As only regular languages are considered, it is assumed that the activation of the same mainchart never overlaps.

Both universal and existential charts are considered. Existential charts are used to check if they trace refine the universal charts, and can thus be used to check if the initial requirements in form of existential charts hold in the final system.

The final CSPs can then be model-checked using FDR [11] to check for inconsistency.

The following is a list of LSC constructs and whether they are supported by this approach or not.

Multiple charts Multiple prechart activation is allowed, multiple mainchart activation is not.

Charts Both existential and universal

Prechart Yes. In [50] semantics for LSCs with and without precharts is given.

Subcharts Yes. Via process branching.

Locations Only cold locations, as CSP only supports prefix-closed languages.

Messages Hot synchronous messages only. Asynchronous solution sketched in [50]. Cold messages could be modelled with indefinite delay. Asynchronous messages could be modelled with i FIFO buffer that delays messages.

Coregions No.

Conditions Cold and hot. However only conditions on single instances are allowed.

Timing No.

Branching/Iteration Yes.

Actions No. Local computations cannot be specified.

6.8 Example: Applicative RSC

The following is an example of an applicative RSC specification in RSL. It is a rather simple example of a customer withdrawing money from an ATM. It consists of two RSCs that are given in figure 6.19. Furthermore small parts have been added in RSL in order to show the possibilities of augmenting the specification with information that is not conveyed in the diagrams. The complete example can be found in appendix D.6.

6.8.1 RSCs

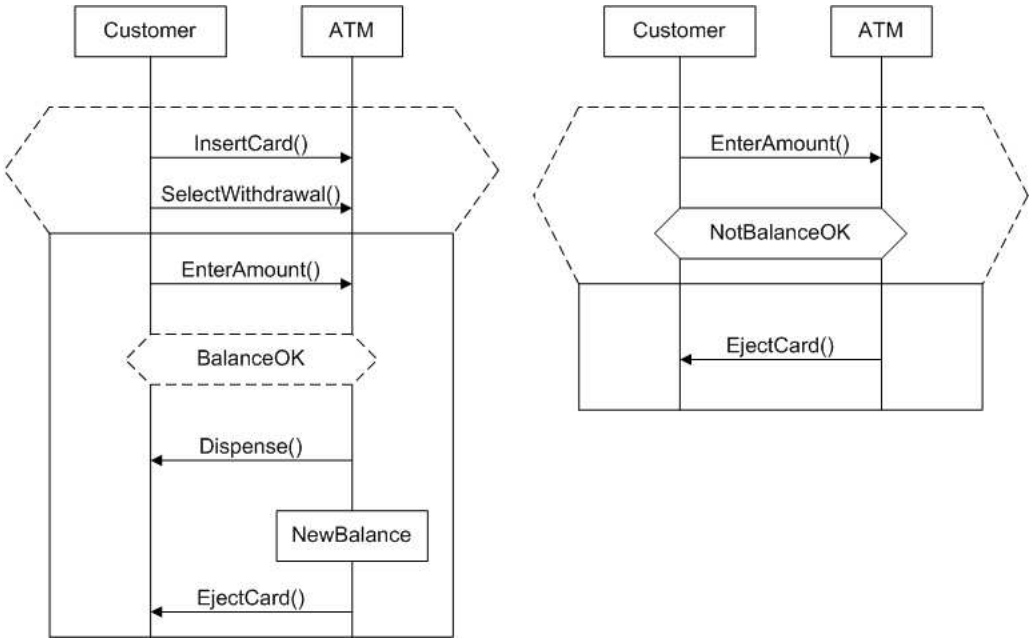

Figure 6.19: The two RSCs that form the basis for the account example.

6.8.2 Specification

The following is the specification of the system. The specification of the two RSCs has been omitted, as it is the same method as used in the example in section 6.6.

The constraints are given by the two RSC's.

value
rscs : T.Collection = {rsc1, rsc2},

The actions, conditions and messages that are present on the RSC's are specified in the respective maps.

actions : **Text** $\overrightarrow{m}$ T.Action =
[″NewBalance″ ↦ new_balance],
conditions : **Text** $\overrightarrow{m}$ T.Condition =
[″BalanceOK″ ↦ balance_ok,
″NotBalanceOK″ ↦ not_balance_ok],
messages : **Text** $\overrightarrow{m}$ T.Message =
[″InsertCard″ ↦ insert_card,
″SelectWithdrawal″ ↦ select_withdrawal,
″EnterAmount″ ↦ enter_amount,
″Dispense″ ↦ dispense, ″EjectCard″ ↦ eject_card],

The following variables have been specified to be used in the system. We only have three accounts in this local ATM.

```
variables : T.Variables =
  [″ATM.amount″ ↦ T.Integer(0),
   ″ATM.balance″ ↦ T.Integer(0),
   ″ATM.cashsupply″ ↦ T.Integer(10000),
   ″ATM.account1″ ↦ T.Integer(100),
   ″ATM.account2″ ↦ T.Integer(64),
   ″ATM.account3″ ↦ T.Integer(500)],
```

The variables that are visible for each chart are specified. Visible variables for a RSC may not be altered by other RSC's as long as its mainchart is active.

```
visible_variables : T.RSC -m-> Text-set =
  [rsc1 ↦
   {″ATM.amount″, ″ATM.balance″, ″ATM.cashsupply″,
    ″ATM.account1″, ″ATM.account2″, ″ATM.account3″},
   rsc2 ↦ {″ATM.amount″, ″ATM.balance″}]
```

The following are the functions that are defined. They consist of the functions prescribed by the RSC and some extra functions that are specified for extending the RSC specification.

The signatures of the functions include the instance-names as specified in the types. They might be used in larger examples where the same message may happen on several instances. In this example this is not necessary and the given parameters are not always used.

balance_ok checks if the current balance stored in the ATM is higher than the requested withdrawal amount.

```
value
  balance_ok : T.Inst_Name-set × T.Variables → Bool
  balance_ok(ins, v) ≡
    if ″ATM″ ∈ ins ∧ ″Customer″ ∈ ins
    then
      T.integer(v(″ATM.balance″)) ≥
        T.integer(v(″ATM.amount″))
    else false
    end,
```

The negation of balance_ok.

```
not_balance_ok :
  T.Inst_Name-set × T.Variables → Bool
not_balance_ok(ins, v) ≡ ∼ balance_ok(ins, v),
```

If the cash is dispensed the variables are updated using the action new_balance.

```
new_balance :
  T.Inst_Name × T.Variables → T.Variables
new_balance(iname, v) ≡
  let
    newbalance =
      T.integer(v(″ATM.balance″)) −
      T.integer(v(″ATM.amount″)),
    newcash =
      T.integer(v(″ATM.cashsupply″)) −
      T.integer(v(″ATM.amount″))
  in
    v †
    [″ATM.balance″ ↦ T.Integer(newbalance),
     ″ATM.cashsupply″ ↦ T.Integer(newcash)]
  end,
```

insert_card retrieves the balance of the account bound to the card that is inserted.

insert_card :
  T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables →
    T.Variables
insert_card(iname1, iname2, v) ≡
  v † [″ATM.balance″ ↦ v(read_account_number())],

Underspecified function of the hardware reading the account number. Note that this function is not part of the RSC's.

read_account_number : **Unit** → **Text**,

Underspecified function for the ATM hardware that a withdrawal has been requested.

select_withdrawal :
  T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables →
    T.Variables
select_withdrawal(iname1, iname2, v) ≡ v,

Function for abstracting the entering of the requested amount via the ATM's keypad.

enter_amount :
  T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables →
    T.Variables
enter_amount(iname1, iname2, v) ≡
  v † [″ATM.amount″ ↦ T.Integer(read_key_pad())],

Underspecfied function for retriving the entered amount on the keypad. Again, this is not part of the RSC's.

read_key_pad : **Unit** → **Int**,

Dispense should tell the ATM hardware to dispense the requested amount. It should read the "ATM.amount" variable and let the hardware dispense the cash.

dispense :
  T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables →
    T.Variables
dispense(iname1, iname2, v) ≡ v,

Underspecified function for abstracting the ATM hardware that ejects the card.

eject_card :
  T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables →
    T.Variables
eject_card(iname1, iname2, v) ≡ v

6.8.3 Complete System

The complete system of the account example is created using the semantics.

scheme system =
  class
    object ACC : account, rsc_check : semantics(account)
  end

6.9 Future work

The work presented here is the current status on RSCs. This means there are definitely grounds for further work on RSCs.

Currently only a subset of LSCs is supported. Future extensions would for example be to include cold locations. However a meaningful semantics must be worked out prior to this. In our current specification this would amount to check that if the events on cold locations happen, they must conform to the partial order induced by the chart.

An obvious alteration would be to include infinite runs. This limitation has its origin in the chart semantics, as the initial plan was to construct an automated simulator, where infinity would not be practical.

The process of refinement of the applicative RSC should also be investigated further. Especially regarding variables. The initial approach is that variables are global. This is not very attractive since RSCs typically may be used to specify distributed systems. As of now it is up to the software engineer to correctly specify variables. A more detailed investigation about how to create local variables would be valuable.

Timing has been omitted as it is a very complicated issue. In order to allow RSCs with timing annotations, the use of Timed RSL/Duration Calculus should be investigated. As of now it is up to the software engineer to introduce timing in the resulting applicative RSC if so desired.

An initial goal was to construct an editor that allowed the drawing, simulation and translation of RSCs. This was not done and it would be obvious to create such a CASE-tool in order to support work with RSCs. It should include automatic generation of the resulting RSL specifications.

It should also be considered that there is an ongoing effort within LSCs and related tools. Integration with a consistency checker as presented in [49] and the Play-Engine [26] would be of great benefit. Unfortunately the lack of a standardised interface language, for example in XML, would complicate this.

More specifically, the applicative RSC system does not consider instance creation at the moment. It is up to the specifier to ensure that created instances are not referenced outside the RSC where the instance is created.

6.10 Conclusion

In this thesis the RSC types and well-formedness conditions have been simplified and completed compared to the preliminary thesis [2]. Furthermore the specifications have been made translatable, allowing testing in order to gain confidence in the correctness. A semantics for a chart was elaborated for use with a RSL equivalent of LSCs.

The initial approach of creating a CSP based RSL version of LSCs was not successful due to semantical problems and that the RSL CSP subset is too small. Others have successfully investigated an approach using purely CSP as described in section 6.7.4. As the CSP interrupt operator is missing in RSL this was not possible using RSL. The goal was to create a specification that allowed augmentation and refinement of the resulting specification. A thing which is not possible in a pure CSP approach. After the failed attempt it was chosen to use an applicative approach instead of the imperative, concurrent CSP approach.

The applicative approach allowed for a more expressive model which allowed the wanted augmentation and later refinement. Thus the LSCs can be used as requirements specifications. These are in nature more high level than design specifications like the CSP approach. It was complicated by the fact that the research in LSCs is very focused about creating executable LSCs, thus going in a more design oriented direction. This is contrary to the initial proposal and foundation in MSCs which are requirements notations. As it is our opinion that RSL is much more suited for the latter, we have followed this approach. This also gives the possibility of following the regular RAISE development method more closely, by going from an applicative specification to an imperative one.

Since a CASE tool was not developed there is not much benefit besides testing in the translatable specifications. This brings us to the next point. In order to be useful the current model must be supported by an integrated CASE tool. It should allow translation back and forth between RSC s and the equivalent RSL specifications. Currently it is too cumbersome to use. If this tool could be developed and the issues mentioned in section 6.9 could be resolved, RSCs could become a viable choice for specifying inter-object communication in RSL.

Part III

Concrete implementation

Chapter 7

Introduction

In order to showcase the achievements presented in chapter 5 a concrete implementation was constructed. Regarding Scheme Diagrams the main goal was to construct a CASE-tool that gives the ability to actually create Scheme Diagrams. The idea was to create an editor that allowed to graphically construct Scheme Diagrams and allowed printing of a concrete RSL specification. The back end was to be formed by the translatable Scheme Diagram specifications.

The following chapters will proceed as follows: We will describe the process of selecting language and GUI library for the implementation in chapter 8. We will present the candidates and give a rationale for the final selection.

This is followed by chapter 9 discussing the actual system that has been implemented. A short overview of the structure is presented followed by a description of the individual parts that make up the system. The emphasis is on the structure of the program. A more technical walk through has been omitted, as there is extensive technical documentation on the companion CD, appendix E.

Chapter 8

Language and library

Contents

8.1 Requirements 97
8.2 Candidates 97
8.2.1 Java:Eclipse+GEF 97
8.2.2 C++: wxWidgets 98
8.3 Selection and rationale 99

8.1 Requirements

The first initial requirement for our implementation was the ability to run under both Linux and Windows environments in order to reach a broader audience with the final program. This was also a consequence of the fact that the *rsltc* tool is available for both Linux and Windows. In order to minimise cross-platform problems Java was deemed the preferred choice of language. The other contemporary program-language we considered as suitable as well, was C++. This mainly stems from the fact that the RSL specifications initially only could be translated to C++. More on this in 9.3, see also previous section 5.4.4. A recent master thesis [29] has come up with a similar translator from RSL to Java. The implementation is however very rudimentary. As an example maps and let expression have not yet been incorporated. The use of C++ for the editor would also lead to the need for two separate compilations, one for each platform. However this is still needed for our translatable Scheme Diagram specification, which forms the back-end regarding our model.

In order to minimise the time required to implement the tool, a definite requirement was to use an already existing framework for graphical editing. This should include ready-made graphics libraries. The framework should also be supported by a rather large community. This would ensure future development, bug fixes and a larger probability of help within forums and the like.

8.2 Candidates

Based on the requirements in the previous section we narrowed down the search to two frameworks, one for C++ and one for Java. We will now describe the benefits and drawbacks of each of them.

8.2.1 Java:Eclipse+GEF

Generally speaking Java is a well known platform independent programming language developed by Sun Microsystems, see `http://java.sun.com`. Eclipse [46] is a recent Java-based open source project, maintained by IBM, developing a highly modularised tool platform. It is a universal IDE which can be extended by

virtually any editor/tool. The focus is on tool integration and the concept of plug-ins, extensions that enhance the functionality of the Eclipse IDE.

Benefits:

- There exists a large library with all the common operations like manipulating figures where already implemented.
- It has already proven itself to be platform independent and we will not have to pay any particular concern about any platform specifics.
- Eclipse in itself is a Java IDE and provides features such as on-line documentation, code completion and debugging.
- Eclipse is especially suited for developing editors as plug-ins. A complete framework for creating new plug-ins is provided along with some examples.
- In its nature Eclipse allows for easy implementation with other tools (e.g. RSL syntax checkers) in the future.
- GEF (Graphical Editing Framework) is present. As the name suggests, it is a framework for creating graphical editors. It comes with examples as well. GEF is based on Draw2d, which is another plug-in for Eclipse, that provides libraries for graphical figures, their manipulation and display.
- Eclipse has a rather large community with active forums.

Drawbacks:

- Performance-wise, Java applications tend to be slow.
- During testing it was evident that Eclipse was very memory-consuming.
- The approach with plug-ins is quite complex, as the whole Eclipse framework must be mastered in order to be able to create plug-ins.
- In our case Eclipse would require attaining knowledge not only about Eclipse itself, but also GEF and Draw2D.
- Eclipse is still under development. This could lead to deprecated methods and structural changes in the framework which would require alteration of the implementation later on.
- The Eclipse documentation is clearly not finished and lacks comprehensive explanations and examples. The same is especially true for GEF.
- A major drawback would be that the use of JNI (Java Native Interface) would be required. JNI allows the use of C++ compiled libraries from Java programs.

8.2.2 C++: wxWidgets

C++ is another contemporary programming language. The only complete graphical editing framework that was found was wxWidgets (formerly known as wxWindows) is a cross platform GUI library for C++. It has been used by many mainstream programs such as:

VideoLAN A cross-platform multimedia player supporting a large variety of codecs, see `http://www.videolan.org`.

Audacity A fast and powerful audio editor, see `http://audacity.sourceforge.net`.

Compared to Eclipse, wxWidgets only consists of the library, no framework for tools is included.

Benefits:

- The resulting program will be platform native program and is thus likely to have higher performance.
- Object Graphics Library (OGL) is available. OGL is a ready-to-use library with predefined shapes, connectors etc. which should be very useful in creating diagrams.

- The general approach is easier since there is no framework to getting used to.
- The choice of IDE is free.
- RSL program could easily be integrated, as it itself is translated to C++.
- wxWidgets has a large community with active forums.
- wxWidgets has been under development for 11 years, and thus must be considered quite stable.
- Good documentation.

Drawbacks:

- The complete editor must be build from scratch, thus requiring more programming.
- We are likely to encounter more problems with regard to making the program work on different platforms.

8.3 Selection and rationale

Considering the above benefits and drawbacks we chose to use the Eclipse framework, based on the following rationale:

- Eclipse is an open-source, fully integrated IDE for Java. This would eliminate the need for finding a C++ IDE. An Eclipse project works on a C++ IDE plug-in as well, however it is not as developed as the Java IDE.
- The plug-in approach would mean that a lot of the basic functionality would be provided, thus eliminating a lot of the programming effort needed. This was weighted against the increased effort needed to learn the framework. The examples and tutorials were deemed sufficient for attaining a steep learning curve.
- Ready-made examples of a basic UML class diagram editor were available. This could kick-start the development of our tool, since the Scheme Diagrams have many similarities with a UML class diagram.
- By using Java, platform independence would be obtained. This was weighted against that we had to use an interface library in order to use the translated RSL specification. As we had a closer look at JNI, we anticipated it would be rather hassle-free to integrate the resulting RSL C++ library with the Java code.
- As Eclipse is very modular, it would be easy to extend the tool in the future with other editors.
- Other factors such as large community, where deemed to be satisfactory for both options.
- Performance was not considered an important aspect, as our tool is a proof of concept.

Chapter 9

System description

Contents

9.1 Overview 101
9.2 Eclipse plug-in 102
9.2.1 Requirements 102
9.2.2 Eclipse 103
9.2.3 GEF 103
9.2.4 Draw2D 105
9.3 Eclipse Scheme Diagram Editor 105
9.3.1 Limitations 106
9.4 Imperative RSL model specification of Scheme Diagram 107
9.4.1 Translation from RSL to C++ 107
9.4.2 Imperative specification 108
9.5 Gluing the Eclipse plug-in and the RSL model together 108
9.5.1 JNI 108
9.5.2 Type conversion 109
9.5.3 The Interface 109
9.6 Test 109

9.1 Overview

The choice presented in the previous chapter required the creation of a new editor plug-in for Scheme Diagrams using GEF and Draw2d in Eclipse. Furthermore it required JNI for integrating the Java plug-in with our translated RSL specifications described in the previous part. The system would thus consist of three parts:

- The RSL specification translated to C++ and compiled as a library. The RSL specification had to be made imperative in order to allow variables to store the model that is being worked on.
- The JNI interface describing the methods needed to call the C++ methods from Java.
- The Eclipse framework that provides the front end and a business model (described later) representation of the Scheme Diagrams using GEF and Draw2D.

These three parts are presented in the following three sections respectively. As they are quite independent they can be arranged and read in any order. In the next section we start by describing the requirements for the tool, Eclipse and the accompanying framework. We then describe our own plug-in structure in more detail. The following section presents the imperative RSL specification and the resulting library. The last section finishes off with the integration of the two.

Figure 9.1 gives an overview of the relation between the three parts.

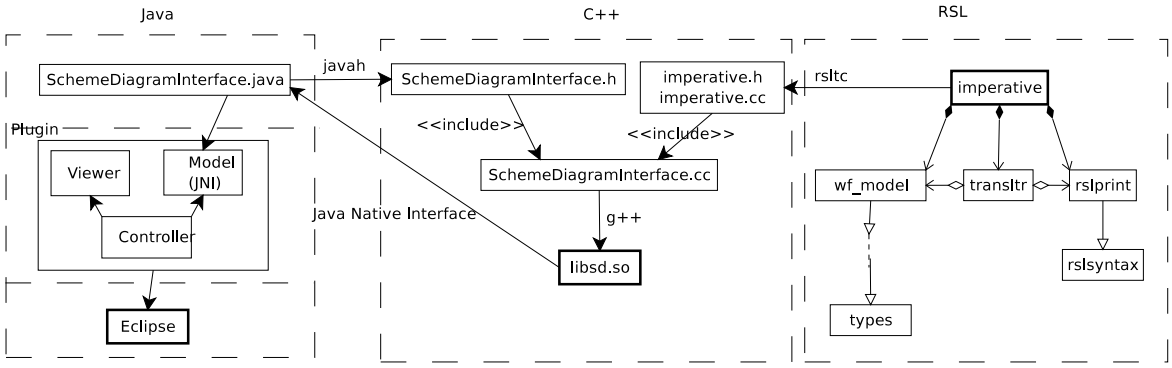

Figure 9.1: The implementation consists of three parts. The left part is the Eclipse part in Java. It consists of the Eclipse Framework and the ESDE plug-in in Java which conforms to the Model View Controller paradigm (explained later). It uses the SchemeDiagramInterface *to connect to the middle part, which is in C++.* Javah *is used to create the* SchemeDiagramInterface.h *header file that corresponds to the* .java *file. The C++ part consists of the interface specification in SchemeDiagramInterface.cc and the resulting C++ code from the specifications,* imperative.cc*. They are compiled using the* g++ *compiler. The specifications on the right form the back end. As indicated, the* rsltc *tool is used to translate the specifications to C++. The right part is displayed using Scheme Diagram notation. The rest is a pseudo notation for indicating the relationships between the various entities.*

9.2 Eclipse plug-in

9.2.1 Requirements

In this section we will shortly present the high level requirements for ESDE. This includes requirements for the graphical user interface. However it excludes any technicalities to the syntax and semantics of the contents of the diagrams, which already have been described in previous sections. Much of the requirements where inspired by already existing UML Class Diagram editors, e.g. Poseidon UML [14]. Regarding the overall GUI it is already given by the Eclipse framework. Other requirements where as follows:

- A canvas for the diagram where the graphical objects can be added/removed/manipulated.
- An outline view which shows an overview of the complete diagram.
- Buttons for changing the layout/appearance of the diagram.
- A palette with access to tools for adding elements.
  - Adding of schemes/modules
  - Adding of relations, e.g. association
  - Adding of values, variables etc.
- Context menu for deleting elements
- Functionality for changing element names/values
- Undo/redo functionality
- Functionality for checking well-formedness
- Functionality for printing the equivalent RSL specification to files
- Functionality for saving the Java-based model.

There are several ways to let the user create a model in the diagram. We will adopt the model of letting the user create the diagram and only syntax check the model upon request request. It is a consequence of the fact that it is not possible to edit a diagram while maintaining it well-formed all the time.

Furthermore, some of the well-formedness conditions will be enforced by the graphical user interface. As an obvious example it should not be possible to connect the same line to more than two containers.

9.2.2 Eclipse

As mentioned earlier Eclipse is a highly modularised tool platform. In this section we will present an abstract overview of the Eclipse structure. For further information on Eclipse we refer to [46].

Figure 9.2 illustrates how Eclipse is structured. The Eclipse platform provides the basic functionality in form of a workspace with all the global settings and a workbench which makes use of an editor. The two plug-ins on the left which are provided with Eclipse are the JDT, which is a Java IDE and the PDE, which is used in order to create new plug-ins within Eclipse. The right side tools describe that new plug-ins may be supplied to the framework. Our plug-in is such an addition. It provides the facility of a Scheme Diagram editor.

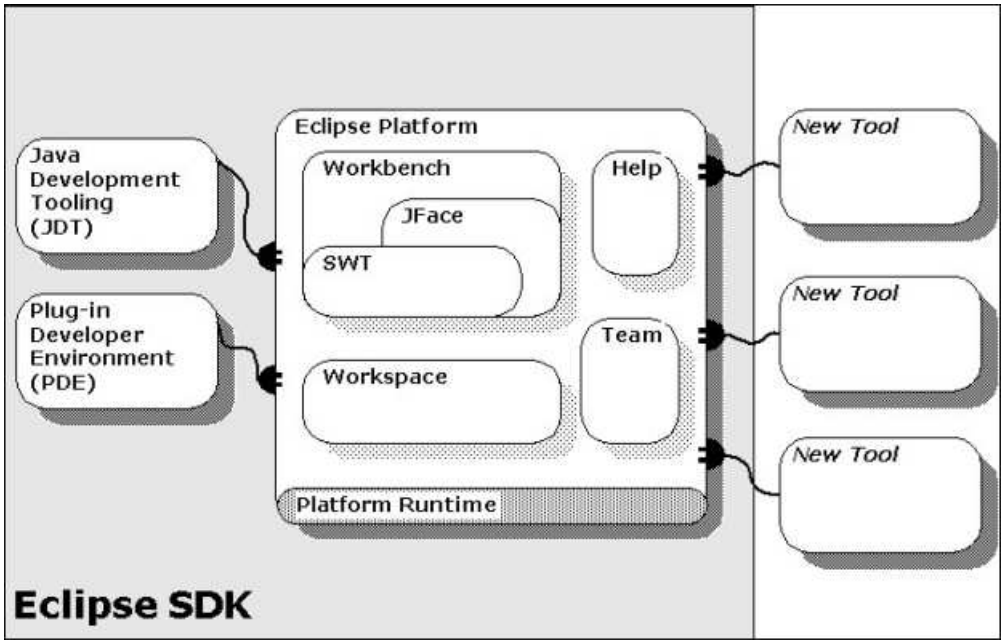

Figure 9.2: Overview of the Eclipse system from [8].

9.2.3 GEF

GEF (Graphical Editing Framework) is a framework primarily for creating graphical editors in Eclipse. It is an editor framework built on top of Draw2D (see section 9.2.4). It supplies a complete predefined structure of new editors. This means that the structural part of our plug-in was given beforehand.

Figure 9.3 is an illustration of the structure of GEF. The top part denotes the so called workbench page, which corresponds to the workbench on figure 9.2. It provides the interface to the rest of the Eclipse framework. The Editor-part is the main part that is responsible for the editing. It will be elaborated below. A palette-viewer is provided which allows for creating a palette with tools for editing. The edit domain is supplied and automatically handles such things as the command stack which allows for undo/redo actions. The edit part viewers are responsible for the actual diagrammatic representation in the editor. In our editor there will only be one, as we only have one view of our model. The outline view is supplied in order to allow for an outline of the complete editor canvas. This gives a quick overview of the displayed figures.

In order to understand the above better, we will now describe how a GEF editor works. GEF works with a model view controller paradigm which is illustrated in figure 9.4. The idea is that the model is completely separated from the graphical representation, the view. The controller is used for controlling changes in the model and for correspondingly updating the view.

In our case the model is the Scheme Diagram model provided in RSL and is the actual data and relations we want to edit and display graphically. In order to create the linking between the RSL model and the plug-in, we worked with the concept of a business model. A business model is an extension to the existing model with information such as placement of boxes and arrows. This is needed since such information is not present in the RSL Scheme Diagram model. A not well-formed model may be edited. This is a consequence of the fact that

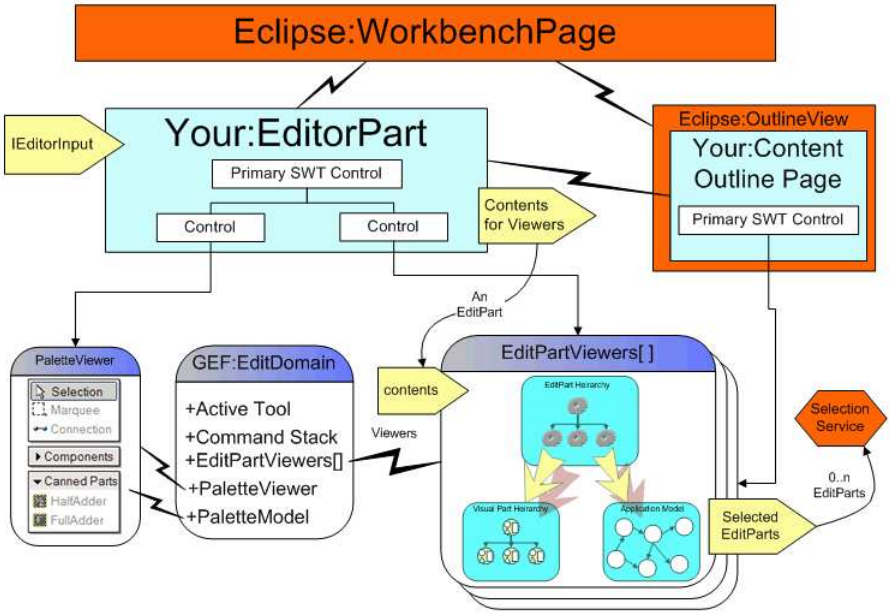

Figure 9.3: Overview of the GEF structure from [8].

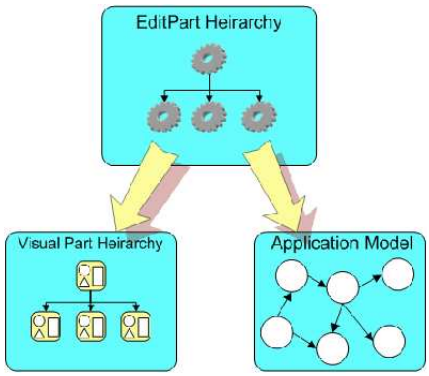

Figure 9.4: Overview of the Model View Controller paradigm from [8].

the model cannot be well-formed at all times when being edited, e.g. when adding global objects. This means that the Scheme Diagram is not checked for well-formedness continuously. The user is given the possibility to choose when to check the model. When editing a diagram, it will for the most part not be well-formed until the user has finished drawing it. Another reason for this was performance, as the editor would have a very sluggish response if all information in the model had to be checked upon each alteration. The model itself is not concerned with how the model is displayed graphically. It simply holds information about the structure of the model.

The controller is called *EditorPart* in GEF. This is the control structure that determines everything regarding the editing. It handles user input and manipulates the model accordingly.

The view (called *VisualPart*)is concerned with the actual graphical rendering of the model. How the boxes look like, how their names are displayed etc. The graphical representation is in our plug-in done using Draw2D figures, see below.

The main idea of this approach is to have a clean separation of the three parts. This allows for easy alteration in each part, as they are very independent. For example a completely different graphical representation of the model could be used by only changing the view part.

9.2.4 Draw2D

Draw2D is a painting and layout plug-in for Eclipse. The structure of Draw2D is given in figure 9.5. The developer must provide the figure hierarchy, the remaining boxes in the figure are the internals of Draw2D. The developer must define how each figure is drawn. The Draw2D framework supplies a canvas with corresponding event handling, actual drawing of the figure, etc.

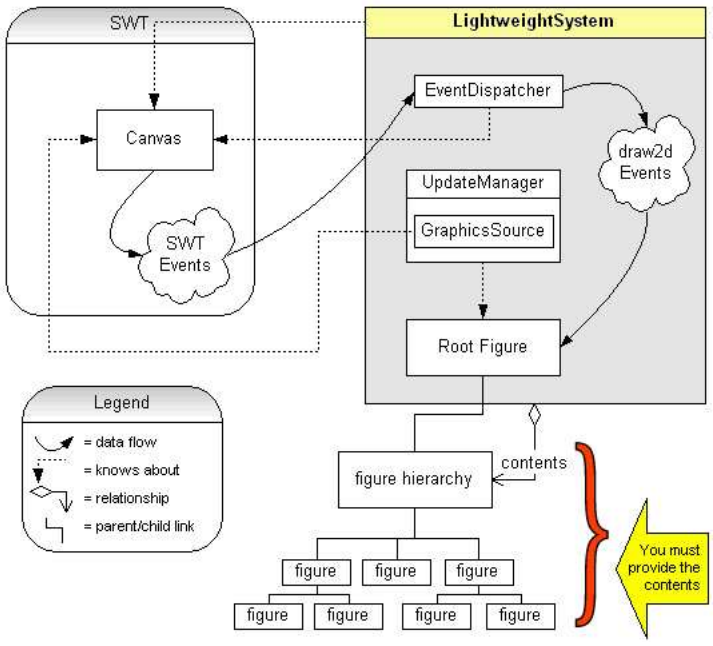

Figure 9.5: Overview of the Draw2D structure from [8].

Draw2D provides predefined figures in form of different shapes, arrows etc. These are easily created without having to worry about the internal structure of Draw2D. Auxiliary classes are also provided for the figures, which e.g. influence the layout of the figures in the canvas. As an example there are *ConnectionRouters*, which automatically routes a connection between two points on the canvas without overlapping with other figures.

9.3 Eclipse Scheme Diagram Editor

The tool is named ESDE, standing for Eclipse Scheme Diagram Editor. In this section we will give a short introduction to the structure of ESDE. Detailed comments on the methods and variables are supplied as JavaDoc

comments in the source code, which can be found on the companion CD-ROM, appendix E). Therefore we will not describe the actual Java code in any detail here.

The plug-in is heavily inspired by two examples. Mainly [53] where the framework fitted our intended plug-in well. The other example is regarding Draw2D. It shows how to use Draw2D to create a (very basic) UML class diagram [37]. The appropriate author is listed in all source code files. In files where we have only supplied additions/alterations we have listed ourselves as authors together with the original author. In files which where only inspired by the examples we have listed ourselves as authors. We have added JavaDoc comments in all files, as these where sparse in the examples.

We will now give an description of the packages that the plug-in consists of:

rsl.esde.action Classes for the actions associated with buttons in the workspace. Only actions which are not standard, i.e. delete, must be coded.

rsl.esde.command Classes for handling user commands on the canvas, i.e. creating a scheme or renaming a connection.

rsl.esde.directedit Helper classes responsible for direct editing of text fields (labels), i.e. the name of a scheme.

rsl.esde.dnd Classes handling a drag and drop on the canvas.

rsl.esde.editor The main editor classes which provide the start up. Including the outline view and the palette with tools.

rsl.esde.figures Classes for creating the displayed graphical figures. Correspond to the view part of the MVC paradigm. Uses Draw2D.

rsl.esde.layout Classes handling layout. It is possible to use automatic and manual layout.

rsl.esde.misc Two classes from Draw2D that had to be modified to work in our framework.

rsl.esde.model Classes for the model elements. Corresponds to the model part of the MVC paradigm.

rsl.esde.part Classes of editor parts that are responsible for performing alterations etc. Corresponds to the controller part of the MVC paradigm.

rsl.esde.policy Policies that govern which operations are permitted in an editor part. Responsible for creating commands and initialising them correctly.

A screen shot of the editor is presented in appendix F. It is annotated with descriptions of the various elements that fulfil the requirements mentioned earlier.

Basically the editor is used to draw the desired diagram. While it is edited it is **not** checked for well-formedness. This is chosen as the model will usually not be well-formed while it is being edited. Instead buttons have been supplied that allow the user to check the well-formedness at will. Via a button the equivalent translated RSL specification can be saved to a chosen location in `.rsl` text files.

9.3.1 Limitations

ESDE has some limitations compared to what is possible in the Scheme Diagram model. We will describe these and give a motivation for the choices.

Objects in the Scheme Diagram model have a state comprising of the variables that are present in the scheme which the object is an instance of. Clearly the object must therefore define those variables. In ESDE we have chosen not to model this. When the well-formedness of a Scheme Diagram is checked, the state is automatically added to objects, as can be seen in (*wf_model*, C.3.1). Ideally ESDE should automatically update the variables of objects depending on the variables of the scheme the object is an instance of. It does not make sense to model the values of these variables as the model is static. It is a consequence of the current implementation, which has a clear cut between the plug-in and the RSL C++ model. It would be possible if the model was implemented in Java. Then there would be tighter integration of the model and the diagram.

There are also limitations with regards to the type declarations and type expressions that are possible to enter in schemes. The problem stems from the JNI interface and the building of the RSL model when checking and

printing a diagram. The input in the diagram must be parsed in order to be able to create the corresponding RSL type declarations and type expressions. This is a consequence of the fact that the type declarations and type expressions are stored as text in the Java model. As the building of a real parser was outside the scope of this project, regular expressions have been used as a quick solution. They are used to validate the input in the diagram and parse the types and expressions when they are added to the RSL model. Regular expressions have been used as a light solution. They are not fully applicable since recursive expressions cannot be specified. A better solution is to create a parser perhaps inspired by the source code of *rsltc*. An alternative is tighter integration with *rsltc* which could be used to do real time parsing.

The syntax is given in appendix F. As regular expressions do not allow recursion, it is only possible to create a subset of the constructs possible in the Scheme Diagram model. E.g. it is not possible to create a type expression that consists of both a Cartesian product and a function. Furthermore access descriptions have not been included as of yet. Type declarations can only be sorts or abbreviation definitions.

9.4 Imperative RSL model specification of Scheme Diagram

9.4.1 Translation from RSL to C++

From early on in this project we decided to use automated code generation from the RSL specification. This had some advantages: our formal specification could be tested and the implementation is more likely to do as specified. Furthermore a large part of our system was ready right away. Otherwise we had to implement the model in a program language ourselves based on the specification.

There where also some disadvantages: It was not possible to make changes to the auto generated code for two reasons: Any changes would be erased if there was made a change to the formal specification and generated the C++ source code again. The generated code was not easy to read thus it would also be cumbersome to make changes. The specifications had to be modified so they would be translatable, thus some abstraction was lost. Furthermore it was extremely cumbersome to debug the translated code.

If we were to consider the modifications to the specification as a refinement and thus maintain two versions, then it had been basically just extra work. After briefly looking at the scope of the changes that had to be made in order to translate into C++ it was evident that some of the elegance of the original specification was to disappear. In particular union definitions had to be converted to variant definitions which means a lot of constructors and destructors. Another problem were curried functions, which had to be converted to a function with one formal argument by making one large Cartesian product.

Despite the disadvantages we thought that the benefits were significant enough to pursue. This was also mentioned several times at the conferences we attended (and later seminars, appendix H): Changes to specifications with automated generation of the underlying specifications will significantly reduce maintenance cost (time in our case).

Regarding “refinement” of RSL syntax in order to create a translatable specification, the following issues where addressed:

- Unions cannot be used due to shortcomings of the C++ translator. However equivalent variant definitions with appropriate constructors and destructors solve this problem. Semantically this will not change the specification.
- In order to ease the rewriting of the union definitions, an emacs macro was used in order to automate the process a bit.
- The rewriting has a notable disadvantage as the specification becomes a lot harder to read and understand. However the macro uses a “standardised” way of naming constructors and destructors, this should ease the understanding of the variant definitions, see below.
- The C++ translator was initially used with gcc 2.95.2. When we tried to compile the specification with gcc 3.3.X we encountered several errors which were solved when downgrading to 2.95.2. A fix to the *rsltc* tool was applied to resolved this, so that the specifications can now be compiled using the newer gcc 3.3.5

- Nonetheless a couple of errors were presented by the compiler. By debugging it was found that the following RSL specification will produce an error. We encountered them at places where union definitions had been rewritten to the equivalent variant definitions.

  type1D2D == type1D2D_from_type1D(type1D2D_to_type1 : type1D) |
  type1D2D_from_type2D(type1D2D_to_type2 : type2D),

  type1D = **Int**,
  type2D = **Int** × **Int**,

  I.e. when a type used in a variant definition uses a product type expression (as in type2). There were 2 solutions to alter the specification in order to make a version that compiles. The first is to remove the type_1D2D_to_type2D destructor. This would mean that a case expression may be needed instead of a destructor if the type2D value was to be extracted. The other solution we found was to rewrite type 2 as a short record definition:

  type2 :: **Int Int**,

  However this is semantically not quite the same, since this construct introduces a sort type2D (due to the implicit variant definition) which initially is not the case. The intended meaning that the second type in the type1D2D variant definition is composed of 2 Ints is however preserved.

  This issue was later resolved by a bug fix after a consultation with the author of the *rsltc* tool [9].

9.4.2 Imperative specification

In order to use the specifications in conjunction with the plug-in we had to make the specification imperative. It is supplied in appendix C.3.1.

A variable is used to store the model in the abstract RSL syntax. It was created in order to allow the creation of interface functions for adding schemes, relations etc. via the JNI interface. This allowed for the checking of well-formedness and printing as text.

9.5 Gluing the Eclipse plug-in and the RSL model together

9.5.1 JNI

JNI is the native programming interface for Java that is part of the JDK. The JNI allows Java code that runs within a Java Virtual machine to call methods in native libraries. In our case the translated and compiled Scheme Diagram specifications. Figure 9.6 illustrates this relationship.

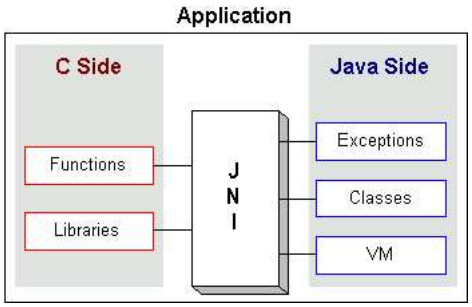

Figure 9.6:

9.5.2 Type conversion

The biggest limitation by using JNI is the restriction regarding parameter types and return types. Only int, double and boolean (Java) types are directly supported, since Arrays, Strings, etc. are objects in Java. For string a special conversion function must be supplied. It is given in C.3.5. In order to avoid arrays the interface functions have been specified in order to allow recursive calls.

9.5.3 The Interface

The interface to call in ESDE is supplied in SchemeDiagramInterface.java, see appendix C.3.2. The methods are declared as native in order to let the Java Virtual Machine know that the methods are supplied in an external native library. The name of the library to load is also specified, in this case **libsd.so**.

Included in the JNI package for Java is the tool **javah**. Based on SchemeDiagramInterface.java it generates the equivalent C++ header file (C.3.3) with the appropriate signatures. The corresponding C++ file was then created. It had to specify which C++ functions in the translated C++ Scheme Diagram where to be called (C.3.4). Our naming convention in the .java file followed Java conventions with capital letters, e.g. addScheme. In the .cc file we used the C naming convention with underscores, e.g. add_scheme. This also helped in distinguishing where the functions belonged.

The following is an example of the functions that add a scheme to the model:

Listing 9.1: SchemeDiagramInterface.java example

```
public native boolean addScheme(String name);
```

Listing 9.2: rsl_esde_libsd_SchemeDiagramInterface.h example

```
/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addScheme
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addScheme
  (JNIEnv *, jobject, jstring);
```

Listing 9.3: rsl_esde_libsd_SchemeDiagramInterface.cc example

```
JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addScheme
    (JNIEnv *env, jobject obj, jstring name)
{
  return add_scheme(jstr2rsl(env, name));
}
```

9.6 Test

When developing the specifications they where primarily tested using the newly introduced **test_case** clause in RSL. When ESDE was finished we decided to test ESDE and the imperative RSL specification at the same time.

This was done by creating several Scheme Diagrams as test cases in ESDE. Both well-formed and not well-formed diagrams were created in order to create confidence in the translation and well formedness conditions. All the test examples can be found on the companion CD, appendix E.

Equivalent RSL specifications (as `.rsl` files) of representative diagrams have been included in appendix C.4 in order to show the correct functioning of the RSL print specification.

Part IV

Postlude

Chapter 10

Conclusion

Isaac Newton: If I have seen farther, it is by standing on the shoulder of giants. ∎

An important aspect of this master thesis has been to build upon the work of others. This is the case for both of the diagrams presented in part II and also for the concrete implementation presented in part III.

The Scheme Diagram presented in this thesis is based on [2] and considerable advancements have been achieved. The diagram has reached a state where it is practically usable as demonstrated by the examples and the implementation. There are still open issues as mentioned in section 5.8, but none have a fundamental impact on the soundness of the diagram. In general the mapping from the diagram to RSL is sound and the inspiration from the Class Diagram has given the diagram a presentable form.

A functional implementation of the Scheme Diagram editor is presented. It demonstrates the convenience of graphical development of specifications. Additionally the diagram can be used as documentation. The tool should, however, be considered proof of concept because the underlying model is based on the translated Scheme Diagram specification. The design of the implementation allows for the model to be substituted with an implementation in Java without altering the graphical part. Hence there is a basis for improvement. The tool could be used by existing RSL users as a supplement, but could also be used to introduce newcomers to the language. Especially users with a background in UML could benefit from an intuitive understanding of RSL through the diagram. A translator from RSL text to the Scheme Diagram would be a considerable addition to the CASE tool. It would allow the diagram to be a continuing part of the development process and not just an initial aid.

The RSC presented is also based on [2]. The syntax and semantics have been considerably rewritten and made translatable. The initial approach of using the CSP subset present in RSL to model the inter-object behaviour of RSCs was found not to be semantically sound. Instead an applicative approach resulted in a much more clean and usable solution. It has been showed that RSCs may be used as a point of departure for further refinement of inter-object behaviour. However, the current status of RSCs definitely needs CASE-tool support in order to be useful. Initially the plan was to construct a simulator/editor for RSC during this thesis. Regarding the simulator part it was deemed superfluous, since the already implemented Play-Engine [26] is useful for that purpose. Regarding the editor the implementation of a Scheme Diagram editor was prioritised higher, as the achievements with Scheme Diagram where more promising.

In general terms, LSCs do not seem to have reached a sufficient level of maturity. There are still many open questions regarding the semantics. This is also worsened by the current research effort of creating various versions of executable LSCs. These issues should be resolved before continuing the effort towards the integration of LSCs/RSCs with RSL.

We have reached the conclusion that RSL was not suitable for translation in our project. The first reason is that the *rsltc* tool is incomplete regarding translation to C++ and restricts the usage of RSL. The second reason is that the tool has a Graphical User Interface (GUI) and consequently requires interaction. Well-formedness is for both diagrams specified as predicates, which do not indicate reasons to violations. An implementation of the Scheme Diagram syntax and well-formedness in Java using the RSL specification as requirements, would be a better approach. It could for example produce better error messages, since features such as exceptions could be used. A third and more general reason is that debugging the RSL specification is difficult, since it is actually the translation to C++ that is debugged. This is not intended to be human readable.

In this thesis we have shown that it is possible to use graphical notations in conjunction with RSL with promising results. There are definitely grounds for further research in that direction.

Bibliography

[1] Satyajit Acharya and Chris George. Specifying a Mobile Computing Application Environment Using RSL. Technical Report 300, UNU-IIST, P.O.Box 3058, Macau, May 2004.

[2] Steffen Andersen and Steffen Holmslykke. From UML to RSL – and back again!, July 2004.

[3] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel. Formalization of the Whole-Part Relationship in the Unified Modeling Language. *IEEE Trans. Softw. Eng.*, 29(5):459–470, 2003.

[4] Hanene Ben-Abdallah and Stefan Leue. Timing constraints in message sequence chart specifications. In *FORTE X / PSTV XVII '97: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII)*, pages 91–106. Chapman & Hall, Ltd., 1998.

[5] Dines Bjørner. *Software Engineering*. Springer-Verlag, Berlin, 2005. To be published Spring 2005.

[6] Yves Bontemps and Patrick Heymans. Turning High-Level Live Sequence Charts into Automata. Technical report, Univ. of Namur - Computer Science Dept, March 2002. `http://www.info.fundp.ac.be/~ybo`.

[7] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.

[8] Eclipse.org. *Help Eclipse Platform*. Built-in Help and Documentation for Eclipse 3.0.1.

[9] Chris W. George et al. RAISE tool. http://www.iist.unu.edu/newrh/III/3/1/page.html.

[10] Design for Validation. Esprit long term research project no. 20072. `http://www.newcastle.research.ec.org/deva/`.

[11] Formal Systems (Europe) Ltd. `http://www.fsel.com`.

[12] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Comput. Stand. Interfaces*, 19(7):325–334, 1998.

[13] Ana Funes and Chris George. Formal foundations in rsl for uml class diagrams. Technical report, UNU/IIST, May 2002. Report No. 253.

[14] Gentleware. Poseidon for UML. `http://www.gentleware.com`.

[15] Chris George and Søren Prehn. The RAISE Justification Handbook, May 1994.

[16] Chris W. George. Discussion with author in person and via mail.

[17] Object Management Group. OMG Unified Modeling Language Specification, version 1.5. `http://www.omg.org/cgi-bin/doc?formal/03-03-01`, March 2003.

[18] The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall International (UK) Ltd, 1992.

[19] The RAISE Method Group. *The RAISE Development Method*. Prentice Hall International (UK) Limited, 1995.

[20] N. Guelfi, O. Biberstein, D. Buchs, E. Canver, M-C. Gaudel, F. von Henke, and D. Schwier. Comparison of object-oriented formal methods. Technical report, University of Newcastle Upon Tyne, Department of Computing Science, 1997.

[21] Gonzalo Génova, Juan Llorens, and Paloma Martinéz. The meaning of multiplicity of n-ary associations in UML. Published online, 2.December 2002. Springer-Verlag.

[22] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[23] David Harel and Eran Gery. *Executable Object Modeling with Statecharts*. IEEE Computer, Vol.30, No. 7, pp. 31-41, 1997.

[24] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *CIAA '00: Revised Papers from the 5th International Conference on Implementation and Application of Automata*, pages 1–33. Springer-Verlag, 2001.

[25] David Harel and Rami Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, page 193. IEEE Computer Society, 2002.

[26] David Harel and Rami Marelly. *Come, Let's Play - Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag Berlin Heidelberg, 2003.

[27] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[28] J. He, Z. Liu, X. Li, and S. Qin. A Relational Model for Object-Oriented Designs. In *Pro. APLAS'2004, Lecture Notes in Computer Science*, Taiwan, 2004. Springer.

[29] Ulrik Hjarnaa. Translation of a subset of rsl into java. Master's thesis, Technical University of Denmark, November 2004. Masters Thesis.

[30] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[31] Steffen Holmslykke. Analysis of UML Class Diagrams and a Formal Model, February 2004.

[32] I-Logix. `http://www.ilogix.com/`.

[33] IFAD. The Rose–VDM++ Link. Technical report, IFAD, Forskerparken 10A, DK-5230 Odense M, 2000. Revised for v6.6.

[34] ITU-T. ITU-T Recommendation Z.120: Message Sequence Charts (MSC). Technical report, ITU-T, 1996.

[35] Yang Jing, Long Quan, Li Xiaoshan, and Zhiming Liu. A Predicative Semantic Model for Integrating UML Models. In *Proceedings of the 1st International Colloquium on Theoretical Aspects of Computing (ICTAC)*, 2004.

[36] Jochen Klose and Hartmut Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–527. Springer-Verlag, 2001.

[37] Daniel Lee. Display a UML Diagram using Draw2D, 2003. `http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html`.

[38] Morten P. Lindegaard and Anne E. Haxthausen. Proof Support for Raise by a Reuse Approach based on Institutions. Informatics and Mathematical Modelling, Technical University of Denmark.

[39] Jing Liu, Zhiming Liu, He Jifeng, and Xiaoshan Li. Linking UML Models of Design and Requirement. Technical report, UNU/IIST, February 2004. Report No. 293.

[40] Zhiming Liu, Xiaoshan Li, Jing Liu, and He Jifeng. Integrating and refining UML models. Technical report, UNU-IIST, March 2004. Report No. 295.

[41] Christian Krog Madsen. Integration of Specification Techniques. Master's thesis, DTU, 2003.

[42] Christian Krog Madsen. Study of Graphical and Temporal Specification Techniques. Technical report, DTU, Nov 2003.

[43] Rami Marelly, David Harel, and Hillel Kugler. Multiple instances and symbolic variables in executable sequence charts. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–100. ACM Press, 2002.

[44] Rami Marelly, David Harel, and Hillel Kugler. Specifying and executing requirements: the play-in/play-out approach. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 84–85. ACM Press, 2002. Previously: Technical Report MCS01-15, Mathematics & Computer Science, Weizmann Institute Of Sience.

[45] A. Pneuli and R. Rosne. Distributed reactive systems are hard to synthesize. In *IEEE Symp. on Foundations of Computer Science*, volume 2, pages 746–757, 1990.

[46] Open Source Project. Eclipse.org Main Page. `http://www.eclipse.org/`.

[47] M. A. Reniers. Static Semantics of Message Sequence Charts, 1996.

[48] Dines Bjørner, Chris W. George, Anne E. Haxthausen, Christian Krog Madsen, Steffen Holmslykke, and Martin Penicka. "UML–ising" Formal Techniques. In *Springer-Verlag book on DG INT projects?*, 2004.

[49] Jun Sun and Jin Song Dong. Model checking live sequence charts. not published, 2004. Paper written in conjunction with Ph.D. Thesis at School of Computing, National University of Singapore, (sunj, dongjs)@comp.nus.edu.sg.

[50] Jun Sun and Jin Song Dong. Synthesizing distributed processes from scenario-based specification. not published, 2004. Paper written in conjunction with Ph.D. Thesis at School of Computing, National University of Singapore, (sunj, dongjs)@comp.nus.edu.sg.

[51] Tao Wang, Abhik Roychoudhury, Roland H. C. Yap, and S. C. Choudhary. Symbolic execution of behavioral requirements. In *PADL*, pages 178–192, 2004.

[52] Xia Yong and Chris George. An operational semantics for timed raise. In *World Congress on Formal Methods*, pages 1008–1027, 1999.

[53] Phil Zoio. A shape diagram editor, 2004. http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html.

Part V

Appendix

Appendix A

Glossary

In the following glossaries for Scheme Diagram and LSC are presented. All names in bold are defined.

A.1 Scheme Diagram

The Scheme Diagram is inspired by the way the UML Class Diagram is depicted. But also by some of the natural language terms. Consequently the terms used in conjunction with the Scheme Diagram are a mixture of RSL and UML.

Association A relation in the Scheme Diagram. It is a precondition for qualification and for associations of kind parameter and nested. It also corresponds to object declarations. The term is borrowed from the UML Class Diagram. See section 5.5.5.

Client Denotes the dependent participant of a binary relation. See definition in section 5.4.

Global An association *kind* kind where the $supplier$ of the association must be a global object. See section 5.5.5.

Inheritance Originates from object oriented terminology. It resembles the **extend** operator in RSL but there are differences. See section 5.5.6.

Model The term is borrowed from UML and corresponds to a specification in RSL. The term is used since the term specification in RSL only includes modules. The Scheme Diagram also includes relations which is emphasised by the term model. See section 5.5.1.

Multiplicity Corresponds to arrays in RSL terminology. See section 5.5.5.

Navigability The term is borrowed from UML and is used to indicate the direction in which an association can be traversed.

Nested An association *kind* which corresponds to a nested object declaration in a scheme. See section 5.5.5.

Parameter An association *kind* which corresponds to a formal parameter of a scheme. See section 5.5.5.

Rolename Is used in conjunction with the association relation. It is the name of the object that is an instance of the $client$ in that association. See section 5.5.5.

State Used in conjunction with objects and denotes the set of variables which are available with corresponding values. See section 5.5.4 and 5.4.3.

Supplier Denotes the providing participant of a binary relation. See definition in section 5.4.

Visibility Denotes the visibility of a declaration from outside the module. The possibilities are either Private or Public. See section 5.5.3.

A.2 LSC

action Is an **event** that denotes a local computation on an **instance**. It is not specified in detail but denoted by a name.

active mainchart Is a mainchart whose **prechart behaviour** has been fulfilled at some given moment. The system must now exhibit the **behaviour** expressed in the **active mainchart**.

asynchronous A **message event** may be **asynchronous**, meaning that the reception of the message may be later than the sending.

behaviour **LSCs** are used for describing wanted **behaviour**, i.e. **events** and computations that are done by some given system.

chart Consists of a given number of **instances**. It is used to group these in order to allow to show some given **behaviour**.

cold **Temperature** of events. Denotes possible **behaviour**. See also **hot**. Applies for **conditions, locations** and **messages**.

collection A set of **LSCs** that form a requirements specification.

condition An **event** that uses a predicate in order to determine if a **chart** should progress beyond the **condition**. May be **hot** or **cold**. At a true **condition**, the **chart** proceeds. At a false **cold condition** exits the current **chart**. A **false hot condition** is a violation of the specification and is thus not allowed.

consistency Two **LSCs** can specify **behaviour** that contradicts each other. If a **collection** is **consistent**, no two **LSCs** in a **collection** contradict each other.

coregion Consists of **message events**. The ordering in which those events happen is specified to be irrelevant by using a **coregion**.

creation An **instance** may create a new **instance**. The new instance can only be referenced in the **LSC** it is created.

cut Denotes the progress of a **chart**. A **cut** shows the **location** after the **events** that have already been performed.

enabled event Is an **event** that in a given **state** may be performed as the next event without violating the **LSC**.

event Is an atomic part of a **behaviour.** It may be a **action, condition or input/output message event**.

existential A **LSC** may be existential. This denotes that the **behaviour** specified by the **LSC** must at least be exhibited once in the lifetime of the system that is described.

hot **Temperature** of events. Denotes mandatory **behaviour**. See also **cold**. Applies for **conditions, locations** and **messages**.

instance Is an ordered list of **locations**. Instances may denote object, processes etc. and have a name.

LSC Is used for describing a scenario with **behaviour** that the system that is described must exhibit. It consists of a **prechart** and a **mainchart.**

location Is a part of an **instance**. It has a **temperature** and an **event**.

mainchart Is the part of the **LSC** that describes **behaviour** that the system that is described must exhibit under certain conditions.

message Consists of a message output **event** and a corresponding message input **event**. Together they denote that some form of communication happens between two **instances**. They have a temperature and are synchronous or asynchronous.

multiplicity Is the maximum number of times a **subchart** may be repeated.

play-in/play-out The concept of automatically generating **LSCs** based on users specification of wanted behaviour using a mock GUI (play-in). Play-out is to execute the resulting **collection** of **LSCs** as the finished program that behaves according to the **LSCs**.

prechart A precondition in order to **activate a mainchart.** A prechart may not use **cold conditions.**

subchart Can be used in order to specify repeated **behaviour** using **multiplicity**. May also used for branching. A **subchart** encloses a certain coherent section on a **chart**.

state Is a **cut** annotated with information about already happened **events.** Based on the **state**, all **enabled events** can be identified.

step The process of performing one **event** in a **LSC** in a given **state**. A **step** will thus advance the **state** to the next **state** denoting that the **event** has been exhibited.

synchronous A **message event** may be **synchronous**, meaning that a sent **message** is received immediately.

temperature Used for denoting mandatory vs. optional behaviour, see **hot** and **cold**.

trace Is a list of **states** that uniquely identify in which order the **events** in a run of a system have happened.

universal A **LSC** may be universal. This denotes that the **behaviour** specified by the **LSC** must always be exhibited by the system that is described.

Appendix B

Description of RSL types in RSL

Contents

B.1 rslsyntax.rsl 125
B.2 rslprint.rsl 143

B.1 rslsyntax.rsl

scheme rslsyntax = **class**
type
/∗
Specifies the context for the rsltc tool. That is,
the names of the other RSL that is used. Context
is thus a mapping from a module name to the set of
other module names used. This information is not
included in the rslsyntax scheme since it beyond
the syntax for RSL and more a requirement from the
rsltc tool.
∗/
Context = id $\overrightarrow{m}$ id-**set**

type
- - [p. 269, 1]
specification = {| mdl : module_decl* • **len** mdl > 0 |},

- - [p. 270, 1]

module_decl ==
module_decl_from_scheme_decl(module_decl_to_scheme_decl :
scheme_decl) | module_decl_from_object_decl(module_decl_to_object_decl : object_decl),

- - [p. 270, 1]

decl == decl_from_scheme_decl(decl_to_scheme_decl : scheme_decl) |
decl_from_object_decl(decl_to_object_decl : object_decl) |
decl_from_type_decl(decl_to_type_decl : type_decl) |
decl_from_value_decl(decl_to_value_decl : value_decl) |
decl_from_variable_decl(decl_to_variable_decl : variable_decl) |
decl_from_channel_decl(decl_to_channel_decl : channel_decl) |
decl_from_axiom_decl(decl_to_axiom_decl : axiom_decl),

```
-- scheme_decl ::= scheme scheme_def_3list
-- [p. 270, 1]
scheme_decl :: {| sdl : scheme_def* • len sdl > 0 |},

-- scheme_def ::= opt1_comment_4string id opt4_formal_scheme_parameter = class_expr
-- [p. 270, 1]
scheme_def :: id opt_formal_scheme_parameter class_expr,

-- formal_scheme_parameter ::= ( formal_scheme_argument_2list )
-- [p. 271, 1]
opt_formal_scheme_parameter = formal_scheme_argument*,

-- formal_scheme_argument ::= object_def
-- [p. 271, 1]
formal_scheme_argument :: object_def,

-- object_decl ::= object object_def_3list
-- [p. 272, 1]
object_decl :: {| odl : object_def* • len odl > 0|},

-- object_def ::= opt1_comment_4string id opt4_formal_array_parameter : class_expr
-- [p. 272, 1]
object_def :: id opt_formal_array_parameter class_expr,

-- formal_array_parameter ::= [ typing_2list ]
-- [p. 272, 1]
opt_formal_array_parameter = typing*,

-- type_decl ::= type type_def_3list
-- [p. 273, 1]
type_decl :: {| tdl : type_def* • len tdl > 0|},

-- type_def ::= sort_def | variant_def | union_def | short_record_def | abbreviation_def
-- [p. 273, 1]

type_def == type_def_from_sort_def(type_def_to_sort_def : sort_def) |
type_def_from_variant_def(type_def_to_variant_def : variant_def) |
type_def_from_union_def(type_def_to_union_def : union_def) |
type_def_from_short_record_def(type_def_to_short_record_def : short_record_def) |
type_def_from_abbreviation_def(type_def_to_abbreviation_def : abbreviation_def),

-- sort_def ::= opt1_comment_4string id
-- [p. 273, 1]
sort_def :: id,

-- variant_def ::= opt1_comment_4string id == variant_2choice
-- [p. 274, 1]
variant_def :: id  {| vl : variant* • len vl > 0|},

-- variant ::= constructor | record_variant
-- [p. 274, 1]

variant ==
variant_from_constructor(variant_to_constructor : constructor) |
variant_from_record_variant(variant_to_record_variant : record_variant),

-- record_variant ::= constructor ( component_kind_2list )
```

```
- - [p. 274, 1]
record_variant :: constructor {| ckl : component_kind* • len ckl > 0|},

- - component_kind ::= opt6_destructor type_expr opt5_reconstructor
- - [p. 274, 1]
- - TODO
component_kind :: opt_destructor type_expr opt_reconstructor,
opt_destructor == opt_dest_none | opt_destructor_from_destructor(opt_destructor_to_destructor : destructor),
opt_reconstructor ==
opt_reco_none |
opt_reconstructor_from_reconstructor(opt_reconstructor_to_reconstructor : reconstructor),

- - constructor ::= id_or_op | _
- - [p. 274, 1]

constructor == constructor_from_id_or_op(constructor_to_id_or_op :
id_or_op) | con_wildcard,

- - destructor ::= id_or_op :
- - [p. 274, 1]
destructor = id_or_op,

- - reconstructor ::= <-> id_or_op
- - [p. 274, 1]
reconstructor = id_or_op,

- - union_def ::= opt1_comment_4string id = name_or_wildcard_1choice2
- - [p. 279, 1]
union_def :: id {| nwl : name_or_wildcard* • len nwl > 1|},

- - name_or_wildcard ::= .type_.name | _
- - [p. 279, 1]

name_or_wildcard ==
name_or_wildcard_from_type_name(name_or_wildcard_to_type_name :
type_name) | nw_wildcard,
type_name = {| n:name • name_type(n)|},

- - short_record_def ::= opt1_comment_4string id ::
- - component_kind_5string
- - [p. 279, 1]
short_record_def :: id {| ckl : component_kind* • len ckl > 0 |},

- - abbreviation_def ::= opt1_comment_4string id = type_expr
- - [p. 280, 1]
abbreviation_def :: id type_expr,

- - value_decl ::= value value_def_5list
- - [p. 280, 1]
value_decl :: {| vdl : value_def* • len vdl > 0|},


- - value_def ::= commented_typing | explicit_value_def | implicit_value_def |
- - explicit_function_def | implicit_function_def
- - [p. 280, 1]

value_def == value_def_from_commented_typing(value_def_to_commented_typing : commented_typing) |
   value_def_from_explicit_value_def(value_def_to_explicit_value_def :
```

```
  explicit_value_def) |
  value_def_from_implicit_value_def(value_def_to_implicit_value_def :
  implicit_value_def) |
                    value_def_from_explicit_function_def(value_def_to_explicit_function_def :
                    explicit_function_def) |
  value_def_from_implicit_function_def(value_def_to_implicit_function_def :
  implicit_function_def),


-- explicit_value_def ::= opt1_comment_4string single_typing = .pure_.value_expr
-- [p. 281, 1]
explicit_value_def :: single_typing  {|ve:value_expr • pure(ve)|},

-- implicit_value_def ::= opt1_comment_4string single_typing .pure_.restriction
-- [p. 281, 1]
implicit_value_def :: single_typing {|r : restriction • pure(r)|},

-- explicit_function_def ::= opt1_comment_4string single_typing formal_function_application is value_expr_pr12 opt5_pre_condition
-- [p. 282, 1]
explicit_function_def :: single_typing formal_function_application value_expr opt_pre_condition,

opt_pre_condition ==
opt_prec_none |
opt_pre_condition_from_pre_condition(opt_pre_condition_to_pre_condition : pre_condition),

-- formal_function_application ::= id_application | prefix_application | infix_application
-- [p. 282, 1]

formal_function_application ==
formal_function_application_from_id_application(formal_function_application_to_id_application
: id_application) |
formal_function_application_from_prefix_application(formal_function_application_to_prefix_application :
prefix_application)|
formal_function_application_from_infix_application(formal_function_application_to_infix_application : infix_application),

-- id_application ::= .value_.id formal_function_parameter_0string
-- [p. 282, 1]
id_application :: {|i : id • id_value(i)|} {| ffpl : formal_function_parameter* • len ffpl > 0|},

-- formal_function_parameter ::= ( opt4_binding_2list )
-- [p. 282, 1]
formal_function_parameter :: binding*,

-- prefix_application ::= prefix_op id
-- [p. 282, 1]
prefix_application :: prefix_op id,

-- infix_application ::= id infix_op id
-- [p. 282, 1]
infix_application :: id infix_op id,

-- implicit_function_def ::= opt1_comment_4string single_typing formal_function_application post_condition opt5_pre_condition
-- [p. 284, 1]
implicit_function_def :: single_typing formal_function_application post_condition opt_pre_condition,

-- variable_decl ::= variable variable_def_3list
-- [p. 287, 1]
variable_decl :: {| vdl : variable_def* • len vdl > 0 |},
```

```
- - variable_def ::= single_variable_def | multiple_variable_def,
- - [p. 287]
variable_def ==
variable_def_from_single_variable_def(variable_def_to_single_variable_def
: single_variable_def) | variable_def_from_multiple_variable_def(variable_def_to_multiple_variable_def : multiple_variabl

- - single_variable_def ::= opt1_comment_4string id : type_expr opt5_initialisation
- - [p. 287]

single_variable_def :: id type_expr opt_initialisation,

opt_initialisation ==
opt_init_none |
opt_initialisation_from_initialisation(opt_initialisation_to_initialisation : initialisation),
initialisation = {|ve : value_expr • pure(ve)|},

- - multiple_variable_def ::= opt1_comment_4string id_1list2 : type_expr
- - [P. 287]
multiple_variable_def :: {| idl : id* • len idl > 1|} type_expr,

- - channel_decl ::= channel channel_def_3list
- - [p. 288, 1]
channel_decl :: {| cdl : channel_def* • len cdl > 0|},

- - channel_def ::= single_channel_def | multiple_channel_def
- - [p.288]

channel_def ==
channel_def_from_single_channel_def(channel_def_to_single_channel_def
: single_channel_def) |
channel_def_from_multiple_channel_def(channel_def_to_multiple_channel_def : multiple_channel_def),

- - single_channel_def ::= opt1_comment_4string id : type_expr
- - [p.288]
single_channel_def :: id type_expr,

- - multiple_channel_def ::= opt1_comment_4string id_1list2 : type_expr
- - [p.288]
multiple_channel_def :: {| idl : id* • len idl > 1|} type_expr,

- - axiom_decl ::= axiom axiom_def_5list
- - [p. 289, 1]
axiom_decl :: {| adl : axiom_def* • len adl > 0 |},

- - axiom_def ::= opt1_comment_4string opt6_axiom_naming .readonly_logical_.value_expr
- - [p. 289]
axiom_def :: opt_axiom_naming {|ve : value_expr • readonly(ve) ∧ logical(ve)|},
opt_axiom_naming ==
opt_axio_none |
opt_axiom_naming_from_axiom_naming(opt_axiom_naming_to_axiom_naming : axiom_naming),


- - axiom_naming ::= [ id ]
- - [p.289]
axiom_naming :: id,

- - class_expr ::= basic_class_expr | extending_class_expr | hiding_class_expr |
```

```
-- renaming_class_expr | with_class_expr | scheme_instantiation
-- [p.291]

class_expr ==
class_expr_from_basic_class_expr(class_expr_to_basic_class_expr :
basic_class_expr) |
class_expr_from_extending_class_expr(class_expr_to_extending_class_expr : extending_class_expr) |
class_expr_from_hiding_class_expr(class_expr_to_hiding_class_expr :
hiding_class_expr) |
class_expr_from_renaming_class_expr(class_expr_to_renaming_class_expr
: renaming_class_expr) |
class_expr_from_with_class_expr(class_expr_to_with_class_expr :
with_class_expr) |
class_expr_from_scheme_instantiation(class_expr_to_scheme_instantiation : scheme_instantiation),

-- basic_class_expr ::= class opt8_decl_3string end
-- [p.292]
basic_class_expr :: decl*,

-- extending_class_expr ::= extend class_expr with class_expr
-- [p.292]
extending_class_expr :: class_expr class_expr,

-- hiding_class_expr ::= hide defined_item_1list in class_expr
-- [p.293]
hiding_class_expr :: {| dil : defined_item* • len dil > 0 |}  class_expr,

-- renaming_class_expr ::= use rename_pair_2list in class_expr
-- [p.293]
renaming_class_expr :: {| rpl : rename_pair* • len rpl > 0 |}  class_expr,

-- with_class_expr ::= with .element_.object_expr in class_expr
-- [p. NEW]
with_class_expr ::  {|oel : object_expr* • element(oel) ∧ len oel > 0|} class_expr,

-- scheme_instantiation ::= .scheme_.name opt4_actual_scheme_parameter
-- [p.294]
scheme_instantiation :: {|n: name • name_scheme(n)|}  opt_actual_scheme_parameter,
opt_actual_scheme_parameter ==
opt_asp_none |
opt_actual_scheme_parameter_from_actual_scheme_parameter(opt_actual_scheme_parameter_to_actual_scheme_parameter
: actual_scheme_parameter),

-- actual_scheme_parameter ::= ( object_expr_2list )
-- [p.294]
actual_scheme_parameter :: {| oel : object_expr* • len oel > 0 |},

-- rename_pair ::= defined_item for defined_item
--[p.297]
rename_pair :: defined_item defined_item,

-- defined_item ::= id_or_op | disambiguated_item
-- [p.297]

defined_item == defined_item_from_id_or_op(defined_item_to_id_or_op :
id_or_op) | defined_item_from_disambiguated_item(defined_item_to_disambiguated_item : disambiguated_item),

-- disambiguated_item ::= id_or_op : type_expr
```

```
- - [p.297]
disambiguated_item :: id_or_op type_expr,


- - object_expr ::= .object_.name | element_object_expr |
- - array_object_expr | fitting_object_expr
- -[p.299]

object_expr == object_expr_from_object_name(object_expr_to_object_name
: object_name) |
object_expr_from_element_object_expr(object_expr_to_element_object_expr
: element_object_expr) |
object_expr_from_array_object_expr(object_expr_to_array_object_expr :
array_object_expr) |
object_expr_from_fitting_object_expr(object_expr_to_fitting_object_expr : fitting_object_expr),
object_name = {| n: name • name_object(n)|},

- - element_object_expr ::= .array_.object_expr actual_array_parameter
- -[p.300]
element_object_expr :: {|oe:object_expr • array(oe)|} actual_array_parameter,

- - actual_array_parameter ::= [ .pure_.value_expr_2list ]
- -[p.300]
actual_array_parameter :: {| vel : value_expr* • len vel > 0 ∧ pure(vel) |},

- - array_object_expr ::= [| typing_2list :- .element_.object_expr |]
- -[p.300]
array_object_expr :: {| tyl : typing* • len tyl > 0 |} {| oe : object_expr • element(oe)|},

- - fitting_object_expr ::= object_expr  rename_pair_2list
- - [p.301]
fitting_object_expr :: object_expr {| rpl : rename_pair* • len rpl > 0 |},


- - [p.302]
type_expr == type_expr_from_type_literal(type_expr_to_type_literal :
type_literal) | type_expr_from_name(type_expr_to_name : name) |
type_expr_from_product_type_expr(type_expr_to_product_type_expr :
product_type_expr) | type_expr_from_set_type_expr(type_expr_to_set_type_expr : set_type_expr) |
type_expr_from_list_type_expr(type_expr_to_list_type_expr :
list_type_expr) |
type_expr_from_map_type_expr(type_expr_to_map_type_expr :
map_type_expr) |
type_expr_from_function_type_expr(type_expr_to_function_type_expr :
function_type_expr) |
type_expr_from_subtype_expr(type_expr_to_subtype_expr : subtype_expr)
| type_expr_from_bracketed_type_expr(type_expr_to_bracketed_type_expr : bracketed_type_expr),

- - type_literal ::= Unit | Bool | Int | Nat | Real | Text | Char
- - [p.302]

type_literal == tl_Unit | tl_Bool | tl_Int | tl_Nat | tl_Real | tl_Text | tl_Char,

- - product_type_expr ::= type_expr_pr1_product2
- - [p.305]
product_type_expr :: {| tel : type_expr* • len tel > 1 |},

- - set_type_expr ::= finite_set_type_expr | infinite_set_type_expr
```

```
-- [p.305]

set_type_expr ==
set_type_expr_from_finite_set_type_expr(set_type_expr_to_finite_set_type_expr
: finite_set_type_expr) |
set_type_expr_from_infinite_set_type_expr(set_type_expr_to_infinite_set_type_expr : infinite_set_type_expr),

-- finite_set_type_expr ::= type_expr_pr0-set
-- [p.305]
finite_set_type_expr :: type_expr,

-- infinite_set_type_expr ::= type_expr_pr0-infset
-- [p.305]
infinite_set_type_expr :: type_expr,

-- list_type_expr ::= finite_list_type_expr | infinite_list_type_expr
-- [p.306]

list_type_expr ==
list_type_expr_from_finite_list_type_expr(list_type_expr_to_finite_list_type_expr
: finite_list_type_expr) |
list_type_expr_from_infinite_list_type_expr(list_type_expr_to_infinite_list_type_expr :
infinite_list_type_expr),

-- finite_list_type_expr ::= type_expr_pr0-list
-- [p.306]
finite_list_type_expr :: type_expr,

-- infinite_list_type_expr ::= type_expr_pr0-inflist
-- [p.306]
infinite_list_type_expr :: type_expr,

-- map_type_expr ::= finite_map_type_expr | infinite_map_type_expr
-- [p.306]
map_type_expr ==
map_type_expr_from_finite_map_type_expr(map_type_expr_to_finite_map_type_expr
: finite_map_type_expr) |
map_type_expr_from_infinite_map_type_expr(map_type_expr_to_infinite_map_type_expr : infinite_map_type_expr),

-- finite_map_type_expr ::= type_expr_pr2 -m-> type_expr_pr3
-- [p.306]
finite_map_type_expr :: type_expr type_expr,

-- infinite_map_type_expr ::= type_expr_pr2 -~m-> type_expr_pr3
-- [p.306]
infinite_map_type_expr :: type_expr type_expr,

-- function_type_expr ::= type_expr_pr2 function_arrow result_desc
-- [p.307]
function_type_expr :: type_expr function_arrow result_desc,

-- function_arrow ::= -~-> | ->

function_arrow == fa_total | fa_partial,
-- [p.307]

-- result_desc ::= opt3_access_desc_1string type_expr_pr3
-- [p.307]
```

```
result_desc :: accss_desc* type_expr,

- - subtype_expr ::= | single_typing .pure_.restriction |
- - [p.308]
subtype_expr :: single_typing {| r : restriction • pure(r) |},

- - bracketed_type_expr ::= ( type_expr )
- - [p.309]
bracketed_type_expr :: type_expr,

- - access_desc ::= access_mode access_1list
- - [p.309]
accss_desc :: accss_mode {| acl : accss* • len acl > 0 |},

- - access_mode ::= read | write | in | out
- - [p.309]
accss_mode == am_read | am_write | am_in | am_out,

- - access ::= .variable_or_channel_.name |
- - enumerated_access | completed_access | comprehended_access
- - [p.309]

accss == accss_from_acc_name(accss_to_acc_name : acc_name) |
accss_from_enumerated_accss(accss_to_enumerated_accss :
enumerated_accss) |
accss_from_completed_accss(accss_to_completed_accss :
completed_accss) | accss_from_comprehended_accss(accss_to_comprehended_accss
: comprehended_accss),

acc_name = {| n: name • name_channel(n) ∨ name_variable(n)|},

- - enumerated_access ::=  opt4_access_1list
- - [p.309]
enumerated_accss :: accss*,

- - completed_access ::= opt7_qualification any
- - [p.309]
completed_accss :: opt_qualification,
opt_qualification ==
opt_qual_none |
opt_qualification_from_qualification(opt_qualification_to_qualification : qualification),

- - comprehended_access ::=  access | .pure_.set_limitation
- - [p.309]
comprehended_accss :: accss {|sl: set_limitation • pure(sl)|},


- - [p.312]
value_expr ==
ve_val_l(value_literal) |
ve_name({| n:name • name_value(n) ∨ name_variable(n)|}) |
ve_pren(pre_name) |
ve_bas_e(basic_expr) |
ve_pro_e(product_expr) |
ve_set_e(set_expr) |
ve_lis_e(list_expr) |
ve_map_e(map_expr) |
ve_fun_e(function_expr) |
```

```
ve_app_e(application_expr) |
ve_qua_e(quantified_expr) |
ve_equ_e(equivalence_expr) |
ve_pos_e(post_expr) |
ve_dis_e(disambiguation_expr) |
ve_bra_e(bracketed_expr) |
ve_inf_e(infix_expr)|
ve_pre_e(prefix_expr) |
ve_com_e(comprehended_expr) |
ve_ini_e(initialise_expr) |
ve_ass_e(assignment_expr) |
ve_inp_e(input_expr) |
ve_out_e(output_expr) |
ve_str_e(structured_expr),



-- value_literal ::= unit_literal | bool_literal | int_literal |
-- real_literal | text_literal | char_literal
-- [p.315]
value_literal == unit_literal | bool_literal(Bool) | int_literal(Int) |
real_literal(Real) | text_literal(Text) | char_literal(Char),

-- unit_literal ::= ( )
-- [p.315]
--unit_literal,

-- bool_literal ::= true | false
-- [p.315]
/*
bool_literal == bl_true | bl_false,
int_literal,
real_literal,
text_literal,
char_literal,*/
-- pre_name ::= .variable_.name ‘
-- [p.316]
pre_name = {| n:name • name_variable(n)|},

-- basic_expr ::= chaos | skip | stop | swap
-- [p.316]
basic_expr == be_chaos | be_skip | be_stop | be_swap,

-- product_expr ::= ( value_expr_2list2 )
-- [p.316]
product_expr :: {| vel : value_expr* • len vel > 1 |},

-- set_expr ::= ranged_set_expr | enumerated_set_expr | comprehended_set_expr
-- [p.317]

set_expr == set_expr_from_ranged_set_expr(set_expr_to_ranged_set_expr
: ranged_set_expr) |
set_expr_from_enumerated_set_expr(set_expr_to_enumerated_set_expr :
enumerated_set_expr) |
set_expr_from_comprehended_set_expr(set_expr_to_comprehended_set_expr : comprehended_set_expr),

-- ranged_set_expr ::=  .readonly_integer_.value_expr .. .readonly_integer_.value_expr
-- [p.317]
```

```
ranged_set_expr :: {|ve:value_expr • readonly(ve) ∧ integer(ve)|} {|ve:value_expr • readonly(ve) ∧ integer(ve)|},

- - enumerated_set_expr ::=  .readonly_.opt4_value_expr_2list
- - [p.317]
enumerated_set_expr :: {|ve:value_expr • readonly(ve)|}*,

- - comprehended_set_expr ::=  .readonly_.value_expr | set_limitation
- - [p.318]
comprehended_set_expr :: {|ve:value_expr • readonly(ve)|} set_limitation,

- - set_limitation ::= typing_2list opt2_restriction
- - [p.318]
set_limitation :: {| tyl : typing* • len tyl > 0 |} opt_restriction,
opt_restriction == opt_rest_none | opt_restriction_from_restriction(opt_restriction_to_restriction : restriction),

- - restriction ::= :- .readonly_logical_.value_expr
- - [p. 318, 1]
restriction :: {|ve:value_expr • readonly(ve) ∧ logical(ve)|},


- - list_expr ::= ranged_list_expr | enumerated_list_expr | comprehended_list_expr
- - [p.319]
list_expr ==
list_expr_from_ranged_list_expr(list_expr_to_ranged_list_expr :
ranged_list_expr) |
list_expr_from_enumerated_list_expr(list_expr_to_enumerated_list_expr
: enumerated_list_expr) |
list_expr_from_comprehended_list_expr(list_expr_to_comprehended_list_expr : comprehended_list_expr),


- - ranged_list_expr ::= <. .integer_.value_expr .. .integer_.value_expr .>
- - [p.319]
ranged_list_expr :: {|ve:value_expr • integer(ve)|} {|ve:value_expr • integer(ve)|},

- - enumerated_list_expr ::= <. opt4_value_expr_2list .>
- - [p.319]
enumerated_list_expr :: value_expr*,

- - comprehended_list_expr ::= <. value_expr | list_limitation .>
- - [p.320]
comprehended_list_expr :: value_expr list_limitation,

- - list_limitation ::= binding in .readonly_list_.value_expr opt2_restriction
- - [p.320]
list_limitation :: binding {|ve:value_expr • readonly(ve) ∧list(ve)|} opt_restriction,

- - map_expr ::= enumerated_map_expr | comprehended_map_expr
- - [p.321]
map_expr ==
    map_expr_from_enumerated_map_expr(map_expr_to_enumerated_map_expr :
    enumerated_map_expr) |
    map_expr_from_comprehended_map_expr(map_expr_to_comprehended_map_expr :
    comprehended_map_expr),

- - enumerated_map_expr ::= [ opt4_value_expr_pair_2list ]
- - [p.321]
enumerated_map_expr :: value_expr_pair*,
```

```
- - value_expr_pair ::= .readonly_.value_expr +> .readonly_.value_expr
- - [p.321]
value_expr_pair :: value_expr value_expr,

- - comprehended_map_expr ::= [ value_expr_pair | set_limitation ]
- - [p.322]
comprehended_map_expr :: value_expr_pair set_limitation,

- - function_expr ::= - lambda_parameter :- value_expr_pr14
- - [p.322]
function_expr :: lambda_parameter value_expr,

- - lambda_parameter ::= lambda_typing | single_typing
- - [p.322]

lambda_parameter ==
lambda_parameter_from_lambda_typing(lambda_parameter_to_lambda_typing
: lambda_typing) | lambda_parameter_from_single_typing(lambda_parameter_to_single_typing : single_typing),

- - lambda_typing ::= ( opt4_typing_2list )
- - [p.322]
lambda_typing :: typing*,

- - application_expr ::= .list_or_map_or_function_.value_expr_pr255 actual_function_parameter_0string
- - [p.323]
application_expr :: value_expr actual_function_parameter,

- - actual_function_parameter ::= ( opt4_value_expr_2list )
- - [p.324]
actual_function_parameter :: value_expr*,

- - quantified_expr ::= quantifier typing_2list restriction
- - [p.325]
quantified_expr :: quantifier {| tyl:typing* • len tyl > 0 |} restriction,

- - quantifier ::= all | exists | exists!
- - [p.322]
quantifier == qu_all | qu_exists | qu_existsem,

- - equivalence_expr ::= value_expr_pr12 is value_expr_pr12 opt5_pre_condition
- - [p.326]
equivalence_expr :: value_expr value_expr opt_pre_condition,

- - pre_condition ::= pre .readonly_logical_.value_expr_pr12
- - [p. 326, 1]
pre_condition :: {|ve:value_expr • readonly(ve) ∧ logical(ve)|},

- - post_expr ::= value_expr_pr12 post_condition opt5_pre_condition
- - [p. 327, 1]
post_expr :: value_expr post_condition opt_pre_condition,

- - post_condition ::= opt2_result_naming post .readonly_logical_.value_expr_pr12
- - [p. 327, 1]
post_condition :: opt_result_naming {|ve:value_expr • readonly(ve) ∧ logical(ve)|},
opt_result_naming ==
opt_resn_none |
opt_result_naming_from_result_naming(opt_result_naming_to_result_naming : result_naming),
```

```
- - result_naming ::= as binding
- - [p. 327, 1]
result_naming :: binding,


- - disambiguation_expr ::= value_expr_pr1 : type_expr
- - [p.328]
disambiguation_expr :: value_expr type_expr,

- - bracketed_expr ::= ( value_expr )
- - [p.328]
bracketed_expr :: value_expr,

- - [p.329]
infix_expr ==
infix_expr_from_stmt_infix_expr(infix_expr_to_stmt_infix_expr :
stmt_infix_expr) |
infix_expr_from_axiom_infix_expr(infix_expr_to_axiom_infix_expr :
axiom_infix_expr) | infix_expr_from_value_infix_expr(infix_expr_to_value_infix_expr : value_infix_expr),

stmt_infix_expr :: value_expr infix_combinator value_expr,

axiom_infix_expr :: {|ve: value_expr • logical(ve) |} infix_connective value_expr,

value_infix_expr :: value_expr infix_op value_expr,

- -[p.330]
prefix_expr ==
prefix_expr_from_axiom_prefix_expr(prefix_expr_to_axiom_prefix_expr :
axiom_prefix_expr) |
prefix_expr_from_universal_prefix_expr(prefix_expr_to_universal_prefix_expr
: universal_prefix_expr) |
prefix_expr_from_value_prefix_expr(prefix_expr_to_value_prefix_expr : value_prefix_expr),

axiom_prefix_expr  :: prefix_connective {| ve:value_expr • logical(ve) |},

universal_prefix_expr :: {| ve:value_expr • logical(ve) ∧ readonly(ve) |},

value_prefix_expr :: prefix_op value_expr,


- - comprehended_expr ::= .associative_commutative_.infix_combinator  value_expr | set_limitation
- - [p.331]
comprehended_expr :: {| ic:infix_combinator • associative(ic) ∧ commutative(ic) |} value_expr set_limitation,

- - initialise_expr ::= opt7_qualification initialise
- -[p.332]
initialise_expr :: opt_qualification,

- - assignment_expr ::= .variable_.name := value_expr_pr9
- - [p.332]
assignment_expr :: {|n:name • name_variable(n)|} value_expr,

- - input_expr ::= .channel_.name ?
- - [p.331]
input_expr :: {|n:name • name_channel(n)|},

- - output_expr ::= .channel_.name ! value_expr_pr9
```

```
-- [p.331]
output_expr :: {|n:name • name_channel(n)|} value_expr,

-- structured_expr ::= local_expr | let_expr | if_expr | case_expr | while_expr | until_expr | for_expr
-- [p.334]

structured_expr ==
structured_expr_from_local_expr(structured_expr_to_local_expr :
local_expr) |
structured_expr_from_let_expr(structured_expr_to_let_expr : let_expr)
| structured_expr_from_if_expr(structured_expr_to_if_expr : if_expr) |
structured_expr_from_case_expr(structured_expr_to_case_expr :
case_expr) |
structured_expr_from_while_expr(structured_expr_to_while_expr :
while_expr) |
structured_expr_from_until_expr(structured_expr_to_until_expr :
until_expr) | structured_expr_from_for_expr(structured_expr_to_for_expr : for_expr),

-- local_expr ::= local opt8_decl_3string in value_expr end
-- [p.334]
local_expr :: decl* value_expr,

-- let_expr ::= let let_def_2list in value_expr end
-- [p.335]
let_expr :: let_def value_expr,

-- let_def ::= typing | explicit_let | implicit_let
-- [p.335]

let_def == let_def_from_typing(let_def_to_typing : typing) |
let_def_from_explicit_let(let_def_to_explicit_let : explicit_let) |
let_def_from_implicit_let(let_def_to_implicit_let : implicit_let),

-- explicit_let ::= let_binding = value_expr
-- [p.335]
explicit_let :: let_binding value_expr,

-- implicit_let ::= single_typing restriction
-- [p.335]
implicit_let :: single_typing restriction,

-- let_binding ::= binding | record_pattern | list_pattern
-- [p.335]

let_binding == let_binding_from_binding(let_binding_to_binding :
binding) |
let_binding_from_record_pattern(let_binding_to_record_pattern :
record_pattern) | let_binding_from_list_pattern(let_binding_to_list_pattern : list_pattern),


/*if_expr ::=
    if logical_value_expr then
        value_expr
    opt3_elsif_branch_1string
    opt3_else_branch
    end*/
-- [p.336]
if_expr :: {| ve:value_expr • logical(ve)|} value_expr elsif_branch* opt_else_branch,
```

```
opt_else_branch == opt_else_none | opt_else_branch_from_else_branch(opt_else_branch_to_else_branch : else_branch),

- - elsif_branch ::= elsif .logical_.value_expr then value_expr
- - [p.336]
elsif_branch :: {|ve: value_expr • logical(ve)|} value_expr,

- - else_branch ::= else value_expr
- - [p.336]
else_branch :: value_expr,

- - case_expr ::= case value_expr of case_branch_2list end
- - [p.337]
case_expr :: value_expr {| cbl : case_branch* • len cbl > 0 |},

- - case_branch ::= pattern -> value_expr
- - [p.338]
case_branch :: pattern value_expr,

- - while_expr ::= while .logical_.value_expr do .unit_.value_expr end
- - [p.338]
while_expr :: {| ve:value_expr • logical(ve)|} {| ve:value_expr • unit(ve)|},

- - until_expr ::= do .unit_.value_expr until .logical_.value_expr end
- - [p.339]
until_expr :: {| ve:value_expr • unit(ve)|} {| ve:value_expr • logical(ve)|},

- - for_expr ::= for list_limitation do .unit_.value_expr end
- - [p.339]
for_expr :: list_limitation {| ve:value_expr • unit(ve)|},


- - binding ::= id_or_op | product_binding
- - [p. 340, 1]
binding == binding_from_id_or_op(binding_to_id_or_op : id_or_op) |
binding_from_product_binding(binding_to_product_binding : product_binding),

- - product_binding ::= ( binding_2list2 )
- - [p. 340, 1]
product_binding :: {| bl : binding* • len bl > 1 |},

- - typing ::= single_typing | multiple_typing
- - [p. 342, 1]

typing == typing_from_single_typing(typing_to_single_typing :
single_typing) | typing_from_multiple_typing(typing_to_multiple_typing : multiple_typing),

- - single_typing ::= binding : type_expr
- - [p. 342, 1]
single_typing :: binding  type_expr,


- - multiple_typing ::= binding_2list2 : type_expr
- - [p. 342, 1]
multiple_typing :: {| bl : binding* • len bl > 1|}  type_expr,

- - commented_typing ::= opt1_comment_4string typing
- - [p. 342, 1]
commented_typing :: typing,
```

```
-- pattern ::= value_literal | .pure_value_.name | wildcard_pattern | product_pattern | record_pattern | list_pattern
-- [p. 344, 1]

pattern == pattern_from_value_literal(pattern_to_value_literal :
value_literal) | pattern_from_pv_name(pattern_to_pv_name : pv_name) |
pa_wildcard_pattern | pattern_from_product_pattern(pattern_to_product_pattern : product_pattern) |
pattern_from_record_pattern(pattern_to_record_pattern :
record_pattern) | pattern_from_list_pattern(pattern_to_list_pattern : list_pattern),

pv_name = {| n:name • name_pure(n) ∧ name_value(n) |},

-- product_pattern ::= ( inner_pattern_2list2 )
-- [p. 346, 1]
product_pattern :: {| ipl : inner_pattern* • len ipl > 1|},

-- record_pattern ::= .pure_value_.name ( inner_pattern_2list )
-- [p. 346, 1]
record_pattern :: {| n:name • name_pure(n) ∧ name_value(n) |} {| ipl : inner_pattern* • len ipl > 1 |},

-- list_pattern ::= enumerated_list_pattern | concatenated_list_pattern | right_list_pattern
-- [p. 347, 1]

list_pattern ==
list_pattern_from_enumerated_list_pattern(list_pattern_to_enumerated_list_pattern
: enumerated_list_pattern) |
list_pattern_from_concatenated_list_pattern(list_pattern_to_concatenated_list_pattern
: concatenated_list_pattern),
--| right_list_pattern,

-- enumerated_list_pattern ::= <. opt4_inner_pattern_2list .>
-- [p. 347, 1]
enumerated_list_pattern :: inner_pattern*,

-- concatenated_list_pattern ::= enumerated_list_pattern "hat" inner_pattern
-- [p. 348, 1]
concatenated_list_pattern :: enumerated_list_pattern  inner_pattern,

-- inner_pattern ::= value_literal | id_or_op | wildcard_pattern | product_pattern | record_pattern | list_pattern | equality_pattern
-- [p. 348, 1]

inner_pattern ==
inner_pattern_from_value_literal(inner_pattern_to_value_literal :
value_literal) | inner_pattern_from_id_or_op(inner_pattern_to_id_or_op
: id_or_op) |
ip_wildcard_pattern | inner_pattern_from_product_pattern(inner_pattern_to_product_pattern
: product_pattern)|
inner_pattern_from_record_pattern(inner_pattern_to_record_pattern :
record_pattern) |
inner_pattern_from_list_pattern(inner_pattern_to_list_pattern :
list_pattern) | inner_pattern_from_equality_pattern(inner_pattern_to_equality_pattern : equality_pattern),

-- equality_pattern ::= = .pure_value_.name
-- [p. 349, 1]
equality_pattern :: {| n:name • name_pure(n) ∧ name_value(n) |},

-- name ::= qualified_id | qualified_op
-- [p. 351, 1]
```

```
name == name_from_qualified_id(name_to_qualified_id : qualified_id) |
name_from_qualified_op(name_to_qualified_op : qualified_op),

- - qualified_id ::= opt7_qualification id
- - [p. 351, 1]
qualified_id :: opt_qualification id,

- - qualification ::= .element_.object_expr .
- - [p. 351, 1]
qualification :: {| oe:object_expr • element(oe) |},

- - qualified_op ::= opt7_qualification ( op )
- - [p. 352, 1]
qualified_op :: opt_qualification op,

- - id_or_op ::= id | op
- - [p. 353, 1]
id_or_op == id_or_op_from_id(id) | id_or_op_from_op(op),

- - op ::= infix_op | prefix_op
- - [p. 353, 1]

op == op_from_infix_op(op_to_infix_op : infix_op) |
op_from_prefix_op(op_to_prefix_op : prefix_op),

- - adheres to book [p. 354, 1] plus infix "==".
infix_op ==
io_eq | io_ieq | io_eqeq | io_gt | io_lt | io_gteq | io_lteq |
io_proper_superset | io_proper_subset | io_superset | io_subset |
io_isin | io_nisin | io_plus | io_minus | io_remainder_diff_restr |
io_concat | io_union | io_override | io_mult | io_div |
io_composition | io_inter | io_exponentation,

/*
prefix_op ::=
    minus |
    plus |
    abs |
    int |
    real |
    card |
    len |
    inds |
    elems |
    hd |
    tl |
    dom |
    rng
*/
- - [p. 359, 1]
prefix_op == po_minus | po_plus  | po_abs | po_int |
   po_real | po_card | po_len | po_inds |
   po_elems | po_hd | po_tl | po_dom | po_rng,

- - [p. 362, 1]
infix_connective == ic_imply | ic_or | ic_and,
```

```
-- prefix_connective ::= ∼
-- [p. 363, 1]
-- not necessary, since it is only text ("∼")
prefix_connective == pc_not,

/*
infix_combinator ::= infix_combinator_pr12 | infix_combinator_pr11
infix_combinator_pr12 ::= [] | |hat| | || | #
infix_combinator_pr11 ::= ;
*/

infix_combinator == icb_ext_choice | icb_int_choice | icb_concurrent |
icb_interlocked | icb_sequential,

id = Text

value
    element : object_expr → Bool
    element(oe) ≡ true,
    element : object_expr* → Bool
    element(oel) ≡ true,
    array : object_expr → Bool
    array(oe) ≡ true,
    pure : set_limitation → Bool
    pure(sl) ≡ true,
    pure : restriction → Bool
    pure(r) ≡ true,
    name_pure : name → Bool
    name_pure(n) ≡ true,
    name_type : name → Bool
    name_type(n) ≡ true,
    name_value : name → Bool
    name_value(n) ≡ true,
    name_variable : name → Bool
    name_variable(n) ≡ true,
    name_channel : name → Bool
    name_channel(n) ≡ true,
    name_scheme : name → Bool
    name_scheme(n) ≡ true,
    name_object : name → Bool
    name_object(n) ≡ true,
    id_value : id → Bool
    id_value(n) ≡ true,
    associative : infix_combinator → Bool
    associative(n) ≡ true,
    commutative : infix_combinator → Bool
    commutative(n) ≡ true,
    unit : value_expr → Bool
    unit(n) ≡ true,
    logical : value_expr → Bool
    logical(n) ≡ true,
    integer : value_expr → Bool
    integer(n) ≡ true,
    list : value_expr → Bool
    list(n) ≡ true,
    map : value_expr → Bool
    map(n) ≡ true,
    function : value_expr → Bool
```

```
    function(n) ≡ true,
    pure : value_expr → Bool
    pure(n) ≡ true,
    pure : value_expr* → Bool
    pure(n) ≡ true,
    readonly : value_expr → Bool
    readonly(n) ≡ true
```

B.2 rslprint.rsl

```
/∗ Page numbers refer to relevant pages in [18].∗/
scheme rslprint =
    extend rslsyntax with
    class
        value
            print_context : Context × id → Text
            print_context(c, n) ≡
                if c(n) = {} then ""
                else
                    let n′ = hd c(n) in
                        n′ ^
                        if card c(n) > 1
                        then ", " ^ print_context([ n ↦ c(n) \ {n′} ], n)
                        else ""
                        end
                    end
                end

        value
            -- [p. 269]
            print_specification : specification → Text
            print_specification(x) ≡ print_module_decl_list(x),

            print_module_decl_list : module_decl* → Text
            print_module_decl_list(x) ≡
                case x of
                    ⟨⟩ → "",
                    ⟨a⟩ ^ ⟨⟩ → print_module_decl(a),
                    ⟨a⟩ ^ b →
                        print_module_decl(a) ^ ", \n" ^
                        print_module_decl_list(b)
                end,

            -- [p. 270]
            print_module_decl : module_decl → Text
            print_module_decl(x) ≡
                case x of
                    module_decl_from_scheme_decl(a) →
                        print_scheme_decl(a),
                    module_decl_from_object_decl(a) →
                        print_object_decl(a)
                end,

            print_decl_list : decl* → Text
            print_decl_list(x) ≡
```

```
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_decl(a),
        ⟨a⟩ ⌢ b →
            print_decl(a) ⌢ ″ \n″ ⌢ print_decl_list(b)
    end,

- - [p. 270]
print_decl : decl → Text
print_decl(x) ≡
    case x of
        decl_from_scheme_decl(a) → print_scheme_decl(a),
        decl_from_object_decl(a) → print_object_decl(a),
        decl_from_type_decl(a) → print_type_decl(a),
        decl_from_value_decl(a) → print_value_decl(a),
        decl_from_variable_decl(a) → print_variable_decl(a),
        decl_from_channel_decl(a) → print_channel_decl(a),
        decl_from_axiom_decl(a) → print_axiom_decl(a)
    end,

- - [p. 270]
print_scheme_decl : scheme_decl → Text
print_scheme_decl(x) ≡
    case (x) of
        mk_scheme_decl(a) →
            ″scheme ″ ⌢ print_scheme_def_list(a)
    end,

print_scheme_def_list : scheme_def* → Text
print_scheme_def_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_scheme_def(a),
        ⟨a⟩ ⌢ b →
            print_scheme_def(a) ⌢ ″, \n″ ⌢
            print_scheme_def_list(b)
    end,

- - [p. 270]
print_scheme_def : scheme_def → Text
print_scheme_def(x) ≡
    case (x) of
        mk_scheme_def(a, b, c) →
            print_id(a) ⌢ ″ ″ ⌢
            print_opt_formal_scheme_parameter(b) ⌢ ″= ″ ⌢
            print_class_expr(c)
    end,

- - [p. 271]
print_opt_formal_scheme_parameter :
    opt_formal_scheme_parameter → Text
print_opt_formal_scheme_parameter(x) ≡
    case (x) of
        ⟨⟩ → ″″,
        _ → ″(" ^ print_formal_scheme_argument_list(x) ^ ") ″
    end,

print_formal_scheme_argument_list :
```

```
    formal_scheme_argument* → Text
print_formal_scheme_argument_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ⌢ ⟨⟩ → print_formal_scheme_argument(a),
        ⟨a⟩ ⌢ b →
            print_formal_scheme_argument(a) ⌢ ", \n" ⌢
            print_formal_scheme_argument_list(b)
    end,

-- [p. 271]
print_formal_scheme_argument :
    formal_scheme_argument → Text
print_formal_scheme_argument(x) ≡
    case (x) of
        mk_formal_scheme_argument(a) → print_object_def(a)
    end,

-- [p. 272]
print_object_decl : object_decl → Text
print_object_decl(x) ≡
    case (x) of
        mk_object_decl(a) →
            "object " ⌢ print_object_def_list(a)
    end,

print_object_def_list : object_def* → Text
print_object_def_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ⌢ ⟨⟩ → print_object_def(a),
        ⟨a⟩ ⌢ b →
            print_object_def(a) ⌢ ", \n" ⌢
            print_object_def_list(b)
    end,

-- [p. 272]
print_object_def : object_def → Text
print_object_def(x) ≡
    case (x) of
        mk_object_def(a, b, c) →
            print_id(a) ⌢ " " ⌢
            print_opt_formal_array_parameter(b) ⌢ ": " ⌢
            print_class_expr(c)
    end,

-- [p. 272]
print_opt_formal_array_parameter :
    opt_formal_array_parameter → Text
print_opt_formal_array_parameter(x) ≡
    case (x) of
        ⟨⟩ → "",
        _ → "[" ^ print_typing_list(x) ^ "] "
    end,

-- [p. 273]
print_type_decl : type_decl → Text
print_type_decl(x) ≡
```

```
    case (x) of
        mk_type_decl(a) → "type " ^ print_type_def_list(a)
    end,

print_type_def_list : type_def* → Text
print_type_def_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_type_def(a),
        ⟨a⟩ ^ b →
            print_type_def(a) ^ ", \n" ^ print_type_def_list(b)
    end,

-- [p. 273]
print_type_def : type_def → Text
print_type_def(x) ≡
    case x of
        type_def_from_sort_def(a) → print_sort_def(a),
        type_def_from_variant_def(a) → print_variant_def(a),
        type_def_from_union_def(a) → print_union_def(a),
        type_def_from_short_record_def(a) →
            print_short_record_def(a),
        type_def_from_abbreviation_def(a) →
            print_abbreviation_def(a)
    end,

-- [p. 273]
print_sort_def : sort_def → Text
print_sort_def(x) ≡
    case (x) of
        mk_sort_def(a) → print_id(a)
    end,

-- [p. 274]
print_variant_def : variant_def → Text
print_variant_def(x) ≡
    case (x) of
        mk_variant_def(a, b) →
            print_id(a) ^ " == " ^ print_variant_list(b)
    end,

print_variant_list : variant* → Text
print_variant_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_variant(a),
        ⟨a⟩ ^ b →
            print_variant(a) ^ " | " ^ print_variant_list(b)
    end,

-- [p. 274]
print_variant : variant → Text
print_variant(x) ≡
    case x of
        variant_from_constructor(a) → print_constructor(a),
        variant_from_record_variant(a) →
            print_record_variant(a)
    end,
```

```
-- [p. 274]
print_record_variant : record_variant → Text
print_record_variant(x) ≡
    case (x) of
        mk_record_variant(a, b) →
            print_constructor(a) ^ "(" ^
            print_component_kind_list(b) ^ ")  "
    end,

print_component_kind_list : component_kind* → Text
print_component_kind_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_component_kind(a),
        ⟨a⟩ ^ b →
            print_component_kind(a) ^ "  " ^
            print_component_kind_list(b)
    end,

-- [p. 274]
print_component_kind : component_kind → Text
print_component_kind(x) ≡
    case (x) of
        mk_component_kind(a, b, c) →
            print_opt_destructor(a) ^ print_type_expr(b) ^
            print_opt_reconstructor(c)
    end,

print_opt_destructor : opt_destructor → Text
print_opt_destructor(x) ≡
    case x of
        opt_dest_none → "",
        opt_destructor_from_destructor(a) →
            print_destructor(a) ^ "  "
    end,

print_opt_reconstructor : opt_reconstructor → Text
print_opt_reconstructor(x) ≡
    case x of
        opt_reco_none → "",
        opt_reconstructor_from_reconstructor(a) →
            print_reconstructor(a)
    end,

-- [p. 274]
print_constructor : constructor → Text
print_constructor(x) ≡
    case x of
        constructor_from_id_or_op(a) → print_id_or_op(a),
        con_wildcard → "_"
    end,

-- [p. 274]
print_destructor : destructor → Text
print_destructor(x) ≡ print_id_or_op(x) ^ ":  ",

-- [p. 274]
```

```
print_reconstructor : reconstructor → Text
print_reconstructor(x) ≡ "<->" ^ print_id_or_op(x),

- - [p. 279]
print_union_def : union_def → Text
print_union_def(x) ≡
    case (x) of
        mk_union_def(a, b) →
            print_id(a) ^ "= " ^ print_name_or_wildcard_list(b)
    end,

print_name_or_wildcard_list :
    name_or_wildcard* → Text
print_name_or_wildcard_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_name_or_wildcard(a),
        ⟨a⟩ ^ b →
            print_name_or_wildcard(a) ^ " | " ^
            print_name_or_wildcard_list(b)
    end,

- - [p. 279]
print_name_or_wildcard : name_or_wildcard → Text
print_name_or_wildcard(x) ≡
    case x of
        name_or_wildcard_from_type_name(a) →
            print_type_name(a),
        nw_wildcard → "_"
    end,

print_type_name : type_name → Text
print_type_name(x) ≡ print_name(x),

- - [p. 279]
print_short_record_def : short_record_def → Text
print_short_record_def(x) ≡
    case (x) of
        mk_short_record_def(a, b) →
            print_id(a) ^ " :: " ^ print_component_kind_list(b)
    end,

- - [p. 280]
print_abbreviation_def : abbreviation_def → Text
print_abbreviation_def(x) ≡
    case (x) of
        mk_abbreviation_def(a, b) →
            print_id(a) ^ " = " ^ print_type_expr(b)
    end,

- - [p. 280]
print_value_decl : value_decl → Text
print_value_decl(x) ≡
    case (x) of
        mk_value_decl(a) → "value " ^ print_value_def_list(a)
    end,

print_value_def_list : value_def* → Text
```

```
print_value_def_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_value_def(a),
        ⟨a⟩ ⌢ b →
            print_value_def(a) ⌢ ″, \n″ ⌢ print_value_def_list(b)
    end,

-- [p. 280]
print_value_def : value_def → Text
print_value_def(x) ≡
    case x of
        value_def_from_commented_typing(a) →
            print_commented_typing(a),
        value_def_from_explicit_value_def(a) →
            print_explicit_value_def(a),
        value_def_from_implicit_value_def(a) →
            print_implicit_value_def(a),
        value_def_from_explicit_function_def(a) →
            print_explicit_function_def(a),
        value_def_from_implicit_function_def(a) →
            print_implicit_function_def(a)
    end,

-- [p. 281]
print_explicit_value_def : explicit_value_def → Text
print_explicit_value_def(x) ≡
    case (x) of
        mk_explicit_value_def(a, b) →
            print_single_typing(a) ⌢ ″ = ″ ⌢ print_value_expr(b)
    end,

-- [p. 281]
print_implicit_value_def : implicit_value_def → Text
print_implicit_value_def(x) ≡
    case (x) of
        mk_implicit_value_def(a, b) →
            print_single_typing(a) ⌢ ″ ″ ⌢ print_restriction(b)
    end,

-- [p. 282]
print_explicit_function_def :
    explicit_function_def → Text
print_explicit_function_def(x) ≡
    case (x) of
        mk_explicit_function_def(a, b, c, d) →
            print_single_typing(a) ⌢ ″ ″ ⌢
            print_formal_function_application(b) ⌢ ″ is ″ ⌢
            print_value_expr(c) ⌢ print_opt_pre_condition(d)
    end,

print_opt_pre_condition : opt_pre_condition → Text
print_opt_pre_condition(x) ≡
    case x of
        opt_prec_none → ″″,
        opt_pre_condition_from_pre_condition(a) →
            ″ ″ ⌢ print_pre_condition(a)
    end,
```

```
- - [p. 282]
print_formal_function_application :
    formal_function_application → Text
print_formal_function_application(x) ≡
    case x of
        formal_function_application_from_id_application(a) →
            print_id_application(a),
        formal_function_application_from_prefix_application(a) →
            print_prefix_application(a),
        formal_function_application_from_infix_application(a) →
            print_infix_application(a)
    end,

- - [p. 282]
print_id_application : id_application → Text
print_id_application(x) ≡
    case (x) of
        mk_id_application(a, b) →
            print_id(a) ^ "(" ^
            print_formal_function_parameter_list(b)
    end,

print_formal_function_parameter_list :
    formal_function_parameter* → Text
print_formal_function_parameter_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ →
            "(" ^ print_formal_function_parameter(a) ^ ")",
        ⟨a⟩ ^ b →
            "(" ^ print_formal_function_parameter(a) ^ ")" ^
            print_formal_function_parameter_list(b)
    end,

- - [p. 282]
print_formal_function_parameter :
    formal_function_parameter → Text
print_formal_function_parameter(x) ≡
    case (x) of
        mk_formal_function_parameter(a) →
            "(" ^ print_binding_list(a) ^ ") "
    end,

- - [p. 282]
print_prefix_application : prefix_application → Text
print_prefix_application(x) ≡
    case (x) of
        mk_prefix_application(a, b) →
            print_prefix_op(a) ^ "(" ^ print_id(b)
    end,

- - [p. 282]
print_infix_application : infix_application → Text
print_infix_application(x) ≡
    case (x) of
        mk_infix_application(a, b, c) →
            print_id(a) ^ print_infix_op(b) ^ print_id(c)
```

```
    end,

-- [p. 284]
print_implicit_function_def :
    implicit_function_def → Text
print_implicit_function_def(x) ≡
    case (x) of
        mk_implicit_function_def(a, b, c, d) →
            print_single_typing(a) ⌢ ″ ″ ⌢
            print_formal_function_application(b) ⌢ ″ ″ ⌢
            print_post_condition(c) ⌢ print_opt_pre_condition(d)
    end,

-- [p. 287]
print_variable_decl : variable_decl → Text
print_variable_decl(x) ≡
    case (x) of
        mk_variable_decl(a) →
            ″variable ″ ⌢ print_variable_def_list(a)
    end,

print_variable_def_list : variable_def* → Text
print_variable_def_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_variable_def(a),
        ⟨a⟩ ⌢ b →
            print_variable_def(a) ⌢ ″, \n ″ ⌢
            print_variable_def_list(b)
    end,

-- [p. 287]
print_variable_def : variable_def → Text
print_variable_def(x) ≡
    case x of
        variable_def_from_single_variable_def(a) →
            print_single_variable_def(a),
        variable_def_from_multiple_variable_def(a) →
            print_multiple_variable_def(a)
    end,

-- [p. 287]
print_single_variable_def : single_variable_def → Text
print_single_variable_def(x) ≡
    case (x) of
        mk_single_variable_def(a, b, c) →
            print_id(a) ⌢ ″ : ″ ⌢ print_type_expr(b) ⌢
            print_opt_initialisation(c)
    end,

print_opt_initialisation : opt_initialisation → Text
print_opt_initialisation(x) ≡
    case x of
        opt_init_none → ″″,
        opt_initialisation_from_initialisation(a) →
            ″ ″ ⌢ print_initialisation(a)
    end,
```

```
print_initialisation : initialisation → Text
print_initialisation(x) ≡ ″:= ″ ⌢ print_value_expr(x),

- - [P. 287]
print_multiple_variable_def :
    multiple_variable_def → Text
print_multiple_variable_def(x) ≡
    case (x) of
        mk_multiple_variable_def(a, b) →
            print_id_list(a) ⌢ ″ : ″ ⌢ print_type_expr(b)
    end,

- - [p. 288]
print_channel_decl : channel_decl → Text
print_channel_decl(x) ≡
    case (x) of
        mk_channel_decl(a) →
            ″channel ″ ⌢ print_channel_def_list(a)
    end,

print_channel_def_list : channel_def* → Text
print_channel_def_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_channel_def(a),
        ⟨a⟩ ⌢ b →
            print_channel_def(a) ⌢ ″, \n ″ ⌢
            print_channel_def_list(b)
    end,

- - [p.288]
print_channel_def : channel_def → Text
print_channel_def(x) ≡
    case x of
        channel_def_from_single_channel_def(a) →
            print_single_channel_def(a),
        channel_def_from_multiple_channel_def(a) →
            print_multiple_channel_def(a)
    end,

- - [p.288]
print_single_channel_def : single_channel_def → Text
print_single_channel_def(x) ≡
    case (x) of
        mk_single_channel_def(a, b) →
            print_id(a) ⌢ ″ : ″ ⌢ print_type_expr(b)
    end,

- - [p.288]
print_multiple_channel_def : multiple_channel_def → Text
print_multiple_channel_def(x) ≡
    case (x) of
        mk_multiple_channel_def(a, b) →
            print_id_list(a) ⌢ ″ : ″ ⌢ print_type_expr(b)
    end,

- - [p. 289]
print_axiom_decl : axiom_decl → Text
```

```
print_axiom_decl(x) ≡
    case (x) of
        mk_axiom_decl(a) → "axiom " ^ print_axiom_def_list(a)
    end,

print_axiom_def_list : axiom_def* → Text
print_axiom_def_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_axiom_def(a),
        ⟨a⟩ ^ b →
            print_axiom_def(a) ^ ",\n " ^
            print_axiom_def_list(b)
    end,

-- [p. 289]
print_axiom_def : axiom_def → Text
print_axiom_def(x) ≡
    case (x) of
        mk_axiom_def(a, b) →
            print_opt_axiom_naming(a) ^ print_value_expr(b)
    end,

print_opt_axiom_naming : opt_axiom_naming → Text
print_opt_axiom_naming(x) ≡
    case x of
        opt_axio_none → "",
        opt_axiom_naming_from_axiom_naming(a) →
            print_axiom_naming(a) ^ " "
    end,

-- [p.289]
print_axiom_naming : axiom_naming → Text
print_axiom_naming(x) ≡
    case (x) of
        mk_axiom_naming(a) → "[" ^ print_id(a) ^ "]"
    end,

-- [p.291]
print_class_expr : class_expr → Text
print_class_expr(x) ≡
    case x of
        class_expr_from_basic_class_expr(a) →
            print_basic_class_expr(a),
        class_expr_from_extending_class_expr(a) →
            print_extending_class_expr(a),
        class_expr_from_hiding_class_expr(a) →
            print_hiding_class_expr(a),
        class_expr_from_renaming_class_expr(a) →
            print_renaming_class_expr(a),
        class_expr_from_with_class_expr(a) →
            print_with_class_expr(a),
        class_expr_from_scheme_instantiation(a) →
            print_scheme_instantiation(a)
    end,

-- [p.292]
print_basic_class_expr : basic_class_expr → Text
```

```
print_basic_class_expr(x) ≡
    case (x) of
        mk_basic_class_expr(a) →
            ″class " ^ print_decl_list(a) ^ " end″
    end,

- - [p.292]
print_extending_class_expr : extending_class_expr → Text
print_extending_class_expr(x) ≡
    case (x) of
        mk_extending_class_expr(a, b) →
            ″extend " ^ print_class_expr(a) ^ " with ″ ^
            print_class_expr(b)
    end,

- - [p.293]
print_hiding_class_expr : hiding_class_expr → Text
print_hiding_class_expr(x) ≡
    case (x) of
        mk_hiding_class_expr(a, b) →
            ″hide " ^ print_defined_item_list(a) ^ " in ″ ^
            print_class_expr(b)
    end,

- - [p.293]
print_renaming_class_expr : renaming_class_expr → Text
print_renaming_class_expr(x) ≡
    case (x) of
        mk_renaming_class_expr(a, b) →
            ″use " ^ print_rename_pair_list(a) ^ " in ″ ^
            print_class_expr(b)
    end,

- - [p. NEW]
print_with_class_expr : with_class_expr → Text
print_with_class_expr(x) ≡
    case (x) of
        mk_with_class_expr(a, b) →
            ″with " ^ print_object_expr_list(a) ^ " in ″ ^
            print_class_expr(b)
    end,

- - [p.294]
print_scheme_instantiation : scheme_instantiation → Text
print_scheme_instantiation(x) ≡
    case (x) of
        mk_scheme_instantiation(a, b) →
            print_name(a) ^ print_opt_actual_scheme_parameter(b)
    end,

print_opt_actual_scheme_parameter :
    opt_actual_scheme_parameter → Text
print_opt_actual_scheme_parameter(x) ≡
    case x of
        opt_asp_none → ″″,
        opt_actual_scheme_parameter_from_actual_scheme_parameter(
            a) →
            ″ ″ ^ print_actual_scheme_parameter(a)
```

```
    end,

- - [p.294]
print_actual_scheme_parameter :
    actual_scheme_parameter → Text
print_actual_scheme_parameter(x) ≡
    case (x) of
        mk_actual_scheme_parameter(a) →
            ″(" ^ print_object_expr_list(a) ^ ")″
    end,

print_rename_pair_list : rename_pair* → Text
print_rename_pair_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_rename_pair(a),
        ⟨a⟩ ⌢ b →
            print_rename_pair(a) ⌢ ″, ″ ⌢
            print_rename_pair_list(b)
    end,

- -[p.297]
print_rename_pair : rename_pair → Text
print_rename_pair(x) ≡
    case (x) of
        mk_rename_pair(a, b) →
            print_defined_item(a) ⌢ ″ for ″ ⌢
            print_defined_item(b)
    end,

print_defined_item_list : defined_item* → Text
print_defined_item_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_defined_item(a),
        ⟨a⟩ ⌢ b →
            print_defined_item(a) ⌢ ″, ″ ⌢
            print_defined_item_list(b)
    end,

- - [p.297]
print_defined_item : defined_item → Text
print_defined_item(x) ≡
    case x of
        defined_item_from_id_or_op(a) → print_id_or_op(a),
        defined_item_from_disambiguated_item(a) →
            print_disambiguated_item(a)
    end,

- - [p.297]
print_disambiguated_item : disambiguated_item → Text
print_disambiguated_item(x) ≡
    case (x) of
        mk_disambiguated_item(a, b) →
            print_id_or_op(a) ⌢ ″ : ″ ⌢ print_type_expr(b)
    end,

print_object_expr_list : object_expr* → Text
```

```
print_object_expr_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ⌢ ⟨⟩ → print_object_expr(a),
        ⟨a⟩ ⌢ b →
            print_object_expr(a) ⌢ ", \n" ⌢
            print_object_expr_list(b)
    end,

--[p.299]
print_object_expr : object_expr → Text
print_object_expr(x) ≡
    case x of
        object_expr_from_object_name(a) →
            print_object_name(a),
        object_expr_from_element_object_expr(a) →
            print_element_object_expr(a),
        object_expr_from_array_object_expr(a) →
            print_array_object_expr(a),
        object_expr_from_fitting_object_expr(a) →
            print_fitting_object_expr(a)
    end,

print_object_name : object_name → Text
print_object_name(x) ≡ print_name(x),

--[p.300]
print_element_object_expr : element_object_expr → Text
print_element_object_expr(x) ≡
    case (x) of
        mk_element_object_expr(a, b) →
            print_object_expr(a) ⌢ " " ⌢
            print_actual_array_parameter(b)
    end,

--[p.300]
print_actual_array_parameter :
    actual_array_parameter → Text
print_actual_array_parameter(x) ≡
    case (x) of
        mk_actual_array_parameter(a) →
            "[" ^ print_value_expr_list(a) ^ "]"
    end,

--[p.300]
print_array_object_expr : array_object_expr → Text
print_array_object_expr(x) ≡
    case (x) of
        mk_array_object_expr(a, b) →
            "[|" ^ print_typing_list(a) ^ " :- " ⌢
            print_object_expr(b) ⌢ "|]"
    end,

-- [p.301]
print_fitting_object_expr : fitting_object_expr → Text
print_fitting_object_expr(x) ≡
    case (x) of
        mk_fitting_object_expr(a, b) →
```

```
            print_object_expr(a) ⌢ ″{″ ⌢
            print_rename_pair_list(b) ⌢ ″}″
    end,

print_type_expr_list : type_expr* → Text
print_type_expr_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_type_expr(a),
        ⟨a⟩ ⌢ b →
            print_type_expr(a) ⌢ ″ >< ″ ⌢
            print_type_expr_list(b)
    end,

-- ?? [p.302]
print_type_expr : type_expr → Text
print_type_expr(x) ≡
    case x of
        type_expr_from_type_literal(a) →
            print_type_literal(a),
        type_expr_from_name(a) → print_name(a),
        type_expr_from_product_type_expr(a) →
            print_product_type_expr(a),
        type_expr_from_set_type_expr(a) →
            print_set_type_expr(a),
        type_expr_from_list_type_expr(a) →
            print_list_type_expr(a),
        type_expr_from_map_type_expr(a) →
            print_map_type_expr(a),
        type_expr_from_function_type_expr(a) →
            print_function_type_expr(a),
        type_expr_from_subtype_expr(a) →
            print_subtype_expr(a),
        type_expr_from_bracketed_type_expr(a) →
            print_bracketed_type_expr(a)
    end,

-- [p.302]
print_type_literal : type_literal → Text
print_type_literal(x) ≡
    case x of
        tl_Unit → ″Unit″,
        tl_Bool → ″Bool″,
        tl_Int → ″Int″,
        tl_Nat → ″Nat″,
        tl_Real → ″Real″,
        tl_Text → ″Text″,
        tl_Char → ″Char″
    end,

-- [p.305]
print_product_type_expr : product_type_expr → Text
print_product_type_expr(x) ≡
    case (x) of
        mk_product_type_expr(a) → print_type_expr_list(a)
    end,

-- [p.305]
```

```
print_set_type_expr : set_type_expr → Text
print_set_type_expr(x) ≡
    case x of
        set_type_expr_from_finite_set_type_expr(a) →
            print_finite_set_type_expr(a),
        set_type_expr_from_infinite_set_type_expr(a) →
            print_infinite_set_type_expr(a)
    end,

- - [p.305]
print_finite_set_type_expr : finite_set_type_expr → Text
print_finite_set_type_expr(x) ≡
    case (x) of
        mk_finite_set_type_expr(a) →
            print_type_expr(a) ^ "-set"
    end,

- - [p.305]
print_infinite_set_type_expr :
    infinite_set_type_expr → Text
print_infinite_set_type_expr(x) ≡
    case (x) of
        mk_infinite_set_type_expr(a) →
            print_type_expr(a) ^ "-infset"
    end,

- - [p.306]
print_list_type_expr : list_type_expr → Text
print_list_type_expr(x) ≡
    case x of
        list_type_expr_from_finite_list_type_expr(a) →
            print_finite_list_type_expr(a),
        list_type_expr_from_infinite_list_type_expr(a) →
            print_infinite_list_type_expr(a)
    end,

- - [p.306]
print_finite_list_type_expr :
    finite_list_type_expr → Text
print_finite_list_type_expr(x) ≡
    case (x) of
        mk_finite_list_type_expr(a) →
            print_type_expr(a) ^ "-list"
    end,

- - [p.306]
print_infinite_list_type_expr :
    infinite_list_type_expr → Text
print_infinite_list_type_expr(x) ≡
    case (x) of
        mk_infinite_list_type_expr(a) →
            print_type_expr(a) ^ "-inflist"
    end,

- - [p.306]
print_map_type_expr : map_type_expr → Text
print_map_type_expr(x) ≡
    case x of
```

```
        map_type_expr_from_finite_map_type_expr(a) →
            print_finite_map_type_expr(a),
        map_type_expr_from_infinite_map_type_expr(a) →
            print_infinite_map_type_expr(a)
    end,

- - [p.306]
print_finite_map_type_expr : finite_map_type_expr → Text
print_finite_map_type_expr(x) ≡
    case (x) of
        mk_finite_map_type_expr(a, b) →
            print_type_expr(a) ^ " -m->" ^ print_type_expr(b)
    end,

- - [p.306]
print_infinite_map_type_expr :
    infinite_map_type_expr → Text
print_infinite_map_type_expr(x) ≡
    case (x) of
        mk_infinite_map_type_expr(a, b) →
            print_type_expr(a) ^ " -~m-> " ^ print_type_expr(b)
    end,

- - [p.307]
print_function_type_expr : function_type_expr → Text
print_function_type_expr(x) ≡
    case (x) of
        mk_function_type_expr(a, b, c) →
            print_type_expr(a) ^ " " ^ print_function_arrow(b) ^
            " " ^ print_result_desc(c)
    end,

print_function_arrow : function_arrow → Text
print_function_arrow(x) ≡
    case x of
        fa_total → "->",
        fa_partial → "-~->"
    end,

- - [p.307]
print_result_desc : result_desc → Text
print_result_desc(x) ≡
    case (x) of
        mk_result_desc(a, b) →
            print_accss_desc_list(a) ^ print_type_expr(b)
    end,

- - [p.308]
print_subtype_expr : subtype_expr → Text
print_subtype_expr(x) ≡
    case (x) of
        mk_subtype_expr(a, b) →
            "{|" ^ print_single_typing(a) ^ " " ^
            print_restriction(b) ^ "|}"
    end,

- - [p.309]
print_bracketed_type_expr : bracketed_type_expr → Text
```

```
print_bracketed_type_expr(x) ≡
    case (x) of
        mk_bracketed_type_expr(a) →
            "(" ^ print_type_expr(a) ^ ")"
    end,

print_accss_desc_list : accss_desc* → Text
print_accss_desc_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_accss_desc(a) ^ " ",
        ⟨a⟩ ^ b →
            print_accss_desc(a) ^ " " ^ print_accss_desc_list(b)
    end,

- - [p.309]
print_accss_desc : accss_desc → Text
print_accss_desc(x) ≡
    case (x) of
        mk_accss_desc(a, b) →
            print_accss_mode(a) ^ " " ^ print_accss_list(b)
    end,

- - [p.309]
print_accss_mode : accss_mode → Text
print_accss_mode(x) ≡
    case x of
        am_read → "read ",
        am_write → "write ",
        am_in → "in",
        am_out → "out"
    end,

print_accss_list : accss* → Text
print_accss_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_accss(a),
        ⟨a⟩ ^ b →
            print_accss(a) ^ " " ^ print_accss_list(b)
    end,

- - [p.309]
print_accss : accss → Text
print_accss(x) ≡
    case x of
        accss_from_acc_name(a) → print_acc_name(a),
        accss_from_enumerated_accss(a) →
            print_enumerated_accss(a),
        accss_from_completed_accss(a) →
            print_completed_accss(a),
        accss_from_comprehended_accss(a) →
            print_comprehended_accss(a)
    end,

print_acc_name : acc_name → Text
print_acc_name(x) ≡ print_name(x),
```

```
-- [p.309]
print_enumerated_accss : enumerated_accss → Text
print_enumerated_accss(x) ≡
    case (x) of
        mk_enumerated_accss(a) →
            "{" ^ print_accss_list(a) ^ "}"
    end,

-- [p.309]
print_completed_accss : completed_accss → Text
print_completed_accss(x) ≡
    case (x) of
        mk_completed_accss(a) →
            print_opt_qualification(a) ^ "any"
    end,

print_opt_qualification : opt_qualification → Text
print_opt_qualification(x) ≡
    case x of
        opt_qual_none → "",
        opt_qualification_from_qualification(a) →
            print_qualification(a)
    end,

-- [p.309]
print_comprehended_accss : comprehended_accss → Text
print_comprehended_accss(x) ≡
    case (x) of
        mk_comprehended_accss(a, b) →
            "{" ^ print_accss(a) ^ " | " ^
            print_set_limitation(b) ^ "}"
    end,

print_value_expr_list : value_expr* → Text
print_value_expr_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_value_expr(a),
        ⟨a⟩ ^ b →
            print_value_expr(a) ^ ", " ^
            print_value_expr_list(b)
    end,

-- [p.312]
print_value_expr : value_expr → Text
print_value_expr(x) ≡
    case x of
        ve_val_l(a) → print_value_literal(a),
        ve_name(a) → print_name(a),
        ve_pren(a) → print_pre_name(a),
        ve_bas_e(a) → print_basic_expr(a),
        ve_pro_e(a) → print_product_expr(a),
        ve_set_e(a) → print_set_expr(a),
        ve_lis_e(a) → print_list_expr(a),
        ve_map_e(a) → print_map_expr(a),
        ve_fun_e(a) → print_function_expr(a),
        ve_app_e(a) → print_application_expr(a),
        ve_qua_e(a) → print_quantified_expr(a),
```

```
        ve_equ_e(a) → print_equivalence_expr(a),
        ve_pos_e(a) → print_post_expr(a),
        ve_dis_e(a) → print_disambiguation_expr(a),
        ve_bra_e(a) → print_bracketed_expr(a),
        ve_inf_e(a) → print_infix_expr(a),
        ve_pre_e(a) → print_prefix_expr(a),
        ve_com_e(a) → print_comprehended_expr(a),
        ve_ini_e(a) → print_initialise_expr(a),
        ve_ass_e(a) → print_assignment_expr(a),
        ve_inp_e(a) → print_input_expr(a),
        ve_out_e(a) → print_output_expr(a),
        ve_str_e(a) → print_structured_expr(a)
    end,

-- [p.315]
print_value_literal : value_literal → Text
print_value_literal(x) ≡
    case x of
        unit_literal → "()",
        bool_literal(a) →
            case a of
                true → "true",
                false → "false"
            end,
        int_literal(a) → RSL_int_to_string(a),
        real_literal(a) → RSL_double_to_string(a),
        text_literal(a) → a,
        char_literal(a) → ⟨a⟩
    end,

-- [p.316]
print_pre_name : pre_name → Text
print_pre_name(x) ≡ print_name(x),

-- [p.316]
print_basic_expr : basic_expr → Text
print_basic_expr(x) ≡
    case x of
        be_chaos → "chaos",
        be_skip → "skip",
        be_stop → "stop",
        be_swap → "swap"
    end,

-- [p.316]
print_product_expr : product_expr → Text
print_product_expr(x) ≡
    case (x) of
        mk_product_expr(a) →
            "(" ^ print_value_expr_list(a) ^ ")"
    end,

-- [p.317]
print_set_expr : set_expr → Text
print_set_expr(x) ≡
    case x of
        set_expr_from_ranged_set_expr(a) →
            print_ranged_set_expr(a),
```

```
        set_expr_from_enumerated_set_expr(a) →
            print_enumerated_set_expr(a),
        set_expr_from_comprehended_set_expr(a) →
            print_comprehended_set_expr(a)
    end,

-- [p.317]
print_ranged_set_expr : ranged_set_expr → Text
print_ranged_set_expr(x) ≡
    case (x) of
        mk_ranged_set_expr(a, b) →
            ″{" ^ print_value_expr(a) ^ " .. ″ ^
            print_value_expr(b) ^ ″}″
    end,

-- [p.317]
print_enumerated_set_expr : enumerated_set_expr → Text
print_enumerated_set_expr(x) ≡
    case (x) of
        mk_enumerated_set_expr(a) →
            ″{" ^ print_value_expr_list(a) ^ "}″
    end,

-- [p.318]
print_comprehended_set_expr :
    comprehended_set_expr → Text
print_comprehended_set_expr(x) ≡
    case (x) of
        mk_comprehended_set_expr(a, b) →
            ″{" ^ print_value_expr(a) ^ " | ″ ^
            print_set_limitation(b) ^ ″}″
    end,

-- [p.318]
print_set_limitation : set_limitation → Text
print_set_limitation(x) ≡
    case (x) of
        mk_set_limitation(a, b) →
            print_typing_list(a) ^ print_opt_restriction(b)
    end,

print_opt_restriction : opt_restriction → Text
print_opt_restriction(x) ≡
    case x of
        opt_rest_none → ″″,
        opt_restriction_from_restriction(a) →
            ″ ″ ^ print_restriction(a)
    end,

-- [p. 318]
print_restriction : restriction → Text
print_restriction(x) ≡
    case (x) of
        mk_restriction(a) → ″: – ″ ^ print_value_expr(a)
    end,

-- [p.319]
print_list_expr : list_expr → Text
```

```
print_list_expr(x) ≡
    case x of
        list_expr_from_ranged_list_expr(a) →
            print_ranged_list_expr(a),
        list_expr_from_enumerated_list_expr(a) →
            print_enumerated_list_expr(a),
        list_expr_from_comprehended_list_expr(a) →
            print_comprehended_list_expr(a)
    end,

- - [p.319]
print_ranged_list_expr : ranged_list_expr → Text
print_ranged_list_expr(x) ≡
    case (x) of
        mk_ranged_list_expr(a, b) →
            ″<." ^ print_value_expr(a) ^ " .. ″^
            print_value_expr(b) ^″.>″
    end,

- - [p.319]
print_enumerated_list_expr : enumerated_list_expr → Text
print_enumerated_list_expr(x) ≡
    case (x) of
        mk_enumerated_list_expr(a) →
            ″<." ^ print_value_expr_list(a) ^ ".>″
    end,

- - [p.320]
print_comprehended_list_expr :
    comprehended_list_expr → Text
print_comprehended_list_expr(x) ≡
    case (x) of
        mk_comprehended_list_expr(a, b) →
            ″<." ^ print_value_expr(a) ^ " | ″^
            print_list_limitation(b) ^″.>″
    end,

- - [p.320]
print_list_limitation : list_limitation → Text
print_list_limitation(x) ≡
    case (x) of
        mk_list_limitation(a, b, c) →
            print_binding(a) ^″ in ″^ print_value_expr(b) ^
            print_opt_restriction(c)
    end,

- - [p.321]
print_map_expr : map_expr → Text
print_map_expr(x) ≡
    case x of
        map_expr_from_enumerated_map_expr(a) →
            print_enumerated_map_expr(a),
        map_expr_from_comprehended_map_expr(a) →
            print_comprehended_map_expr(a)
    end,

- - [p.321]
print_enumerated_map_expr : enumerated_map_expr → Text
```

```
print_enumerated_map_expr(x) ≡
    case (x) of
        mk_enumerated_map_expr(a) →
            "[" ^ print_value_expr_pair_list(a) ^ "]"
    end,

print_value_expr_pair_list : value_expr_pair* → Text
print_value_expr_pair_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_value_expr_pair(a),
        ⟨a⟩ ^ b →
            print_value_expr_pair(a) ^ ", " ^
            print_value_expr_pair_list(b)
    end,

-- [p.321]
print_value_expr_pair : value_expr_pair → Text
print_value_expr_pair(x) ≡
    case (x) of
        mk_value_expr_pair(a, b) →
            print_value_expr(a) ^ " +> " ^ print_value_expr(b)
    end,

-- [p.322]
print_comprehended_map_expr :
    comprehended_map_expr → Text
print_comprehended_map_expr(x) ≡
    case (x) of
        mk_comprehended_map_expr(a, b) →
            "[" ^ print_value_expr_pair(a) ^ " | " ^
            print_set_limitation(b) ^ "]"
    end,

-- [p.322]
print_function_expr : function_expr → Text
print_function_expr(x) ≡
    case (x) of
        mk_function_expr(a, b) →
            "-\\ " ^ print_lambda_parameter(a) ^ " :- " ^
            print_value_expr(b)
    end,

-- [p.322]
print_lambda_parameter : lambda_parameter → Text
print_lambda_parameter(x) ≡
    case x of
        lambda_parameter_from_lambda_typing(a) →
            print_lambda_typing(a),
        lambda_parameter_from_single_typing(a) →
            print_single_typing(a)
    end,

-- [p.322]
print_lambda_typing : lambda_typing → Text
print_lambda_typing(x) ≡
    case (x) of
        mk_lambda_typing(a) →
```

```
            ″(" ^ print_typing_list(a) ^ ")″
    end,

- - [p.323]
print_application_expr : application_expr → Text
print_application_expr(x) ≡
    case (x) of
        mk_application_expr(a, b) →
            print_value_expr(a) ^ ″ ″ ^
            print_actual_function_parameter(b)
    end,

- - [p.324]
print_actual_function_parameter :
    actual_function_parameter → Text
print_actual_function_parameter(x) ≡
    case (x) of
        mk_actual_function_parameter(a) →
            ″(" ^ print_value_expr_list(a) ^ ")″
    end,

- - [p.325]
print_quantified_expr : quantified_expr → Text
print_quantified_expr(x) ≡
    case (x) of
        mk_quantified_expr(a, b, c) →
            print_quantifier(a) ^ ″ ″ ^ print_typing_list(b) ^
            ″ ″ ^ print_restriction(c)
    end,

- - [p.322]
print_quantifier : quantifier → Text
print_quantifier(x) ≡
    case x of
        qu_all → ″all ″,
        qu_exists → ″exists ″,
        qu_existsem → ″exists! ″
    end,

- - [p.326]
print_equivalence_expr : equivalence_expr → Text
print_equivalence_expr(x) ≡
    case (x) of
        mk_equivalence_expr(a, b, c) →
            print_value_expr(a) ^ ″ is ″ ^ print_value_expr(b) ^
            print_opt_pre_condition(c)
    end,

- - [p. 326]
print_pre_condition : pre_condition → Text
print_pre_condition(x) ≡
    case (x) of
        mk_pre_condition(a) → ″pre ″ ^ print_value_expr(a)
    end,

- - [p. 327]
print_post_expr : post_expr → Text
print_post_expr(x) ≡
```

```
    case (x) of
        mk_post_expr(a, b, c) →
            print_value_expr(a) ^ ″ ″ ^
            print_post_condition(b) ^ print_opt_pre_condition(c)
    end,

-- [p. 327]
print_post_condition : post_condition → Text
print_post_condition(x) ≡
    case (x) of
        mk_post_condition(a, b) →
            print_opt_result_naming(a) ^ ″post ″ ^
            print_value_expr(b)
    end,

print_opt_result_naming : opt_result_naming → Text
print_opt_result_naming(x) ≡
    case x of
        opt_resn_none → ″″,
        opt_result_naming_from_result_naming(a) →
            print_result_naming(a) ^ ″ ″
    end,

print_result_naming : result_naming → Text
print_result_naming(x) ≡
    case (x) of
        mk_result_naming(a) → ″as ″ ^ print_binding(a)
    end,

-- [p.328]
print_disambiguation_expr : disambiguation_expr → Text
print_disambiguation_expr(x) ≡
    case (x) of
        mk_disambiguation_expr(a, b) →
            print_value_expr(a) ^ ″ : ″ ^ print_type_expr(b)
    end,

-- [p.328]
print_bracketed_expr : bracketed_expr → Text
print_bracketed_expr(x) ≡
    case (x) of
        mk_bracketed_expr(a) →
            ″(" ^ print_value_expr(a) ^ ")″
    end,

--[p.329]
print_infix_expr : infix_expr → Text
print_infix_expr(x) ≡
    case x of
        infix_expr_from_stmt_infix_expr(a) →
            print_stmt_infix_expr(a),
        infix_expr_from_axiom_infix_expr(a) →
            print_axiom_infix_expr(a),
        infix_expr_from_value_infix_expr(a) →
            print_value_infix_expr(a)
    end,

print_stmt_infix_expr : stmt_infix_expr → Text
```

```
print_stmt_infix_expr(x) ≡
    case (x) of
        mk_stmt_infix_expr(a, b, c) →
            print_value_expr(a) ^ print_infix_combinator(b) ^
            print_value_expr(c)
    end,

print_axiom_infix_expr : axiom_infix_expr → Text
print_axiom_infix_expr(x) ≡
    case (x) of
        mk_axiom_infix_expr(a, b, c) →
            print_value_expr(a) ^ ″ ″ ^
            print_infix_connective(b) ^ ″ ″ ^
            print_value_expr(c)
    end,

print_value_infix_expr : value_infix_expr → Text
print_value_infix_expr(x) ≡
    case (x) of
        mk_value_infix_expr(a, b, c) →
            print_value_expr(a) ^ print_infix_op(b) ^
            print_value_expr(c)
    end,

- -[p.330]
print_prefix_expr : prefix_expr → Text
print_prefix_expr(x) ≡
    case x of
        prefix_expr_from_axiom_prefix_expr(a) →
            print_axiom_prefix_expr(a),
        prefix_expr_from_universal_prefix_expr(a) →
            print_universal_prefix_expr(a),
        prefix_expr_from_value_prefix_expr(a) →
            print_value_prefix_expr(a)
    end,

print_axiom_prefix_expr : axiom_prefix_expr → Text
print_axiom_prefix_expr(x) ≡
    case (x) of
        mk_axiom_prefix_expr(a, b) →
            print_prefix_connective(a) ^ print_value_expr(b)
    end,

print_universal_prefix_expr :
    universal_prefix_expr → Text
print_universal_prefix_expr(x) ≡
    case (x) of
        mk_universal_prefix_expr(a) →
            ″always ″ ^ print_value_expr(a)
    end,

print_value_prefix_expr : value_prefix_expr → Text
print_value_prefix_expr(x) ≡
    case (x) of
        mk_value_prefix_expr(a, b) →
            print_prefix_op(a) ^ print_value_expr(b)
    end,
```

```
-- [p.331]
print_comprehended_expr : comprehended_expr → Text
print_comprehended_expr(x) ≡
    case (x) of
        mk_comprehended_expr(a, b, c) →
            print_infix_combinator(a) ^ "{" ^
            print_value_expr(b) ^ " | " ^
            print_set_limitation(c) ^ "} "
    end,

--[p.332]
print_initialise_expr : initialise_expr → Text
print_initialise_expr(x) ≡
    case (x) of
        mk_initialise_expr(a) →
            print_opt_qualification(a) ^ "initialise"
    end,

-- [p.332]
print_assignment_expr : assignment_expr → Text
print_assignment_expr(x) ≡
    case (x) of
        mk_assignment_expr(a, b) →
            print_name(a) ^ " := " ^ print_value_expr(b)
    end,

-- [p.331]
print_input_expr : input_expr → Text
print_input_expr(x) ≡
    case (x) of
        mk_input_expr(a) → print_name(a) ^ "?"
    end,

-- [p.331]
print_output_expr : output_expr → Text
print_output_expr(x) ≡
    case (x) of
        mk_output_expr(a, b) →
            print_name(a) ^ "!" ^ print_value_expr(b)
    end,

-- [p.334]
print_structured_expr : structured_expr → Text
print_structured_expr(x) ≡
    case x of
        structured_expr_from_local_expr(a) →
            print_local_expr(a),
        structured_expr_from_let_expr(a) → print_let_expr(a),
        structured_expr_from_if_expr(a) → print_if_expr(a),
        structured_expr_from_case_expr(a) →
            print_case_expr(a),
        structured_expr_from_while_expr(a) →
            print_while_expr(a),
        structured_expr_from_until_expr(a) →
            print_until_expr(a),
        structured_expr_from_for_expr(a) → print_for_expr(a)
    end,
```

```
-- [p.334]
print_local_expr : local_expr → Text
print_local_expr(x) ≡
    case (x) of
        mk_local_expr(a, b) →
            ″local " ^ print_decl_list(a) ^ " in ″ ⌢
            print_value_expr(b) ⌢ ″ end″
    end,

-- [p.335]
print_let_expr : let_expr → Text
print_let_expr(x) ≡
    case (x) of
        mk_let_expr(a, b) →
            ″let " ^ print_let_def(a) ^ " in ″ ⌢
            print_value_expr(b) ⌢ ″ end″
    end,

-- [p.335]
print_let_def : let_def → Text
print_let_def(x) ≡
    case x of
        let_def_from_typing(a) → print_typing(a),
        let_def_from_explicit_let(a) → print_explicit_let(a),
        let_def_from_implicit_let(a) → print_implicit_let(a)
    end,

-- [p.335]
print_explicit_let : explicit_let → Text
print_explicit_let(x) ≡
    case (x) of
        mk_explicit_let(a, b) →
            print_let_binding(a) ⌢ ″ = ″ ⌢ print_value_expr(b)
    end,

-- [p.335]
print_implicit_let : implicit_let → Text
print_implicit_let(x) ≡
    case (x) of
        mk_implicit_let(a, b) →
            print_single_typing(a) ⌢ print_restriction(b)
    end,

-- [p.335]
print_let_binding : let_binding → Text
print_let_binding(x) ≡
    case x of
        let_binding_from_binding(a) → print_binding(a),
        let_binding_from_list_pattern(a) →
            print_list_pattern(a)
    end,

-- [p.336]
print_if_expr : if_expr → Text
print_if_expr(x) ≡
    case x of
        mk_if_expr(a, b, c, d) →
            ″if " ^ print_value_expr(a) ^ " then ″ ⌢
```

```
            print_value_expr(b) ⌢ ″ ″ ⌢
            print_elsif_branch_list(c) ⌢
            print_opt_else_branch(d) ⌢ ″end″
    end,

-- [p.336]
print_opt_else_branch : opt_else_branch → Text
print_opt_else_branch(x) ≡
    case x of
        opt_else_none → ″″,
        opt_else_branch_from_else_branch(a) →
            print_else_branch(a) ⌢ ″ ″
    end,

print_elsif_branch_list : elsif_branch* → Text
print_elsif_branch_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_elsif_branch(a) ⌢ ″ ″,
        ⟨a⟩ ⌢ b →
            print_elsif_branch(a) ⌢ ″ ″ ⌢
            print_elsif_branch_list(b)
    end,

-- [p.336]
print_elsif_branch : elsif_branch → Text
print_elsif_branch(x) ≡
    case x of
        mk_elsif_branch(a, b) →
            ″elsif " ^ print_value_expr(a) ^ " then ″ ⌢
            print_value_expr(b)
    end,

-- [p.336]
print_else_branch : else_branch → Text
print_else_branch(x) ≡
    case x of
        mk_else_branch(a) → ″else ″ ⌢ print_value_expr(a)
    end,

-- [p.337]
print_case_expr : case_expr → Text
print_case_expr(x) ≡
    case x of
        mk_case_expr(a, b) →
            ″case " ^ print_value_expr(a) ^ " of ″ ⌢
            print_case_branch_list(b) ⌢ ″ end″
    end,

print_case_branch_list : case_branch* → Text
print_case_branch_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_case_branch(a),
        ⟨a⟩ ⌢ b →
            print_case_branch(a) ⌢ ″, \n″ ⌢
            print_case_branch_list(b)
    end,
```

```
-- [p.338]
print_case_branch : case_branch → Text
print_case_branch(x) ≡
    case x of
        mk_case_branch(a, b) →
            print_pattern(a) ^ " -> " ^ print_value_expr(b)
    end,

-- [p.338]
print_while_expr : while_expr → Text
print_while_expr(x) ≡
    case x of
        mk_while_expr(a, b) →
            "while " ^ print_value_expr(a) ^ " do " ^
            print_value_expr(b)
    end,

-- [p.339]
print_until_expr : until_expr → Text
print_until_expr(x) ≡
    case x of
        mk_until_expr(a, b) →
            "do " ^ print_value_expr(a) ^ " until " ^
            print_value_expr(b)
    end,

-- [p.339]
print_for_expr : for_expr → Text
print_for_expr(x) ≡
    case x of
        mk_for_expr(a, b) →
            "for " ^ print_list_limitation(a) ^ " do " ^
            print_value_expr(b) ^ " end"
    end,

print_binding_list : binding* → Text
print_binding_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_binding(a),
        ⟨a⟩ ^ b →
            print_binding(a) ^ ", " ^ print_binding_list(b)
    end,

-- [p. 340]
print_binding : binding → Text
print_binding(x) ≡
    case x of
        binding_from_id_or_op(a) → print_id_or_op(a),
        binding_from_product_binding(a) →
            print_product_binding(a)
    end,

-- [p. 340]
print_product_binding : product_binding → Text
print_product_binding(x) ≡
    case x of
```

```
        mk_product_binding(a) → print_binding_list(a)
    end,

print_typing_list : typing* → Text
print_typing_list(x) ≡
    case x of
        ⟨⟩ → ″″,
        ⟨a⟩ ⌢ ⟨⟩ → print_typing(a),
        ⟨a⟩ ⌢ b →
            print_typing(a) ⌢ ″, ″ ⌢ print_typing_list(b)
    end,

-- [p. 342]
print_typing : typing → Text
print_typing(x) ≡
    case x of
        typing_from_single_typing(a) → print_single_typing(a),
        typing_from_multiple_typing(a) →
            print_multiple_typing(a)
    end,

-- [p. 342]
print_single_typing : single_typing → Text
print_single_typing(x) ≡
    case x of
        mk_single_typing(a, b) →
            print_binding(a) ⌢ ″ : ″ ⌢ print_type_expr(b)
    end,

-- [p. 342]
print_multiple_typing : multiple_typing → Text
print_multiple_typing(x) ≡
    case x of
        mk_multiple_typing(a, b) →
            print_binding_list(a) ⌢ ″ : ″ ⌢ print_type_expr(b)
    end,

-- [p. 342]
print_commented_typing : commented_typing → Text
print_commented_typing(x) ≡
    case x of
        mk_commented_typing(a) → print_typing(a)
    end,

-- [p. 344]
print_pattern : pattern → Text
print_pattern(x) ≡
    case x of
        pattern_from_value_literal(a) →
            print_value_literal(a),
        pattern_from_pv_name(a) → print_pv_name(a),
        pa_wildcard_pattern → ″_″,
        pattern_from_product_pattern(a) →
            print_product_pattern(a),
        pattern_from_record_pattern(a) →
            print_record_pattern(a),
        pattern_from_list_pattern(a) → print_list_pattern(a)
    end,
```

```
print_pv_name : pv_name → Text
print_pv_name(x) ≡ print_name(x),

- - [p. 346]
print_product_pattern : product_pattern → Text
print_product_pattern(x) ≡
    case x of
        mk_product_pattern(a) →
            "(" ^ print_inner_pattern_list(a) ^ ")"
    end,

- - [p. 346]
print_record_pattern : record_pattern → Text
print_record_pattern(x) ≡
    case x of
        mk_record_pattern(a, b) →
            print_name(a) ^ "(" ^ print_inner_pattern_list(b) ^
            ")"
    end,

- - [p. 347]
print_list_pattern : list_pattern → Text
print_list_pattern(x) ≡
    case x of
        list_pattern_from_enumerated_list_pattern(a) →
            print_enumerated_list_pattern(a),
        list_pattern_from_concatenated_list_pattern(a) →
            print_concatenated_list_pattern(a)
    end,

- - [p. 347]
print_enumerated_list_pattern :
    enumerated_list_pattern → Text
print_enumerated_list_pattern(x) ≡
    case x of
        mk_enumerated_list_pattern(x) →
            "<." ^ print_inner_pattern_list(x) ^ ".>"
    end,

print_inner_pattern_list : inner_pattern* → Text
print_inner_pattern_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ^ ⟨⟩ → print_inner_pattern(a),
        ⟨a⟩ ^ b →
            print_inner_pattern(a) ^ ", " ^
            print_inner_pattern_list(b)
    end,

- - [p. 348]
print_concatenated_list_pattern :
    concatenated_list_pattern → Text
print_concatenated_list_pattern(x) ≡
    case x of
        mk_concatenated_list_pattern(a, b) →
            "<." ^ print_enumerated_list_pattern(a) ^ ".>^" ^
            print_inner_pattern(b)
```

```
    end,

-- [p. 348]
print_inner_pattern : inner_pattern → Text
print_inner_pattern(x) ≡
    case x of
        inner_pattern_from_value_literal(a) →
            print_value_literal(a),
        inner_pattern_from_id_or_op(a) → print_id_or_op(a),
        ip_wildcard_pattern → "_",
        inner_pattern_from_product_pattern(a) →
            print_product_pattern(a),
        inner_pattern_from_record_pattern(a) →
            print_record_pattern(a),
        inner_pattern_from_list_pattern(a) →
            print_list_pattern(a),
        inner_pattern_from_equality_pattern(a) →
            print_equality_pattern(a)
    end,

-- [p. 349]
print_equality_pattern : equality_pattern → Text
print_equality_pattern(x) ≡
    case x of
        mk_equality_pattern(a) → print_name(a)
    end,

-- [p. 351]
print_name : name → Text
print_name(x) ≡
    case x of
        name_from_qualified_id(a) → print_qualified_id(a),
        name_from_qualified_op(a) → print_qualified_op(a)
    end,

print_qualified_id : qualified_id → Text
print_qualified_id(x) ≡
    case x of
        mk_qualified_id(a, b) →
            print_opt_qualification(a) ^ print_id(b)
    end,

-- [p. 351]
print_qualification : qualification → Text
print_qualification(x) ≡
    case x of
        mk_qualification(oe) → print_object_expr(oe) ^ "."
    end,

-- [p. 352]
print_qualified_op : qualified_op → Text
print_qualified_op(x) ≡
    case x of
        mk_qualified_op(a, b) →
            print_opt_qualification(a) ^ "(" ^ print_op(b) ^ ")"
    end,

-- [p. 353]
```

```
print_id_or_op : id_or_op → Text
print_id_or_op(x) ≡
    case x of
        id_or_op_from_id(a) → print_id(a),
        id_or_op_from_op(a) → print_op(a)
    end,

- - [p. 353]
print_op : op → Text
print_op(x) ≡
    case x of
        op_from_infix_op(a) → print_infix_op(a),
        op_from_prefix_op(a) → print_prefix_op(a)
    end,

- - adheres to book [p. 354] plus infix "==".
print_infix_op : infix_op → Text
print_infix_op(x) ≡
    case x of
        io_eq → "=",
        io_ieq → "~=",
        io_eqeq → "==",
        io_gt → ">",
        io_lt → "<",
        io_gteq → ">=",
        io_lteq → "<=",
        io_proper_superset → ">>",
        io_proper_subset → "<<",
        io_superset → ">>=",
        io_subset → "<<=",
        io_isin → " isin ",
        io_nisin → " ~isin ",
        io_plus → "+",
        io_minus → "-",
        io_remainder_diff_restr → "\\ ",
        io_concat → "^",
        io_union → " union ",
        io_override → "!!",
        io_mult → " * ",
        io_div → " / ",
        io_composition → "#",
        io_inter → " inter ",
        io_exponentation → "**"
    end,

- - [p. 359]
print_prefix_op : prefix_op → Text
print_prefix_op(x) ≡
    case x of
        po_minus → "minus",
        po_plus → "plus",
        po_abs → "abs",
        po_int → "int",
        po_real → "real",
        po_card → "card",
        po_len → "len",
        po_inds → "inds",
        po_elems → "elems",
```

```
        po_hd → "hd",
        po_tl → "tl",
        po_dom → "dom",
        po_rng → "rng"
    end,

- - [p. 362]
print_infix_connective : infix_connective → Text
print_infix_connective(x) ≡
    case x of
        ic_imply → "=>",
        ic_or → "\\/",
        ic_and → "/\\"
    end,

print_prefix_connective : prefix_connective → Text
print_prefix_connective(x) ≡ "~",

- - [p. 363]
print_infix_combinator : infix_combinator → Text
print_infix_combinator(x) ≡
    case x of
        icb_ext_choice → "|=|",
        icb_int_choice → "|^|",
        icb_concurrent → "||",
        icb_interlocked → "++",
        icb_sequential → ";"
    end,

print_id_list : id* → Text
print_id_list(x) ≡
    case x of
        ⟨⟩ → "",
        ⟨a⟩ ⌢ ⟨⟩ → print_id(a),
        ⟨a⟩ ⌢ b → print_id(a) ⌢ " " ⌢ print_id_list(b)
    end,

print_id : id → Text
print_id(x) ≡ x,

- - functions for converting int and double
- - they are defined in RSL ++ header files
- - included here in order to enable compilation
- -
RSL_int_to_string : Int → Text,
RSL_double_to_string : Real → Text,

- ————————-Test section————————————
t_id : id = "testint",
t_type_literal : type_literal = tl_Int,
t_te : type_expr =
    type_expr_from_type_literal(t_type_literal),
t_vl : value_literal = int_literal(5),
t_ve : value_expr = ve_val_l(t_vl),
t_init : initialisation = ve_val_l(t_vl),
t_opt_init : opt_initialisation =
    opt_initialisation_from_initialisation(t_ve),
t_svd : single_variable_def =
```

```
            mk_single_variable_def(t_id, t_te, t_opt_init),
        t_vd : variable_def =
            variable_def_from_single_variable_def(t_svd),
        t_vdecl : variable_decl = mk_variable_decl(⟨t_vd⟩),
        t_bdecl : decl = decl_from_variable_decl(t_vdecl),
        t_bce : basic_class_expr =
            mk_basic_class_expr(⟨t_bdecl, t_bdecl⟩),
        t_ce : class_expr =
            class_expr_from_basic_class_expr(t_bce),
        t_ofsp : opt_formal_scheme_parameter = ⟨⟩,
        t_scheme_id : id = ″testscheme″,
        t_scheme_def : scheme_def =
            mk_scheme_def(t_scheme_id, t_ofsp, t_ce),
        t_scheme_decl : scheme_decl =
            mk_scheme_decl(⟨t_scheme_def, t_scheme_def⟩),
        t_decl : decl = decl_from_scheme_decl(t_scheme_decl)

    test_case
        [ test1 ]
            print_id(t_id),
        [ test2 ]
            print_decl(t_decl)
end
```

Appendix C

RSL specifications for the Scheme Diagram

Contents

C.1 Scheme Diagram syntax 179
C.1.1 types.rsl 179
C.1.2 auxiliary.rsl 182
C.1.3 wf_types.rsl 185
C.1.4 wf_scheme.rsl 190
C.1.5 wf_implement.rsl 193
C.1.6 wf_object.rsl 202
C.1.7 wf_extend.rsl 204
C.1.8 wf_association.rsl 207
C.1.9 wf_model.rsl 208
C.2 Translation of Scheme Diagram to RSL. 209
C.3 Imperative Scheme Diagram 225
C.3.1 RSL Part 225
C.3.2 SchemeDiagramInterface.java 235
C.3.3 rsl_esde_libsd_SchemeDiagramInterface.h 237
C.3.4 rsl_esde_libsd_SchemeDiagramInterface.cc 241
C.3.5 convert.cc 244
C.4 Test 245
C.4.1 Applicative 245
C.4.2 Imperative: printed .rsl files 249

C.1 Scheme Diagram syntax

C.1.1 types.rsl

class
--FILE:model_prime.rsl
type
Model′ ::
schemes : Name $\overrightarrow{m}$ ClassExpr ↔ replace_schemes

```
        objects : Name $\overrightarrow{m}$ Object $\leftrightarrow$ replace_objects
        associations : RID $\overrightarrow{m}$ Association $\leftrightarrow$ replace_associations
        extends : RID $\overrightarrow{m}$ Extend $\leftrightarrow$ replace_extends
        implements : RID $\overrightarrow{m}$ Implement $\leftrightarrow$ replace_implements,
    RID = Nat

--FILE:class_expr.rsl
type
    ClassExpr ::
        types : TypeDecls $\leftrightarrow$ replace_types
        values : ValueDecls $\leftrightarrow$ replace_values
        variables : VariableDecls $\leftrightarrow$ replace_variables
        channels : ChannelDecls $\leftrightarrow$ replace_channels
        axioms : AxiomDecls $\leftrightarrow$ replace_axioms

--FILE:type_decls.rsl
type
    TypeDecls = (TypeDecl $\times$ Visibility)$^*$,
    TypeDecl ==
        SortDef(Name) |
        AbbreviationDef(Name, TypeExpr) |
        VariantDef(Name, {| vdl : Variant$^*$ • len vdl $\geq$ 1 |}) |
        UnionDef(Name, {| nwl : NameOrWildcard$^*$ • len nwl $\geq$ 2 |}) |
        ShortRecordDef(Name, ComponentKind$^*$),
    NameOrWildcard == udName(qname : QualifiedName) | udWildcard

--FILE:variant.rsl
type
    Variant ==
        RecordVariant(
            constructor : Variant, components : ComponentKind$^*$) |
        Wildcard |
        Constructor(Name),
    ComponentKind ::
        destructor : OptDestructor
        expr : TypeExpr
        reconstructor : OptReconstructor,
    OptDestructor == deNone | Destructor(Name),
    OptReconstructor == reNone | Reconstructor(Name)

--FILE:type_expr.rsl
type
    TypeExpr ==
        tl_Unit |
        tl_Bool |
        tl_Int |
        tl_Nat |
        tl_Real |
        tl_Text |
        tl_Char |
        TypeName(Name, Qualification) |
        ProductTypeExpr({| tel : TypeExpr$^*$ • len tel $\geq$ 2 |}) |
        BracketedTypeExpr(expr : TypeExpr) |
        FiniteSetTypeExpr(TypeExpr) |
        InfiniteSetTypeExpr(TypeExpr) |
        FiniteListTypeExpr(TypeExpr) |
        InfiniteListTypeExpr(TypeExpr) |
        MapTypeExpr(domain : TypeExpr, range : TypeExpr) |
```

```
        FunctionTypeExpr(
            param : TypeExpr,
            arrow : FunctionArrow,
            result : ResultDescr) |
        SubtypeExpr(TypeExpr, restriction : QualifiedName),
    FunctionArrow == fa_total | fa_partial,
    ResultDescr = AccessDescr* × TypeExpr

--FILE:access_descr.rsl
type
    AccessDescr = AccessMode × Access*,
    AccessMode == am_read | am_write | am_in | am_out,
    Access ==
        NameAccess(QualifiedName) |
        EnumeratedAccess(Access*) |
        CompletedAccess(Qualification)

--FILE:value_decls.rsl
type
    ValueDecls = (ValueDecl × Visibility)*,
    ValueDecl ::
        vdname : Name ↔ replace_vdname
        vdte : TypeExpr ↔ replace_vdte

--FILE:variable_decls.rsl
type
    VariableDecls = (VariableDecl × Visibility)*,
    VariableDecl ::
        vdname : Name ↔ replace_vdname
        vdte : TypeExpr ↔ replace_vdte

--FILE:channel_decls.rsl
type
    ChannelDecls = (ChannelDecl × Visibility)*,
    ChannelDecl ::
        cdname : Name ↔ replace_cdname
        cdte : TypeExpr ↔ replace_cdte

--FILE:axiom_decls.rsl
type
    AxiomDecls = AxiomDecl*,
    AxiomDecl :: adname : Name ↔ replace_adname

--FILE:object.rsl
type
    Object ::
        instance_of : Name
        actual_parameters : ActualParameters
        state : State,
    State = QualifiedName ~m~> Value,
    Value = Text

--FILE:actual_parameters.rsl
type
    ActualParameters = Name ~m~> Name × Fitting,
    Fitting = Name ~m~> Name

--FILE:association.rsl
```

```
    type
        Association ::
            client : Name
            kind : Kind
            supplier : Name
            rolename : Name
            mul : Multiplicity,
        Kind ==
            Nested(Visibility, ActualParameters) |
            Parameter(ActualParameters) |
            Global,
        Multiplicity == None | Index(binding : Name, mtype : TypeExpr)

    - -FILE:extend.rsl
    type
        Extend ::
            client : Name
supplier : Name

    - -FILE:implement.rsl
    type
        Implement ::
            client : Name
            supplier : Name

    type
        Name = Text,
        Qualification = Name*,
        QualifiedName = Name × Qualification,
        Visibility == Private | Public
```

C.1.2 auxiliary.rsl

```
scheme auxiliary =
  extend types with
  class
```

Returns the union of scheme names and object names present in the model.

```
        modules : Model′ → Name-set
        modules(mdl) ≡ dom schemes(mdl) ∪ dom objects(mdl),
```

Returns the union of association, extend and implement id′s present in the model

```
        relations : Model′ → RID-set
        relations(mdl) ≡
            dom associations(mdl) ∪ dom extends(mdl) ∪
            dom implements(mdl)
```

For a given relation in the model the name of the client is returned.

```
        client_of : Model′ × RID ~→ Name
        client_of(mdl, r) ≡
            if r ∈ associations(mdl) then client(associations(mdl)(r))
            else
                if r ∈ extends(mdl) then client(extends(mdl)(r))
                else client(implements(mdl)(r))
```

```
        end
    end
pre r ∈ relations(mdl),
```

For a given relation in the model the name of the supplier is returned.

```
supplier_of : Model′ × RID ~→ Name
supplier_of(mdl, r) ≡
    if r ∈ associations(mdl)
    then supplier(associations(mdl)(r))
    else
        if r ∈ extends(mdl) then supplier(extends(mdl)(r))
        else supplier(implements(mdl)(r))
        end
    end
pre r ∈ relations(mdl),
```

Returns the set of scheme names mapped to the relation for all relations which the specified name is a client. CAUTION: Between client A and supplier B there are two associations relations with the same rolename. Thus the map would have two keys with the name of B making the map non−deterministic.

```
suppliers : Model′ × Name ~→ Name ~m~> RID
suppliers(mdl, s) ≡
    [ supplier_of(mdl, r) ↦ r |
     r : RID • r ∈ relations(mdl) ∧ client_of(mdl, r) = s ]
pre s ∈ modules(mdl),
```

Same as above but only include extend relations.

```
suppliers_ext : Model′ × Name ~→ Name ~m~> RID
suppliers_ext(mdl, s) ≡
    let ss = suppliers(mdl, s) in
        [ sn ↦ ss(sn) |
         sn : Name • sn ∈ ss ∧ ss(sn) ∈ extends(mdl) ]
    end
pre s ∈ modules(mdl),
```

Same as above but only include extend relations.

```
suppliers_ass : Model′ × Name ~→ Name ~m~> RID
suppliers_ass(mdl, s) ≡
    let ss = suppliers(mdl, s) in
        [ sn ↦ ss(sn) |
         sn : Name • sn ∈ ss ∧ ss(sn) ∈ associations(mdl) ]
    end
pre s ∈ modules(mdl)
```

Produces a list of names which is all the type names declared with a given class expression.

```
declared_type_names : ClassExpr → Name*
declared_type_names(s) ≡ ⟨td_name(td) | (td, v) in types(s)⟩,
```

Returns the name of a type declaration

```
td_name : TypeDecl → Name
td_name(td) ≡
    case td of
        SortDef(n) → n,
        VariantDef(n, _) → n,
        UnionDef(n, _) → n,
        ShortRecordDef(n, _) → n,
        AbbreviationDef(n, _) → n
    end,
```

Produces a list of names which are names of the value declarations within the given class expression.

```
declared_value_names : ClassExpr → Name*
declared_value_names(s) ≡ ⟨vdname(vd) | (vd, v) in values(s)⟩,
```

Produces a list of names which are names of the variable declarations within the given class expression.

```
declared_variable_names : ClassExpr → Name*
declared_variable_names(s) ≡
    ⟨vdname(vd) | (vd, v) in variables(s)⟩,
```

Produces a list of names which are names of the channel declarations within the given class expression.

```
declared_channel_names : ClassExpr → Name*
declared_channel_names(s) ≡
    ⟨cdname(cd) | (cd, v) in channels(s)⟩
```

If the given name is a scheme the function will produce a map from rolenames to associations where for all the associations the scheme name is a client. Associations are inherited in the scheme diagram, hence if the scheme is a client in the extend relation then all associations available from the supplier is included. This is repeated recursively. Thus the domain represent all the objects which are reachable from the scheme. If the given name is an object the associations for that object only is returned. This is used when instantiating a global object and technically the objects available in that regard does not include does that become available from the instantiated scheme. A name which does not represent a module in the model will result in an empty map.

```
--FILE:associations.rsl
associations : Model′ × Name → (Name ~m~> Association)
associations(mdl, n) ≡
    let am = associations(mdl) in
        [ case kind(am(rid)) of
            Global → supplier(am(rid)),
            _ → rolename(am(rid))
          end ↦ am(rid) |
          rid : RID •
            rid ∈ dom am ∧ n ∈ modules(mdl) ∧
            n = client(am(rid)) ]
    end †
    let ss = suppliers_ext(mdl, n) in
        if ss = [ ] then [ ] else associations(mdl, hd ss) end
    end,
```

Returns a map of rolanames to associations of all the nested associations emanating from the specified scheme. Parameter and global associations are omitted.

```
associations_nested : Model′ × Name → (Name ~m~> Association)
associations_nested(mdl, n) ≡
    let am = associations(mdl, n) in
        [ n ↦ am(n) | n : Name • n ∈ dom am ∧ is_nested(am(n)) ]
    end,
```

Same as above but includes only associations of kind parameter

```
associations_param : Model′ × Name → (Name ~m~> Association)
associations_param(mdl, n) ≡
    let am = associations(mdl, n) in
        [ n ↦ am(n) |
          n : Name • n ∈ dom am ∧ is_parameter(am(n)) ]
    end,
```

Is the given association of kind Parameter?

```
is_parameter : Association → Bool
is_parameter(a) ≡
    case kind(a) of
        Parameter(_) → true,
        _ → false
    end,
```

Is the given association of kind Nested?

```
is_nested : Association → Bool
is_nested(a) ≡
    case kind(a) of
        Nested(_) → true,
        _ → false
    end,
```

Is the given association of kind Global?

```
is_global : Association → Bool
is_global(a) ≡
    case kind(a) of
        Global → true,
        _ → false
    end
```

Given a name of a module the corresponding scheme name is given. If the module name is the name of a scheme, then the same name is returned. If the module name is the name of an object then the name of the scheme which the object is an instance of is returned.

```
scheme_name : Model′ × Name ~→ Name
scheme_name(mdl, n) ≡
    if n ∈ schemes(mdl) then n
    else instance_of(objects(mdl)(n))
    end
pre n ∈ modules(mdl)
```

C.1.3 wf_types.rsl

The functions presented in this file fall into one of the following four categories. The first it a predicate for checking the well−formedness of type expression; the primary function is *wf_type_expr*. The second is function which determines the maximal type expression for a given type expression; the primary function is *maximal_type*. The third is traversal of the associations of the model which is used by the two previous categories; the two primary functions are *valid_qualification* and *follow_qualification*. The fourth and last is a simple predicate for the *Model'* type which has some simple well−formedness conditions which other predicates can rely on.

wf_extend

```
scheme wf_types =
    extend wf_extend with
    class
```

The well−formedness of a type expressions is basically reduced to the type names, since the rest is captured by the structure of the type expression.

```
wf_type_expr : Model′ × Name × TypeExpr → Bool
wf_type_expr(mdl, m, te) ≡
```

```
    m ∈ modules(mdl) ∧
    case te of
        tl_Unit → true,
        tl_Bool → true,
        tl_Int → true,
        tl_Nat → true,
        tl_Real → true,
        tl_Text → true,
        tl_Char → true,
        TypeName(_, _) → wf_typename_expr(mdl, m, te),
        FiniteSetTypeExpr(te′) →
            wf_type_expr(mdl, m, te′),
        InfiniteSetTypeExpr(te′) →
            wf_type_expr(mdl, m, te′),
        FiniteListTypeExpr(te′) →
            wf_type_expr(mdl, m, te′),
        InfiniteListTypeExpr(te′) →
            wf_type_expr(mdl, m, te′),
        MapTypeExpr(tdom, trng) →
            wf_type_expr(mdl, m, tdom) ∧
            wf_type_expr(mdl, m, trng),
        FunctionTypeExpr(p, _, (a, r)) →
            wf_type_expr(mdl, m, p) ∧
            wf_access_descr(mdl, m, a) ∧
            wf_type_expr(mdl, m, r),
        SubtypeExpr(te′, restriction) →
            wf_subtype_expr(mdl, m, te),
        BracketedTypeExpr(te′) →
            wf_type_expr(mdl, m, te′),
        ProductTypeExpr(tel) →
            wf_product_type_expr(mdl, m, tel)
    end,
```

The type name is a reference to a declared type which thus must exist relatively to the context in which the type name is used.

```
    --FILE:wf_typename_expr.rsl
    wf_typename_expr : Model′ × Name × TypeExpr → Bool
    wf_typename_expr(mdl, m, tn) ≡
        case tn of
            TypeName(n, q) →
                valid_qualification(mdl, m, q) ∧
                let schn = follow_qualification(mdl, m, q) in
                    schn ∈ schemes(mdl) ∧
                    n ∈ declared_type_names(extend_history(mdl,schn))
                end,
            _ → false
        end,
```

A subtype expression is well−formed if the supertype expression is well−formed and the restriction predicate exists. If the qualification for the restriction is empty then private predicates are also allowed. TODO: the type expression of the functions must be a function with return type boolean and the parameter must be the same maximal type as the supertype.

```
    wf_subtype_expr : Model′ × Name × TypeExpr → Bool
    wf_subtype_expr(mdl, m, ste) ≡
        case ste of
            SubtypeExpr(ste, (rn, rq)) →
                wf_type_expr(mdl, m, expr(ste)) ∧
                valid_qualification(mdl, m, rq) ∧
```

```
            let
                n = follow_qualification(mdl, m, rq),
                s =
                    if n ∈ schemes(mdl) then schemes(mdl)(n)
                    else
                        schemes(mdl)(instance_of(objects(mdl)(n)))
                    end
            in
                (∃ (vd, visi) : ValueDecl × Visibility •
                    (vd, visi) ∈ values(s) ∧
                    vdname(vd) = rn ∧
                    (visi = Public ∨ rq = ⟨⟩))
            end,
        _ → false
    end,
```

Recurses through a list and checks if every type expression in the list is well−formed. An all quantification would have been better, but recursion in quantification is not allowed in the RSL to C♯ translator.

```
wf_product_type_expr :
    Model′ × Name × TypeExpr* → Bool
wf_product_type_expr(mdl, m, tel) ≡
    tel = ⟨⟩ ∨
    (tel ≠ ⟨⟩ ∧ wf_type_expr(mdl, m, hd tel) ∧
     wf_product_type_expr(mdl, m, tl tel)),
```

Checks an access description list for well−formedness. This is basically a check for the presence of a variable or channel for read or write and in or out respectively.

```
wf_access_descr :
    Model′ × Name × AccessDescr* → Bool
wf_access_descr(mdl, m, adl) ≡
    (∀ (am, al) : AccessMode × Access* •
        (am, al) ∈ adl ⇒
            (∀ a : Access •
                a ∈ al ⇒ wf_access(mdl, m, am, a))),

wf_access :
    Model′ × Name × AccessMode × Access → Bool
wf_access(mdl, m, am, a) ≡
    case a of
        NameAccess((n, q)) →
            valid_qualification(mdl, m, q) ∧
            let
                qend = follow_qualification(mdl, m, q),
                s =
                    if qend ∈ schemes(mdl)
                    then schemes(mdl)(qend)
                    else
                        schemes(mdl)(
                            instance_of(objects(mdl)(qend)))
                    end
            in
                ((am = am_read ∨ am = am_write) ∧
                 (∃
                        (vd, visi) : VariableDecl × Visibility
                    •
                        (vd, visi) ∈ variables(s) ∧
                        vdname(vd) = n ∧ visi = Public)) ∨
                ((am = am_in ∨ am = am_out) ∧
```

```
            (∃
                (cd, visi) : ChannelDecl × Visibility
              •
                (cd, visi) ∈ channels(s) ∧
                cdname(cd) = n ∧ visi = Public))
        end,
    EnumeratedAccess(al′) →
        al′ = ⟨⟩ ∨
        (al′ ≠ ⟨⟩ ∧ wf_access(mdl, m, am, hd al′) ∧
         wf_access(mdl, m, am, EnumeratedAccess(tl al′))),
    CompletedAccess(q) →
        valid_qualification(mdl, m, q)
end
```

Determines the maximal type for a given type expression. Since the structure of a type expression is recursive, the function is also recursive. It will terminate when a native type literal is encountered or a sort. The parameter 'q′ is the qualification used so far to reach the scheme 's′. The only reason for giving the entire model, qualification and scheme as parameter is because of type names. This is a kind of reference to other types also allowing qualification, thus it is possible to reference types outside the current scheme.

```
maximal_type :
    Model′ × Qualification × Name × TypeExpr ~→
        TypeExpr
maximal_type(mdl, q, s, te) ≡
    case te of
        tl_Unit → tl_Unit,
        tl_Bool → tl_Bool,
        tl_Int → tl_Int,
        tl_Nat → tl_Int,
        tl_Real → tl_Real,
        tl_Text → InfiniteListTypeExpr(tl_Char),
        tl_Char → tl_Char,
        FiniteSetTypeExpr(te′) →
            InfiniteSetTypeExpr(maximal_type(mdl, q, s, te′)),
        InfiniteSetTypeExpr(te′) →
            InfiniteSetTypeExpr(maximal_type(mdl, q, s, te′)),
        FiniteListTypeExpr(te′) →
            InfiniteListTypeExpr(
                maximal_type(mdl, q, s, te′)),
        InfiniteListTypeExpr(te′) →
            InfiniteListTypeExpr(
                maximal_type(mdl, q, s, te′)),
        MapTypeExpr(tdom, trng) →
            MapTypeExpr(
                maximal_type(mdl, q, s, tdom),
                maximal_type(mdl, q, s, trng)),
        FunctionTypeExpr(p, a, (adl, adl_te)) →
            FunctionTypeExpr(
                maximal_type(mdl, q, s, p), fa_partial,
                (adl, maximal_type(mdl, q, s, adl_te))),
        BracketedTypeExpr(te′) →
            BracketedTypeExpr(maximal_type(mdl, q, s, te′)),
        SubtypeExpr(te′, _) →
            maximal_type(mdl, q, s, te′),
        TypeName(n′, q′) →
            maximal_type_name(mdl, q, s, (n′, q′)),
        ProductTypeExpr(tel) →
            ProductTypeExpr(
```

```
                maximal_product_type_expr(mdl, q, s, tel))
    end,
```

Determines the maximal type of a type name expression. A type name expression is the name of a type declaration thus it is maximal type of the associated type expression that is the result. There exist two possibilities. First the type declaration can be an abbreviation which means the result is the maximal type of the abbreviation expression. Second the type declaration is a sort thus the maximal type is the name of the sort prepended with the qualification used to reach the sort.

```
maximal_type_name :
    Model′ × Qualification × Name ×
    (Name × Qualification) ~→
        TypeExpr
maximal_type_name(mdl, q, s, (n′, q′)) ≡
    let
        next_s = follow_qualification(mdl, s, q′),
        td =
            find_type_decl(
                q, types(schemes(mdl)(next_s)), n′)
    in
        case td of
            AbbreviationDef(_, te′) →
                maximal_type(mdl, q ⌢ q′, next_s, te′),
            _ → TypeName(n′, q ⌢ q′)
        end
    end
pre valid_qualification(mdl, s, q′),
```

Finds the type declaration with the given name from a list of type declarations. It is assumed that the there exists a type declaration with the given name in the list. If no qualification is used at all then it is also allowed to see private type declarations.

```
find_type_decl :
    Qualification × TypeDecls × Name ~→ TypeDecl
find_type_decl(q, tdl, n) ≡
    let (td, visi) = hd tdl in
        if td_name(td) = n ∧ (visi = Public ∨ q = ⟨⟩)
        then td
        else find_type_decl(q, tl tdl, n)
        end
    end
pre tdl ≠ ⟨⟩,
```

Determines the maximal type expression for a product by recursing through the list and determine the maximal type expression for each element. The list of maximal type expressions is returned. 'q′ is the qualification used so far to reach the scheme 's′. 'tel′ is the list of type expressions which the product type expression consists of. It is obvious to used a comprehended list expression but it is not translatable into C‖.

```
maximal_product_type_expr :
    Model′ × Qualification × Name × TypeExpr* →
        TypeExpr*
maximal_product_type_expr(mdl, q, s, tel) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
        ⟨maximal_type(mdl, q, s, hd tel)⟩ ⌢
        maximal_product_type_expr(mdl, q, s, tl tel)
    end,
```

Determines whether a given qualification is valid from a specific scheme within the model. This is done by traversing the associations in the model recursively. 'n′ denotes the module currently visited and 'q′ the remainder of the qualification to reach the destination module. if 'q′ is empty and 'n′ is a module then it was a

valid qualification. if 'q′ is not empty then the next module to be reached from 'n′ must be available from 'n′. If 'n′ is an object then the association map 'am′ is taken from the scheme which 'n′ is an instance of.

```
valid_qualification :
    Model′ × Name × Qualification ~→ Bool
valid_qualification(mdl, n, q) ≡
    n ∈ modules(mdl) ∧
    let am = associations(mdl, scheme_name(mdl, n)) in
        q = ⟨⟩ ∨
        (hd q ∈ dom am ∧
         valid_qualification(mdl, supplier(am(hd q)), tl q)
        )
    end,
```

Given a valid qualification the name of the scheme at the end is returned. See the function *valid_qualification*.

```
follow_qualification :
    Model′ × Name × Qualification ~→ Name
follow_qualification(mdl, n, q) ≡
    let
        sn = scheme_name(mdl, n),
        am = associations(mdl, sn)
    in
        if q = ⟨⟩ then sn
        else
            follow_qualification(
                mdl, supplier(am(hd q)), tl q)
        end
    end
pre valid_qualification(mdl, n, q)
```

C.1.4 wf_scheme.rsl

The *wf_scheme.rsl* file contains well−formed predicates for the scheme aspect of the diagram. There are essentially two predicates which checks a scheme. The first, *wf_class_expr*, which checks the names used for the declarations in the scheme, and the second, *wf_scheme_decl_expr*, checks the type expressions of the declarations.

wf_implement

```
scheme wf_scheme =
    extend wf_implement with
    class

        wf_schemes : Model′ → Bool
        wf_schemes(mdl) ≡
            (∀ n : Name •
                n ∈ schemes(mdl) ⇒
                    wf_scheme_decl_expr(mdl, n) ∧
                    wf_class_expr(mdl, n, extend_history(mdl, n))),
```

Er denne overhovedet nødvendig eller skal den udvides i forbindelse med extend? Nope, This function is used on an expanded schemel by wf_extend.

- Names within type declaratinos must be unique.

- Names within value declarations must be unique unless the maximal signature is diffrent (overloading).
- Value and variable names may not overlap.
- Names within channel declarations must be unique.

```
wf_class_expr : Model′ × Name × ClassExpr → Bool
wf_class_expr(mdl, n, s) ≡
    let
        typdl = declared_type_names(s),
        valdl = ⟨vd | (vd, visi) in values(s)⟩,
        valdl′ = ⟨vdname(vd) | vd in valdl⟩,
        vardl = declared_variable_names(s),
        chadl = declared_channel_names(s)
    in
        len typdl = card elems typdl ∧
        wf_value_overloading(mdl, n, valdl) ∧
        len vardl = card elems vardl ∧
        elems vardl ∩ elems valdl′ = {} ∧
        len chadl = card elems chadl
    end
pre n ∈ schemes(mdl),
```

wf_value_overloading: Checks a list of value declarations for valid names. Names within value declarations must be unique unless the maximal signature is diffrent (overloading). It is necessary to use indices since using the name would at some time yield the same entry, meaning that the to selected declarations would be identical and thus fail.

```
--FILE:wf_value_overloading.rsl
wf_value_overloading :
    Model′ × Name × ValueDecl* → Bool
wf_value_overloading(mdl, n, valdl) ≡
    (∀ i : Nat •
        i ∈ inds valdl ⇒
            (∀ j : Nat •
                j ∈ inds valdl ∧ i ≠ j ⇒
                    vdname(valdl(i)) ≠ vdname(valdl(j)) ∨
                    maximal_type(mdl, ⟨⟩, n, vdte(valdl(i))) ≠
                        maximal_type(
                            mdl, ⟨⟩, n, vdte(valdl(j)))))
```

In the Scheme Diagram only type expressions are included, thus also the only expression kind that can be checked. This must be done for type declarations, value declarations, variable declarations, and channel declarations, since the signature for these declarations are included. Axioms and object declarations are omitted.

```
wf_scheme_decl_expr : Model′ × Name → Bool
wf_scheme_decl_expr(mdl, n) ≡
    n ∈ schemes(mdl) ∧
    let s = schemes(mdl)(n) in
        wf_type_decls(mdl, n, types(s)) ∧
        wf_value_decls(mdl, n, values(s)) ∧
        wf_variable_decls(mdl, n, variables(s)) ∧
        wf_channel_decls(mdl, n, channels(s))
    end,
```

The type expressions used in the list of type declaraitons are all checked for well−formedness.

```
wf_type_decls : Model′ × Name × TypeDecls → Bool
wf_type_decls(mdl, n, tdl) ≡
    (∀ (td, visi) : TypeDecl × Visibility •
        (td, visi) ∈ tdl ⇒ wf_type_decl(mdl, n, td)),
```

Determines the well−formedness of type expressions used in a type declaration.

```
wf_type_decl : Model′ × Name × TypeDecl → Bool
wf_type_decl(mdl, n, td) ≡
    case td of
        SortDef(_) → true,
        AbbreviationDef(_, e) → wf_type_expr(mdl, n, e),
        VariantDef(_, vdl) →
            (∀ v : Variant •
                v ∈ vdl ⇒ wf_variant(mdl, n, v)),
        ShortRecordDef(_, ckl) →
            (∀ ck : ComponentKind •
                ck ∈ ckl ⇒
                    wf_type_expr(mdl, n, expr(ck)) ∧
                    (destructor(ck) = deNone ⇒
                            reconstructor(ck) = reNone)),
        UnionDef(_, nwl) → wf_union_def(mdl, n, nwl)
    end,

--FILE:wf_union_def.rsl
wf_union_def :
    Model′ × Name × NameOrWildcard* → Bool
wf_union_def(mdl, n, nwl) ≡
    (∀ i : Int •
        i ∈ inds nwl ⇒
            case nwl(i) of
                udName((n′, q′)) →
                    valid_qualification(mdl, n, q′) ∧
                    n′ ∈
                        declared_type_names(
                                schemes(mdl)(
                                    follow_qualification(mdl, n, q′))),
                udWildcard → true
            end),
```

Tests the well−formedness of the Variant type consisting of the following three requirements:

- If the Variant is a RecordVariant the Variant representing the constructor must actually be a Constructor.
- If a reconstructor is present the a destructor must also be present.
- All type expressions used in the component kinds must be well−formed.

```
wf_variant : Model′ × Name × Variant → Bool
wf_variant(mdl, n, v) ≡
    case v of
        RecordVariant(con, ckl) →
            case con of
                Constructor(_) → true,
                _ → false
            end ∧
            (∀ ck : ComponentKind •
                ck ∈ ckl ⇒
                    wf_type_expr(mdl, n, expr(ck)) ∧
                    (destructor(ck) = deNone ⇒
                            reconstructor(ck) = reNone)),
        _ → true
    end,
```

The following three functions check the type expressions used in the signatures of respectively value declarations, variable declarations, and channel declarations.

```
wf_value_decls : Model′ × Name × ValueDecls → Bool
wf_value_decls(mdl, n, vdl) ≡
    (∀ (vd, visi) : ValueDecl × Visibility •
        (vd, visi) ∈ vdl ⇒
            wf_type_expr(mdl, n, vdte(vd))),

wf_variable_decls :
    Model′ × Name × VariableDecls → Bool
wf_variable_decls(mdl, n, vdl) ≡
    (∀ (vd, visi) : VariableDecl × Visibility •
        (vd, visi) ∈ vdl ⇒
            wf_type_expr(mdl, n, vdte(vd))),

wf_channel_decls :
    Model′ × Name × ChannelDecls → Bool
wf_channel_decls(mdl, n, cdl) ≡
    (∀ (cd, visi) : ChannelDecl × Visibility •
        (cd, visi) ∈ cdl ⇒
            wf_type_expr(mdl, n, cdte(cd)))
```

C.1.5 wf_implement.rsl

wf_types

scheme wf_implement =
extend wf_types **with**
class

Checks if every implement relation in the model is well−formed.

```
wf_implements : Model′ → Bool
wf_implements(mdl) ≡
    (∀ rid : RID •
        rid ∈ dom implements(mdl) ⇒
            wf_implement(mdl, rid)),
```

Determines if an implement relation is well−formed based on its relation id.

- The client and supplier must both be schemes.
- The client must statically implement the supplier.

```
--FILE:wf_implement.rsl
wf_implement : Model′ × RID → Bool
wf_implement(mdl, rid) ≡
    rid ∈ implements(mdl) ∧
    let i = implements(mdl)(rid) in
        {client(i), supplier(i)} ⊆ dom schemes(mdl) ∧
        static_implement(mdl, client(i), supplier(i))
    end


static_implement : Model′ × Name × Name ~→ Bool
static_implement(mdl, cn, sn) ≡
    static_implement(mdl, (cn, [ ]), sn)
pre {cn, sn} ⊆ dom schemes(mdl),
```

```
static_implement :
    Model′ × (Name × Fitting) × Name ~→ Bool
static_implement(mdl, (cn, fit), sn) ≡
    static_implement_param(mdl, cn, sn) ∧
    let
        (csig_ce′, csig_obj′) = signature(mdl, cn),
        csig_ce = rename(fit, csig_ce′),
        csig_obj = rename(fit, csig_obj′),
        (ssig_ce, ssig_obj) = signature(mdl, sn)
    in
        static_implement_body(
            mdl, (cn, csig_ce), (sn, ssig_ce)) ∧
        static_implement_obj(
            mdl, (cn, csig_obj), (sn, ssig_obj))
    end
pre cn ∈ modules(mdl) ∧ sn ∈ schemes(mdl)

type SchN_ObjM = Name × (Name -m→ Association)
```

All parameters are visible. Should the formal parameters match? What is the relationship? Notice that the relation is reversed for parameters!

```
static_implement_param :
    Model′ × Name × Name ~→ Bool
static_implement_param(mdl, cn, sn) ≡
    let
        cfpm = associations_param(mdl, cn),
        sfpm = associations_param(mdl, sn)
    in
        dom sfpm = dom cfpm ∧
        static_implement_assoc(
            mdl, (sn, sfpm), (cn, cfpm), dom sfpm)
    end
pre {cn, sn} ⊆ dom schemes(mdl),

static_implement_obj :
    Model′ × SchN_ObjM × SchN_ObjM → Bool
static_implement_obj(mdl, (cn, cobj), (s, sobj)) ≡
    dom sobj ⊆ dom cobj ∧
    static_implement_assoc(
        mdl, (cn, cobj), (s, sobj), dom sobj),

static_implement_assoc :
    Model′ × SchN_ObjM × SchN_ObjM × Name-set → Bool
static_implement_assoc(mdl, (cn, cobj), (s, sobj), ons) ≡
    ons = {} ∨
    let on = hd ons in
        static_implement(
            mdl, supplier(cobj(on)), supplier(sobj(on))) ∧
        static_implement_assoc(
            mdl, (cn, cobj), (s, sobj), ons \ {on})
    end,
```

Determines whether mc implements ms Notice that this function does not determine the maximal type. Perhaps it should; yes it should. It does not work; yet. Fx. for value declarations must be identical, however this is not necessarily the case the the supplier declares a sort and the client refines this to an abbreviation declaration. As example look at S1 implements S in examples.rsl.

```
static_implement_body :
```

```
    Model′ × (Name × ClassExpr) × (Name × ClassExpr) →
      Bool
static_implement_body(mdl, (cn, c), (sn, s)) ≡
  let
    mc = maximal_class(mdl, cn, c),
    ms = maximal_class(mdl, sn, s)
  in
    static_implement_types(mc, ms) ∧
    static_implement_values(mc, ms) ∧
    static_implement_variables(mc, ms) ∧
    static_implement_channels(mc, ms)
  end,

static_implement_types :
  ClassExpr × ClassExpr → Bool
static_implement_types(c, s) ≡
  (∀ (td, visi) : TypeDecl × Visibility •
    (td, visi) ∈ types(s) ∧ visi = Public ⇒
      case td of
        SortDef(n) →
          n ∈
            ⟨td_name(td′) |
              (td′, visi′) in types(c) •
                visi′ = Public⟩,
        AbbreviationDef(_, _) →
          (td, Public) ∈ types(c)
      end),
```

This lookup mayby have to be done in types, and variables and channels.

```
static_implement_values :
  ClassExpr × ClassExpr → Bool
static_implement_values(c, s) ≡
  let σ = gen_sig_map(types(c)) in
    (∀ (vd, visi) : ValueDecl × Visibility •
      (vd, visi) ∈ values(s) ∧ visi = Public ⇒
        (replace_vdte(sig_morph(σ, vdte(vd)), vd),
          Public) ∈ values(c))
  end,

static_implement_variables :
  ClassExpr × ClassExpr → Bool
static_implement_variables(c, s) ≡
  let σ = gen_sig_map(types(c)) in
    (∀ (vd, visi) : VariableDecl × Visibility •
      (vd, visi) ∈ variables(s) ∧ visi = Public ⇒
        (replace_vdte(sig_morph(σ, vdte(vd)), vd),
          Public) ∈ variables(c))
  end,

static_implement_channels :
  ClassExpr × ClassExpr → Bool
static_implement_channels(c, s) ≡
  (∀ (cd, visi) : ChannelDecl × Visibility •
    (cd, visi) ∈ channels(s) ∧ visi = Public ⇒
      (cd, Public) ∈ channels(c))
```

TODO: Include qualification in the domain

```
gen_sig_map : TypeDecls → (Text ~m~> TypeExpr)
gen_sig_map(tdl) ≡
  if tdl = ⟨⟩ then [ ]
  else
    case hd tdl of
      (td, Public) →
        case td of
          SortDef(n) → [ n ↦ TypeName(n, ⟨⟩) ],
          AbbreviationDef(n, te) → [ n ↦ te ]
        end,
      (_, Private) → [ ]
    end † gen_sig_map(tl tdl)
  end,
```

If the type expression is a type name representing a sort with qualification then the qualification should be preserved!? TODO:

```
sig_morph :
  (Text ~m~> TypeExpr) × TypeExpr → TypeExpr
sig_morph(σ, te) ≡
  case te of
    TypeName(n, q) → lookup(σ, n, te),
    FiniteSetTypeExpr(te′) →
      FiniteSetTypeExpr(sig_morph(σ, te′)),
    InfiniteSetTypeExpr(te′) →
      InfiniteSetTypeExpr(sig_morph(σ, te′)),
    FiniteListTypeExpr(te′) →
      FiniteListTypeExpr(sig_morph(σ, te′)),
    InfiniteListTypeExpr(te′) →
      InfiniteListTypeExpr(sig_morph(σ, te′)),
    MapTypeExpr(tdom, trng) →
      MapTypeExpr(
        sig_morph(σ, tdom),
        sig_morph(σ, tdom)),
    FunctionTypeExpr(p, arrow, (a, r)) →
      FunctionTypeExpr(
        sig_morph(σ, p), arrow,
        (a, sig_morph(σ, r))),
    SubtypeExpr(te′, restriction) →
      SubtypeExpr(sig_morph(σ, te′), restriction),
    BracketedTypeExpr(te′) →
      BracketedTypeExpr(sig_morph(σ, te′)),
    ProductTypeExpr(tel) →
      ProductTypeExpr(sig_morph_list(σ, tel)),
    _ → te
  end,
```

Recursion is not allowed in compreheded list expression, thus this functions must be specified seperately.

```
sig_morph_list :
  (Text ~m~> TypeExpr) × TypeExpr* →
    TypeExpr*
sig_morph_list(σ, tel) ≡
  if tel = ⟨⟩ then ⟨⟩
  else
    ⟨sig_morph(σ, hd tel)⟩ ⌢
    sig_morph_list(σ, tl tel)
  end,

lookup :
```

```
    (Text →m TypeExpr) × Text × TypeExpr → TypeExpr
  lookup(σ, n, te) ≡
    if n ∈ σ then σ(n) else te end

value
  signature :
    Model′ × Name ~→
      ClassExpr × (Name →m Association)
  signature(mdl, scheme_name) ≡
    (maximal_class(
        mdl, scheme_name,
        expand_class(extend_history(mdl, scheme_name))),
     associations_nested(mdl, scheme_name)),
```

The maximal class expression for the scheme 's′ is determined. The 'mdl′ and 'n′ is used only to define the context in which the maximal class of 's′ is to be found within. However it is recommended that schemes(mdl)(n) = s, but is this too strong? It will probably not be the case after renaming, hence it will not be added as a precondition.

```
  maximal_class :
    Model′ × Name × ClassExpr ~→ ClassExpr
  maximal_class(mdl, n, s) ≡
    let
      types′ =
        ⟨(max_type_decl(mdl, n, t), visi) |
          (t, visi) in types(s)⟩,
      values′ =
        ⟨(max_value_decl(mdl, n, v), visi) |
          (v, visi) in values(s)⟩,
      variables′ =
        ⟨(max_variable_decl(mdl, n, v), visi) |
          (v, visi) in variables(s)⟩,
      channels′ =
        ⟨(max_channel_decl(mdl, n, c), visi) |
          (c, visi) in channels(s)⟩
    in
      mk_ClassExpr(
        types′, values′, variables′, channels′,
        axioms(s))
    end
  pre n ∈ schemes(mdl),

  max_type_decl :
    Model′ × Name × TypeDecl ~→ TypeDecl
  max_type_decl(mdl, n, td) ≡
    case td of
      SortDef(_) → td,
      AbbreviationDef(n, te) →
        AbbreviationDef(
          n, maximal_type(mdl, ⟨⟩, n, te))
    end
  pre
    case td of
      SortDef(_) → true,
      AbbreviationDef(_) → true,
      _ → false
    end,

  max_value_decl :
```

```
        Model′ × Name × ValueDecl ~→ ValueDecl
    max_value_decl(mdl, n, vd) ≡
        replace_vdte(
            maximal_type(mdl, ⟨⟩, n, vdte(vd)), vd),

    max_variable_decl :
        Model′ × Name × VariableDecl ~→ VariableDecl
    max_variable_decl(mdl, n, vd) ≡
        replace_vdte(
            maximal_type(mdl, ⟨⟩, n, vdte(vd)), vd),

    max_channel_decl :
        Model′ × Name × ChannelDecl ~→ ChannelDecl
    max_channel_decl(mdl, n, cd) ≡
        replace_cdte(
            maximal_type(mdl, ⟨⟩, n, cdte(cd)), cd)

value
    expand_class : ClassExpr → ClassExpr
    expand_class(s) ≡
        let
            types′ =
                ⟨(expand_type_decl(t), visi) |
                    (t, visi) in types(s)⟩,
            values′ =
                values(s) ⌢ value_from_type_decl(types(s))
        in
            replace_values(values′, replace_types(types′, s))
        end,

    expand_type_decl : TypeDecl → TypeDecl
    expand_type_decl(td) ≡
        case td of
            VariantDef(name, _) → SortDef(name),
            UnionDef(name, _) → SortDef(name),
            ShortRecordDef(name, _) → SortDef(name),
            _ → td
        end

value
    value_from_type_decl : TypeDecls → ValueDecls
    value_from_type_decl(tdl) ≡
        if tdl = ⟨⟩ then ⟨⟩
        else
            let
                (td, visi) = hd tdl,
                vdl =
                    case td of
                        VariantDef(_, expr) →
                            value_from_variant(td, expr),
                        UnionDef(_) → value_from_union(td),
                        ShortRecordDef(_) →
                            value_from_short_record(td),
                        _ → ⟨⟩
                    end
            in
                ⟨(v, visi) | v in vdl⟩ ⌢
                value_from_type_decl(tl tdl)
```

```
        end
    end,

value_from_variant :
    TypeDecl × Variant* ~→ ValueDecl*
value_from_variant(td, vl) ≡
    case td of
        VariantDef(name, expr) →
            let sort = TypeName(name, ⟨⟩) in
                if vl = ⟨⟩ then ⟨⟩
                else
                    value_from_variant_aux(sort, hd vl) ⌢
                    value_from_variant(td, tl vl)
                end
            end
    end
pre
    case td of
        VariantDef(_) → true,
        _ → false
    end,

value_from_variant_aux :
    TypeExpr × Variant → ValueDecl*
value_from_variant_aux(sort, v) ≡
    case v of
        Wildcard → ⟨⟩,
        Constructor(n) → ⟨mk_ValueDecl(n, sort)⟩,
        RecordVariant(c, ckl) →
            case c of
                Wildcard → ⟨⟩,
                Constructor(n) →
                    ckl2vdl(sort, ckl) ⌢
                    ⟨mk_variant_fun(
                            n,
                            ProductTypeExpr(
                                ⟨expr(ck) | ck in ckl⟩), sort)⟩
            end
    end,

value_from_union : TypeDecl ~→ ValueDecl*
value_from_union(td) ≡
    case td of
        UnionDef(udn, nwl) →
            let
                sort = TypeName(udn, ⟨⟩),
                type_names =
                    ⟨let (n, q) = qname(now) in n end |
                        now in nwl • now ≠ udWildcard⟩
            in
                ⟨mk_variant_fun(
                        udn ⌢ "_from_" ⌢ tn, sort,
                        TypeName(tn, ⟨⟩)) | tn in type_names⟩ ⌢
                ⟨mk_variant_fun(
                        tn ⌢ "_to_" ⌢ udn, TypeName(tn, ⟨⟩),
                        sort) | tn in type_names⟩
            end
    end
```

```
pre
    case td of
        UnionDef(_) → true,
        _ → false
    end,

value_from_short_record : TypeDecl → ValueDecl*
value_from_short_record(td) ≡
    case td of
        ShortRecordDef(srdn, ckl) →
            let sort = TypeName(srdn, ⟨⟩) in
                ⟨mk_variant_fun(
                        "mk_" ^ srdn,
                        ProductTypeExpr(⟨expr(ck) | ck in ckl⟩),
                        sort)⟩ ^ ckl2vdl(sort, ckl)
            end
    end
pre
    case td of
        ShortRecordDef(_) → true,
        _ → false
    end,

-- Before te was called sort and was of type
-- TypeName
--
--
ckl2vdl :
    TypeExpr × ComponentKind* → ValueDecl*
ckl2vdl(te, ckl) ≡
    if ckl = ⟨⟩ then ⟨⟩
    else ck2vdl(te, hd ckl) ^ ckl2vdl(te, tl ckl)
    end,

-- Before te was called sort and was of type
-- TypeName
--
--
ck2vdl : TypeExpr × ComponentKind → ValueDecl*
ck2vdl(te, ck) ≡
    case destructor(ck) of
        Destructor(dname) →
            ⟨mk_variant_fun(dname, te, expr(ck))⟩,
        deNone → ⟨⟩
    end ^
    case reconstructor(ck) of
        Reconstructor(rname) →
            ⟨mk_variant_fun(
                    rname, ProductTypeExpr(⟨expr(ck), te⟩), te
                )⟩,
        reNone → ⟨⟩
    end
pre
    case te of
        TypeName(_) → true,
        _ → false
    end,
```

```
mk_variant_fun :
    Name × TypeExpr × TypeExpr → ValueDecl
mk_variant_fun(fn, param, return) ≡
    mk_ValueDecl(
        fn,
        FunctionTypeExpr(
                param, fa_partial, (⟨⟩, return))),

qname2typename : QualifiedName → TypeExpr
qname2typename(n, q) ≡ TypeName(n, q)
```

returns the scheme from the model for a given name. If the name is an object the scheme which the object is an instance of is returned

```
name2scheme : Model′ × Name ~→ ClassExpr
name2scheme(mdl, n) ≡ schemes(mdl)(n)
pre n ∈ schemes(mdl)
```

Returns the class expression after it has been renamed. See the RSL book p. 213. fit: fitting information s: scheme that has to be fitted

```
rename : Fitting × ClassExpr → ClassExpr
rename(fit, s) ≡
    mk_ClassExpr(
        rename_types(fit, types(s)),
        ⟨(replace_vdname(rename(fit, vdname(vd)), vd), v) |
            (vd, v) in values(s)⟩,
        ⟨(replace_vdname(rename(fit, vdname(vd)), vd), v) |
            (vd, v) in variables(s)⟩,
        ⟨(replace_cdname(rename(fit, cdname(cd)), cd), v) |
            (cd, v) in channels(s)⟩,
        ⟨replace_adname(rename(fit, adname(ad)), ad) |
            ad in axioms(s)⟩),
```

It is not good if an old name is renamed to a new which is already part of the old. Thus it is a precondition.

```
rename :
    Fitting × (Name ~m~> Association) ~→
        (Name ~m~> Association)
rename(fit, am) ≡
    [ rename(fit, n) ↦ am(n) | n : Name • n ∈ dom am ]
pre fit = [ ] ∨ ∼ (rng fit ⊆ dom am),

rename : Fitting × Name → Name
rename(fit, n) ≡ if n ∈ fit then fit(n) else n end,

rename_types : Fitting × TypeDecls → TypeDecls
rename_types(fit, tds) ≡
    ⟨((case td of
            SortDef(name) → SortDef(rename(fit, name)),
            VariantDef(name, aux) →
                VariantDef(rename(fit, name), aux),
            UnionDef(name, aux) →
                UnionDef(rename(fit, name), aux),
            ShortRecordDef(name, aux) →
                ShortRecordDef(rename(fit, name), aux),
            AbbreviationDef(name, aux) →
                AbbreviationDef(rename(fit, name), aux)
        end), v) | (td, v) in tds⟩
```

C.1.6 wf_object.rsl

This file is about objects as the name might suggest. There are three categories of functions. The first is well−formedness of objects in the model combines the following two categories. The second is about the object state: determining the variables that participate in the state and well−formedness. The third and last is well−formedness of scheme instantiation. It is used in conjunction with the *wf_object* function but is also used in *wf_association*.

wf_scheme

```
scheme wf_object =
  extend wf_scheme with
  class
```

Check well−formedness for all objects present in the model.

```
wf_objects : Model′ → Bool
wf_objects(mdl) ≡
  (∀ on : Name •
    on ∈ dom objects(mdl) ⇒ wf_object(mdl, on)),
```

An object is well formed if it is an instance of a scheme present in the model and the scheme instantiation is well−formed. Additionally the state of an object must represent the variables part of the scheme which it is an instance of or reachable via nested associations.

```
wf_object : Model′ × Name → Bool
wf_object(mdl, on) ≡
  on ∈ dom objects(mdl) ∧
  let o = objects(mdl)(on) in
    instance_of(o) ∈ dom schemes(mdl) ∧
    wf_object_state(mdl, on) ∧
    wf_scheme_instantiation(
      mdl, associations(mdl, on) \ {on},
      instance_of(o), actual_parameters(o))
  end
```

The state of an object is well−formed if there for any variable available from the object is a corresponding key in the state map. Nothing is said about the value of a given variable. It is represented by a text string and the user is free to enter anything. If the value should be verified then it would be necessary to include value expressions which is far to cumbersome.

```
wf_object_state : Model′ × Name → Bool
wf_object_state(mdl, on) ≡
  on ∈ dom objects(mdl) ∧
  dom state(objects(mdl)(on)) = state_domain(mdl, on),
```

Determines the set of variables that constitute the state of an object, based on the scheme which the object is an instance of.

```
--FILE:state_domain.rsl
state_domain : Model′ × Name ~→ QualifiedName-set
state_domain(mdl, n) ≡
  let sn = scheme_name(mdl, n) in
    state_domain_scheme(mdl, ⟨⟩, sn) ∪
    {(vdname(vd), ⟨⟩) |
     (vd, visi) : VariableDecl × Visibility •
       (vd, visi) ∈
         elems variables(schemes(mdl)(sn))}
```

```
    end
pre n ∈ modules(mdl),
```

Determines the variables in the scheme sn and recursively through its nested associations. The qualification used to get to the scheme 'sn′ is prepended. Notice that *extend_history* only includes public declarations from suppliers of the extend relationship. This function is mutual recursive with the function *state_domain_ass*.

```
state_domain_scheme :
    Model′ × Qualification × Name → QualifiedName-set
state_domain_scheme(mdl, q, sn) ≡
    let vdl = variables(extend_history(mdl, sn)) in
        {(vdname(vd), q) |
         (vd, visi) : VariableDecl × Visibility •
             (vd, visi) ∈ elems vdl}
    end ∪
    state_domain_ass(
         mdl, q, associations_nested(mdl, sn))
pre sn ∈ dom schemes(mdl),
```

Given a map from rolenames to association, which should be all the available nested associations from a given scheme, a mutual recursive call is made for each entry to the function *state_domain_scheme*. Again 'q′ is the qualification used to reach the current scheme.

```
state_domain_ass :
    Model′ × Qualification × (Name ~m~> Association) →
        QualifiedName-set
state_domain_ass(mdl, q, todos) ≡
    if todos = [ ] then {}
    else
        let next = hd dom todos in
            state_domain_scheme(
                 mdl, q ⌢ ⟨next⟩, supplier(todos(next))) ∪
            state_domain_ass(mdl, q, todos \ {next})
        end
    end
```

This functions is called for each object and association of kind Parameter or Nested in the model from the functions *wf_object* and *wf_association* respectively. In order to check the well−formedness of an scheme instantiation it is necessary to first check that there is an object for each formal parameter. This is delegated to the function *wf_actual_parameters*. If this is fulfilled it is checked that the actual parameters statically implements the formal parameters. The parameters for the function is a map of the available object names mapped to their instantiating association, the name of the scheme which is being instantiated, and link between the formal and actual parameters with fitting information for the actual.

```
--FILE:wf_scheme_instantiation.rsl
wf_scheme_instantiation :
    Model′ × (Name ~m~> Association) × Name ×
    ActualParameters ~~>
        Bool
wf_scheme_instantiation(mdl, avail_objm, supplier, apm) ≡
    let fpm = associations_param(mdl, supplier) in
        wf_actual_parameters(mdl, dom avail_objm, fpm, apm) ∧
        (∀ fp : Name •
            fp ∈ dom fpm ⇒
                let (apn, ap_fit) = apm(fp) in
                    static_implement(
                            mdl,
                            (scheme_name(
                                    mdl, supplier(avail_objm(apn))),
```

```
                                ap_fit), supplier(fpm(fp)))
                    end)
        end,
```

wf_actual_parameters: The function checks the well−formedness of the *ActualParameters* type, which is dependent on its contexts.

- For each formal parameter there must be an actual parameter.
- The actual parameters must all be available from within the client context. The name of the object that is currently being instantiated must be excluded, this is however done by the caller.
- When instantiating parameters, nested objects are not available (this is actually not checked here but is carried out in wf_association().

As parameters is given the set of names of the available objects, a set of formal parameter names (only the domain of the map is used) and the actual parameters. Remember that the *ActualParameters* is a map from formal parameter names to actual parameters and fitting.

```
wf_actual_parameters :
    Model′ × Name-set × (Name m→ Association) ×
    ActualParameters →
        Bool
wf_actual_parameters(mdl, avail_objs, fpm, apm) ≡
    let
        required_objs =
            {apn |
             (apn, fit) : Name × Fitting •
                 (apn, fit) ∈ rng apm}
    in
        dom fpm = dom apm ∧ required_objs ⊆ avail_objs
    end
```

C.1.7 wf_extend.rsl

```
scheme wf_extend =
    extend auxiliary with
    class
```

wf_extends: All extend relations in the model must be well−formed, this is checked by wf_extend.

```
wf_extends : Model′ → Bool
wf_extends(mdl) ≡
    (∀ rid : RID •
         rid ∈ dom extends(mdl) ⇒ wf_extend(mdl, rid)),
```

wf_extend: An extend relation is well−formed is the ends of the relations are two distinct schemes present in the model.

```
wf_extend : Model′ × RID → Bool
wf_extend(mdl, r) ≡
    r ∈ dom extends(mdl) ∧
    let e = extends(mdl)(r) in
        client(e) ≠ supplier(e) ∧
        {client(e), supplier(e)} ⊆ dom schemes(mdl)
    end
```

wf_no_of_extends: A given scheme must at most extend one other scheme. Or said differently, the number of extend relations in the model with the same client name must be less than or equal to one.

```
--FILE:wf_no_of_extends.rsl
wf_no_of_extends : Model′ → Bool
wf_no_of_extends(mdl) ≡
  (∀ n : Name •
    n ∈ schemes(mdl) ⇒
      let
        sup_ext =
          {rid |
            rid : RID •
              rid ∈ extends(mdl) ∧
              client_of(mdl, rid) = n}
      in
        card sup_ext ≤ 1
      end)
```

wf_unique_rolenames: All objects available from a given scheme must have a unique rolename (identifier).

```
--FILE:wf_unique_rolenames.rsl
wf_unique_rolenames : Model′ → Bool
wf_unique_rolenames(mdl) ≡
  (∀ n : Name •
    n ∈ schemes(mdl) ⇒
      let
        ridl = rid_set2list(extend_relations(mdl, n)),
        rnl =
          ⟨rolename(associations(mdl)(r)) |
            r in ridl⟩
      in
        len rnl = card elems rnl
      end),
```

extend_relations: A set of unique relation id′s is computed, where each id represents a single association relation with the scheme *n* as client. The *associations()* function is not applicable here since it maps rolenames to associations and we are interested in duplicate rolenames. All relation id′s are unique even though the rolenames are not.

```
extend_relations : Model′ × Name ~→ RID-set
extend_relations(mdl, n) ≡
  let ss = suppliers_ext(mdl, n) in
    {r |
     r : RID •
       r ∈ associations(mdl) ∧
       client_of(mdl, r) = n} ∪
    if dom ss = {} then {}
    else extend_relations(mdl, hd dom ss)
    end
  end
pre n ∈ schemes(mdl)
```

Given a name of a scheme one basic class expression is created. It is the union of the class expression of the given scheme, and all parents of the scheme.

```
extend_history : Model′ × Name ~→ ClassExpr
extend_history(mdl, n) ≡
  let
    ss = suppliers_ext(mdl, n), s = schemes(mdl)(n)
```

```
    in
        if dom ss = {} then public_decl_only(s)
        else
            public_decl_only(s) ⌢ extend_history(mdl, hd ss)
        end
    end
pre n ∈ schemes(mdl),
```

Removes any hidden declarations from a class expression.

```
public_decl_only : ClassExpr → ClassExpr
public_decl_only(s) ≡
    mk_ClassExpr(
        ⟨(td, visi) |
            (td, visi) in types(s) • visi = Public⟩,
        ⟨(vd, visi) |
            (vd, visi) in values(s) • visi = Public⟩,
        ⟨(vd, visi) |
            (vd, visi) in variables(s) • visi = Public⟩,
        ⟨(cd, visi) |
            (cd, visi) in channels(s) • visi = Public⟩,
        axioms(s)),
```

Creates the union of two class expressions. In the Scheme Diagram syntax the clauses are represented by listed in order to maintain the correct display order for the user. Hence the class expressions are actually concatenated.

```
⌢ : ClassExpr × ClassExpr → ClassExpr
child ⌢ parent ≡
    mk_ClassExpr(
        types(child) ⌢ types(parent),
        values(child) ⌢ values(parent),
        variables(child) ⌢ variables(parent),
        channels(child) ⌢ channels(parent),
        axioms(child) ⌢ axioms(parent)),
```

Converts a set of relation ids to a list of ids. The order of the list is irrelevant.

```
rid_set2list : RID-set → RID*
rid_set2list(rids) ≡
    if rids = {} then ⟨⟩
    else
        let rid = hd rids in
            ⟨rid⟩ ⌢ rid_set2list(rids \ {rid})
        end
    end,
```

Converts a set of names to a list of names. Actually the same as above. One could consider to make the Scheme Diagram types global and make a general functions in a parameterised scheme

```
name_set2list : Name-set → Name*
name_set2list(ns) ≡
    if ns = {} then ⟨⟩
    else
        let n = hd ns in
            ⟨n⟩ ⌢ name_set2list(ns \ {n})
        end
    end,
```

Transform a list of list of names to a list of names. The signature of the function explains it all.

```
name_dlist2list : (Name*)* → Name*
```

```
name_dlist2list(nll) ≡
    if nll = ⟨⟩ then ⟨⟩
    else hd nll ⌢ name_dlist2list(tl nll)
    end
```

C.1.8 wf_association.rsl

```
scheme wf_association =
    extend wf_object with
    class
```

Check if all associations in the model are well−formed

```
wf_associations : Model′ → Bool
wf_associations(mdl) ≡
    (∀ rid : RID •
        rid ∈ dom associations(mdl) ⇒
            wf_association(mdl, rid)),
```

An association is well−formed if the association kind and multiplicity is well−formed. Additionally if the kind is parameter or nested the scheme instantiation must be well−formed. It is not necessary to state that the client is different from supplier since the model may not be circular. TODO: document use of wf_scheme_instantiation.

```
wf_association : Model′ × RID → Bool
wf_association(mdl, rid) ≡
    rid ∈ dom associations(mdl) ⇒
        wf_kind(mdl, rid) ∧ wf_multiplicity(mdl, rid) ∧
        let a = associations(mdl)(rid) in
            (is_global(a) ∨
             let
                 aom = associations(mdl, client(a)),
                 aom′ =
                     [ n ↦ aom(n) |
                         n : Name •
                             n ∈ aom ∧
                             ((is_parameter(a) ∧ ∼ is_nested(aom(n))) ∨
                              ∼ is_parameter(a)) ]
             in
                 wf_scheme_instantiation(
                         mdl, aom′ \ {rolename(a)}, supplier(a),
                         actual_parameters(a))
             end)
        end,
```

The association kind denotes if an association is nested, parameter or global. The supplier of a global association must be an object, and since an object already has a name the rolename of the association must be empty. A client is normally a scheme. It is, however, necessary to allow objects to be clients since global objects can be used as actual parameters in a scheme instantiation. Parameter and nested associations are relations between two schemes and the rolename must be non−empty. Unique rolenames is verified by *wf_unique_rolenames()*.

```
wf_kind : Model′ × RID → Bool
wf_kind(mdl, rid) ≡
    rid ∈ dom associations(mdl) ∧
    let a = associations(mdl)(rid) in
        case kind(a) of
            Global →
                supplier(a) ∈ dom objects(mdl) ∧
```

```
            client(a) ∈
                dom schemes(mdl) ∪ dom objects(mdl) ∧
            rolename(a) = "",
        _ →
            {client(a), supplier(a)} ⊆ dom schemes(mdl) ∧
            rolename(a) ≠ ""
    end
end,
```

The multiplicity (array index) is well−formed if the type expression is well−formed

```
wf_multiplicity : Model′ × RID → Bool
wf_multiplicity(mdl, rid) ≡
    rid ∈ dom associations(mdl) ∧
    let a = associations(mdl)(rid) in
        a ∈ rng associations(mdl) ⇒
            case mul(a) of
                None → true,
                Index(binding, expr) →
                    wf_type_expr(mdl, client(a), expr)
            end
    end,
```

Returns the actual parameters for a given association. Global association does not have actual parameters (they are present in the Object type) and will always produce the empty map.

```
actual_parameters : Association → ActualParameters
actual_parameters(a) ≡
    case kind(a) of
        Nested(_, ap) → ap,
        Parameter(ap) → ap,
        Global → [ ]
    end
```

C.1.9 wf_model.rsl

```
scheme wf_model =
    extend wf_association with
    class
```

The Model type represents a well−formed Scheme Diagram.

```
Model = {| mdl : Model′ • wf_model(mdl) |}
```

wf_model: Collects all the predicates defined for the model. If a Model′ fulfill this predicate then it is well−formed.

```
wf_model : Model′ → Bool
wf_model(mdl) ≡
    wf_module_names(mdl) ∧ wf_relation_ids(mdl) ∧
    wf_non_cyclic(mdl) ∧ wf_associations(mdl) ∧
    wf_extends(mdl) ∧ wf_no_of_extends(mdl) ∧
    wf_unique_rolenames(mdl) ∧ wf_objects(mdl) ∧
    wf_schemes(mdl) ∧ wf_implements(mdl),

wf_module_names : Model′ → Bool
wf_module_names(mdl) ≡
    dom schemes(mdl) ∩ dom objects(mdl) = {},
```

```
    wf_relation_ids : Model′ → Bool
    wf_relation_ids(mdl) ≡
        dom associations(mdl) ∩ dom extends(mdl) ∩
        dom implements(mdl) = {},
```

The predicate cyclic returns true if there are any cyclic paths/relations in the model. That is a path from any module to itself via any combination of relations.

```
    - - FILE:wf_non_cyclic.rsl
    wf_non_cyclic : Model′ → Bool
    wf_non_cyclic(mdl) ≡
        ∼ (∃ s : Name •
                s ∈ schemes(mdl) ∧ path(mdl, s, s)),
```

Determines if there is a path between two schemes in the model, where a path is a directed relation between two schemes with any number of intermediate schemes.

```
    - - FILE:path1.rsl
    path : Model′ × Name × Name ~→ Bool
    path(mdl, org, dst) ≡
        path(mdl, org, dom suppliers(mdl, org), dst)
    pre {org, dst} ⊆ modules(mdl),

    - - FILE:path2.rsl
    path : Model′ × Name × Name-set × Name ~→ Bool
    path(mdl, org, intermediate, dst) ≡
        intermediate ≠ {} ∧
        (dst ∈ intermediate ∨
         let n = hd intermediate in
             path(mdl, n, dom suppliers(mdl, n), dst) ∨
             path(mdl, dst, intermediate \ {n}, dst)
         end)
    pre {org, dst} ⊆ modules(mdl)
- - ENDFILE
```

C.2 Translation of Scheme Diagram to RSL.

```
scheme transltr(RSL : rslprint, SD : wf_model) =
class
```

Translate an entire scheme diagram to rsl. ml : List of all module names in the model m : A single module name Determine the leafs and translate them first. First all the schemes are translated. Thus the ordering of the formal parameters are determined which is necessary to translate scheme instantiations.

```
    transltr : SD.Model → RSL.specification
    transltr(mdl) ≡
        let
            scheme_spec = leaf_scheme(mdl, ⟨⟩),
            object_spec =
                ⟨RSL.module_decl_from_object_decl(
                        transltr_Object(
                            scheme_spec, on, SD.objects(mdl)(on))) |
                    on in SD.name_set2list(dom SD.objects(mdl))⟩
        in
            scheme_spec ⌢ object_spec
        end,
```

Todo: Must include not only the class expression of an object but also the object itself. This is due to the added global association between global objects.

```
transltr_context : SD.Model → RSL.Context
transltr_context(mdl) ≡
  [ n ↦
    (dom SD.suppliers(mdl, n) ∪
     if n ∈ SD.objects(mdl)
     then {SD.instance_of(SD.objects(mdl)(n))}
     else {}
     end) | n : SD.Name • n ∈ SD.modules(mdl) ],
```

Remove globals in ass. todos may not be empty! dones : the set of schemes which have already been translated. If a scheme is to be translated then cannot have any associations or all the suppliers of the associations must already have been translated. Remember that it is not allowed to make circular association relations. Put a if−statement around todos if it is empty? It should however not be necessary!?? Besides returning the finished specification after translating all the schemes, it is also necessary to pass the specification that has been done so far since there may be schemes that are dependent on the already translated schemes. This could fx be parameterised schemes. specification is not used as a parameter since we start out with the empty specification (the specification type cannot be empty). It is a requirement that the returned specification has at least one element thus the Model given as parameter must not be empty. Due to the well−formedness conditions for the model there will always be at least one scheme if the model is not empty. This is because all objects must be an instance of a scheme. dones: the set of scheme names which have been translated. todos: the set of scheme names which have not been translated Argh: todos $= \{\} \wedge$ spec $= \langle\rangle$ must not be true

```
leaf_scheme :
  SD.Model × RSL.module_decl* ~→
    RSL.module_decl*
leaf_scheme(mdl, spec) ≡
  let
    dones = elems spec2namel(spec),
    todos = dom SD.schemes(mdl) \ dones
  in
    if todos = {} then spec
    else
      let
        next = next_scheme(mdl, todos, dones),
        spec′ =
          spec ⌢
          ⟨RSL.module_decl_from_scheme_decl(
                transltr_Scheme(mdl, spec, next))⟩
      in
        leaf_scheme(mdl, spec′)
      end
    end
  end
pre dom SD.schemes(mdl) ≠ {},
```

∗ Selects the next scheme to be translated from the model. The name of the scheme that is chosen may only depend on other schemes which have already been translated. Thus the supplier schemes of all the associations in which the selected scheme is client must already have been translated. The extend does not directly add restrictions to the choice however if the supplier is a parameterised scheme then the client will ″`inherit`″ the formal parameters. The functions 'associations′ determines all the associations which the scheme is a client in, including inherited associations.

```
next_scheme :
  SD.Model × SD.Name-set × SD.Name-set ~→ SD.Name
next_scheme(mdl, todos, dones) ≡
  let
    next : SD.Name •
```

```
            next ∈ todos ∧
            dom SD.suppliers(mdl, next) ⊆ dones
        in
            next
        end
    pre
        todos ≠ {} ∧
        todos ∪ dones = dom SD.schemes(mdl) ∧
        todos ∩ dones = {},
```

Extracts the names of the modules in the specification.

```
    spec2namel : RSL.module_decl* → SD.Name*
    spec2namel(spec) ≡
        if spec = ⟨⟩ then ⟨⟩
        else
            case hd spec of
                RSL.module_decl_from_scheme_decl(
                        RSL.mk_scheme_decl(sdl)) →
                    ⟨case sd of
                            RSL.mk_scheme_def(n, ofsp, ce) →
                                id2Name(n)
                        end | sd in sdl⟩,
                RSL.module_decl_from_object_decl(
                        RSL.mk_object_decl(odl)) →
                    ⟨case od of
                            RSL.mk_object_def(n, ofap, ce) →
                                id2Name(n)
                        end | od in odl⟩
            end ⌢ spec2namel(tl spec)
        end,
```

A scheme is translated into RSL by first determining all the declarations within the scheme including object declarations from nested associations. If there are names that should be hidden a hidden class expression is created.

```
    transltr_Scheme :
        SD.Model × RSL.module_decl* × SD.Name ~→
            RSL.scheme_decl
    transltr_Scheme(mdl, spec, n) ≡
        let
            s = SD.name2scheme(mdl, n),
            (hide_me, decll) =
                transltr_NestedAss(spec, mdl, n) ⌢
                transltr_TypeDecls(SD.types(s)) ⌢
                transltr_ValueDecls(SD.values(s)) ⌢
                transltr_VariableDecls(SD.variables(s)) ⌢
                transltr_ChannelDecls(SD.channels(s)) ⌢
                transltr_AxiomDecls(SD.axioms(s)),
            ce =
                RSL.class_expr_from_basic_class_expr(
                        RSL.mk_basic_class_expr(decll)),
            ce′ =
                if hide_me = ⟨⟩ then ce
                else
                    RSL.class_expr_from_hiding_class_expr(
                            RSL.mk_hiding_class_expr(
                                ⟨RSL.defined_item_from_id_or_op(
                                        RSL.id_or_op_from_id(hm)) |
                                    hm in hide_me⟩, ce))
```

```
        end,
    ce'' = transltr_Extend(mdl, spec, n, ce')
  in
    RSL.mk_scheme_decl(
        ⟨RSL.mk_scheme_def(
            n, transltr_formal_param(spec, mdl, n),
            ce'')⟩)
  end
pre n ∈ dom SD.schemes(mdl),
```

For a given scheme all the nested associations are found and returned in rsl syntax as object declarations. mdl : Entire model sn : The scheme in which the objects are instantiated. objm : Mapping of names to nested associations of the objects - - that are to be instantiated. obj_name_list : The list of the names of to be instantiated.

```
transltr_NestedAss :
  RSL.module_decl* × SD.Model × SD.Name ~→
    RSL.id* × RSL.decl*
transltr_NestedAss(spec, mdl, sn) ≡
  let
    objm' = SD.associations_nested(mdl, sn),
    objm =
      [ n ↦ objm'(n) |
        n : SD.Name •
          n ∈ objm' ∧ SD.client(objm'(n)) = sn ],
    obj_name_list = SD.name_set2list(dom objm),
    obj_def_list =
      ⟨RSL.mk_object_def(
          objn,
          transltr_Multiplicity(SD.mul(objm(objn))),
          RSL.class_expr_from_scheme_instantiation(
           RSL.mk_scheme_instantiation(
             RSL.name_from_qualified_id(
              RSL.mk_qualified_id(
                RSL.opt_qual_none,
                SD.supplier(objm(objn)))),
             actual_param(
              spec, SD.supplier(objm(objn)),
              actual_parameters(objm(objn)))))) |
        objn in obj_name_list⟩,
    idl =
      ⟨aname |
        aname in obj_name_list •
          case SD.kind(objm(aname)) of
            SD.Nested(visi, _) → visi = SD.Private,
            _ → false
          end⟩
  in
    (idl,
      if obj_def_list = ⟨⟩ then ⟨⟩
      else
        ⟨RSL.decl_from_object_decl(
            RSL.mk_object_decl(obj_def_list))⟩
      end)
  end,


transltr_Multiplicity :
  SD.Multiplicity → RSL.opt_formal_array_parameter
```

```
transltr_Multiplicity(mul) ≡
    case mul of
        SD.None → ⟨⟩,
        SD.Index(b, te) →
            ⟨RSL.typing_from_single_typing(
                    RSL.mk_single_typing(
                        RSL.binding_from_id_or_op(
                            RSL.id_or_op_from_id(b)),
                        transltr_TypeExpr(te)))⟩
    end,
```

transltr_Extend: Returns an extended class expression if the scheme which the class expression is the body of is an extension of another scheme. It is assumed that a scheme can only extend one other scheme (part of the well−formedness for the scheme diagram).

```
transltr_Extend :
    SD.Model × RSL.module_decl* × SD.Name ×
    RSL.class_expr ~→
        RSL.class_expr
transltr_Extend(mdl, spec, scheme_name, ce) ≡
    let sem = SD.suppliers_ext(mdl, scheme_name) in
       if dom sem = {} then ce
       else
        let
            sup_name = hd dom sem,
            sup_rsl_name =
               RSL.name_from_qualified_id(
                   RSL.mk_qualified_id(
                       RSL.opt_qual_none,
                       transltr_Name(sup_name)))
        in
            RSL.class_expr_from_extending_class_expr(
               RSL.mk_extending_class_expr(
                RSL.class_expr_from_scheme_instantiation(
                   RSL.mk_scheme_instantiation(
                    sup_rsl_name,
                    actual_param(
                       spec, sup_name,
                       [ fp ↦ (fp, [ ]) |
                        fp : SD.Name •
                           fp ∈
                            dom SD.associations_param(
                               mdl, sup_name) ])),ce))
        end
       end
    end
pre card dom SD.suppliers_ext(mdl, scheme_name) ≤ 1,
```

transltr_Object: Translation of globally declared objects must be done after tranlation of schemes. The specification which is a result of the translation of the schemes are given as parameter. This is necessary since it contains the ordering of the formal parameters for parameterized schemes, which is needed to place the actual parameters in a corresponding order when instantiating the scheme. The name of the scheme is without qualification since all schemes are globally available, nested schemes are not allowed. Support for global object arrays have been omitted (the empty list in the mk_object_decl)!?

```
transltr_Object :
    RSL.specification × SD.Name × SD.Object →
        RSL.object_decl
transltr_Object(spec, n, o) ≡
    let
```

```
        scheme_name =
            RSL.name_from_qualified_id(
                RSL.mk_qualified_id(
                    RSL.opt_qual_none, SD.instance_of(o))),
        ce =
            RSL.class_expr_from_scheme_instantiation(
                RSL.mk_scheme_instantiation(
                    scheme_name,
                    actual_param(
                        spec, SD.instance_of(o),
                        SD.actual_parameters(o))))
    in
        RSL.mk_object_decl(
            ⟨RSL.mk_object_def(n, ⟨⟩, ce)⟩)
    end,
```

actual_param:

```
actual_param :
    RSL.specification × SD.Name × SD.ActualParameters →
        RSL.opt_actual_scheme_parameter
actual_param(spec, inst_of, ap) ≡
    let fpol = formal_param_ordering(spec, inst_of) in
        transltr_ActualParameters(fpol, ap)
    end,
```

formal_param_ordering: The list of formal parameters for a given scheme is determined using the already translated specification for the schemes, spec, and the name of the scheme. Some assumptions are made regarding the provided information. The length of 'fpll′ should be either 0 or 1. If it is more than one it would mean that there are several formal parameters with the same name, which is a violation of the well−formedness. Only scheme names are searched since anything else would be irrelevant. This function only make sense for schemes. If the scheme is parameterised then the list of formal parameter names will be returned otherwise the empty list. A note: A 'specification′ is a list of scheme_decl and object_decl. A scheme_decl is again a list of 'scheme_def′. The first list is traversed using recursion, the second using a comprehended list expression.

```
formal_param_ordering :
    RSL.module_decl* × SD.Name → SD.Name*
formal_param_ordering(spec, n) ≡
    if spec = ⟨⟩ then ⟨⟩
    else
        case hd spec of
            RSL.module_decl_from_scheme_decl(
                RSL.mk_scheme_decl(sdl)) →
                let
                    (found, ofsp) =
                        find_first_scheme_def(sdl, n)
                in
                    if found
                    then formal_param_to_name_list(ofsp)
                    else formal_param_ordering(tl spec, n)
                    end
                end,
            _ → formal_param_ordering(tl spec, n)
        end
    end,
```

WRITE COMMENT

```
find_first_scheme_def :
    RSL.scheme_def* × SD.Name →
```

```
        Bool × RSL.opt_formal_scheme_parameter
    find_first_scheme_def(sdl, n) ≡
        if sdl = ⟨⟩ then (false, ⟨⟩)
        else
            case hd sdl of
                RSL.mk_scheme_def(id, ofp, _) →
                    if id = n then (true, ofp)
                    else find_first_scheme_def(tl sdl, n)
                    end,
                _ → find_first_scheme_def(tl sdl, n)
            end
        end,
```

WRITE COMMENT

```
    formal_param_to_name_list :
        RSL.opt_formal_scheme_parameter → SD.Name*
    formal_param_to_name_list(ofsp) ≡
        if ofsp = ⟨⟩ then ⟨⟩
        else
            case hd ofsp of
                RSL.mk_formal_scheme_argument(
                        RSL.mk_object_def(id, _, _)) →
                    ⟨id2Name(id)⟩,
                _ → ⟨⟩
            end ⌢ formal_param_to_name_list(tl ofsp)
        end,
```

id2Name: Conversion of names from the rsl syntax to the scheme diagram names.

```
    id2Name : RSL.id → SD.Name
    id2Name(id) ≡ id,
```

transltr_ActualParameters: The schemes in the diagram are translated before the global objects so the ordering of the formal parameters are settled. 'fpol′ is a list of formal parameter names in the same order as chosen by the translation of the scheme. Again ap is a mapping from the formal parameter names to the actual parameter names with fitting information. The names of fpol must match the domain of ap.

```
    transltr_ActualParameters :
        SD.Name* × SD.ActualParameters ~→
            RSL.opt_actual_scheme_parameter
    transltr_ActualParameters(fpol, ap) ≡
        if fpol = ⟨⟩ then RSL.opt_asp_none
        else
            RSL.opt_actual_scheme_parameter_from_actual_scheme_parameter(
                RSL.mk_actual_scheme_parameter(
                    ⟨let
                        (apn, fit) = ap(fpn),
                        objname =
                            RSL.object_expr_from_object_name(
                                RSL.name_from_qualified_id(
                                 RSL.mk_qualified_id(
                                    RSL.opt_qual_none, apn)))
                    in
                        if fit = [ ] then objname
                        else
                            RSL.object_expr_from_fitting_object_expr(
                                RSL.mk_fitting_object_expr(
                                    objname, transltr_Fitting(fit)))
                        end
```

```
                    end | fpn in fpol⟩))
    end
  pre elems fpol = dom ap,
```

transltr_Fitting: Fitting is specifies the use of a new name instead of an old one. This functions just convert the information into the form used by the rslsyntax.

```
  transltr_Fitting : SD.Fitting → RSL.rename_pair*
  transltr_Fitting(fit) ≡
    ⟨RSL.mk_rename_pair(
          RSL.defined_item_from_id_or_op(
              RSL.id_or_op_from_id(nn)),
          RSL.defined_item_from_id_or_op(
              RSL.id_or_op_from_id(fit(nn)))) |
      nn in SD.name_set2list(dom fit)⟩,
```

transltr_TypeDecls: Iterates through the list of type declarations from the sd and returns two lists. The first is a list of id′s which must be hidden and the second

```
  transltr_TypeDecls :
    SD.TypeDecls → RSL.id* × RSL.decl*
  transltr_TypeDecls(tdl) ≡
    let (idl, tdefl) = transltr_TypeDecls′(tdl) in
      (idl,
        if tdefl = ⟨⟩ then ⟨⟩
        else
          ⟨RSL.decl_from_type_decl(
                RSL.mk_type_decl(tdefl))⟩
        end)
    end,

  transltr_TypeDecls′ :
    SD.TypeDecls → RSL.id* × RSL.type_def*
  transltr_TypeDecls′(tdl) ≡
    if tdl = ⟨⟩ then (⟨⟩, ⟨⟩)
    else
      let
        (idl, tdefl) = transltr_TypeDecl(hd tdl),
        (idl′, tdefl′) = transltr_TypeDecls′(tl tdl)
      in
        (idl ⌢ idl′, tdefl ⌢ tdefl′)
      end
    end,
```

transltr_TypeDecl: Translates a type declaration from the scheme diagram to rslsyntax.

```
  transltr_TypeDecl :
    SD.TypeDecl × SD.Visibility →
      RSL.id* × RSL.type_def*
  transltr_TypeDecl(td, visi) ≡
    (private_idl(visi, SD.td_name(td)),
      ⟨case td of
          SD.SortDef(name) →
            RSL.type_def_from_sort_def(
              RSL.mk_sort_def(transltr_Name(name))),
          SD.VariantDef(name, expr) →
            RSL.type_def_from_variant_def(
              RSL.mk_variant_def(
                  transltr_Name(name),
                  ⟨transltr_Variant(v) | v in expr⟩)),
```

```
            SD.UnionDef(name, expr) →
                RSL.type_def_from_union_def(
                    RSL.mk_union_def(
                            transltr_Name(name),
                            ⟨case e of
                                SD.udName((name, qualification)) →
                                 RSL.name_or_wildcard_from_type_name(
                                    transltr_QualifiedName(
                                     name, qualification)),
                                SD.udWildcard → RSL.nw_wildcard
                                end | e in expr⟩)),
            SD.ShortRecordDef(name, components) →
                RSL.type_def_from_short_record_def(
                    RSL.mk_short_record_def(
                            transltr_Name(name),
                            ⟨transltr_ComponentKind(ck) |
                                ck in components⟩)),
            SD.AbbreviationDef(name, expr) →
                RSL.type_def_from_abbreviation_def(
                    RSL.mk_abbreviation_def(
                            transltr_Name(name),
                            transltr_TypeExpr(expr)))
        end⟩),

transltr_Variant : SD.Variant → RSL.variant
transltr_Variant(v) ≡
    case v of
        SD.Wildcard →
            RSL.variant_from_constructor(RSL.con_wildcard),
        SD.Constructor(name) →
            RSL.variant_from_constructor(
                    RSL.constructor_from_id_or_op(
                        RSL.id_or_op_from_id(transltr_Name(name)))),
        SD.RecordVariant(SD.Constructor(name), components) →
            RSL.variant_from_record_variant(
                    RSL.mk_record_variant(
                        RSL.constructor_from_id_or_op(
                                RSL.id_or_op_from_id(
                                    transltr_Name(name))),
                        ⟨transltr_ComponentKind(ck) |
                            ck in components⟩))
    end,
```

transltr_ComponentKind: transltr_Constructor : SD.Constructor → RSL.constructor transltr_Constructor(c) is case c of SD.Wildcard → RSL.wildcard, SD.mk_Constructor(name) → RSL.id_or_op_from_id(transltr_Name(name)) end,

```
transltr_ComponentKind :
    SD.ComponentKind → RSL.component_kind
transltr_ComponentKind(ck) ≡
    RSL.mk_component_kind(
            case SD.destructor(ck) of
                SD.Destructor(name) →
                    RSL.opt_destructor_from_destructor(
                        RSL.id_or_op_from_id(transltr_Name(name))),
                SD.deNone → RSL.opt_dest_none
            end, transltr_TypeExpr(SD.expr(ck)),
            case SD.reconstructor(ck) of
                SD.Reconstructor(name) →
```

```
                RSL.opt_reconstructor_from_reconstructor(
                    RSL.id_or_op_from_id(transltr_Name(name))),
            SD.reNone → RSL.opt_reco_none
        end)

value
 transltr_ValueDecls :
    SD.ValueDecls → RSL.id* × RSL.decl*
 transltr_ValueDecls(vdl) ≡
    let (idl, vdefl) = transltr_ValueDecls′(vdl) in
        (idl,
            if vdefl = ⟨⟩ then ⟨⟩
            else
                ⟨RSL.decl_from_value_decl(
                        RSL.mk_value_decl(vdefl))⟩
            end)
    end,
```

transltr_ValueDecls': Convert a list of value declarations from the scheme diagram to a corresponding list of value definitions. The function simply recurses through the list and for each element call the transltr_ValueDecl function.

```
 transltr_ValueDecls′ :
    SD.ValueDecls → RSL.id* × RSL.value_def*
 transltr_ValueDecls′(vdl) ≡
    if vdl = ⟨⟩ then (⟨⟩, ⟨⟩)
    else
        let
            (idl, vdefl) = transltr_ValueDecl(hd vdl),
            (idl′, vdefl′) = transltr_ValueDecls′(tl vdl)
        in
            (idl ⌢ idl′, vdefl ⌢ vdefl′)
        end
    end,
```

transltr_ValueDecl: Translate a single value declaration from the scheme diagram to a value definition using the RSL syntax. A value declaration is actually a list of value definitions. It is only possible to specify the name and signature in the Scheme Diagram thus it will always be a commented typing in the RSL syntax. The returned id list contains the name of the value declaration if it should be hidden, otherwise empty. The returned value_def list will always have the length one.

```
 transltr_ValueDecl :
    SD.ValueDecl × SD.Visibility →
        RSL.id* × RSL.value_def*
 transltr_ValueDecl(vd, visi) ≡
    (private_idl(visi, SD.vdname(vd)),
        ⟨RSL.value_def_from_commented_typing(
            RSL.mk_commented_typing(
                RSL.typing_from_single_typing(
                    RSL.mk_single_typing(
                     RSL.binding_from_id_or_op(
                        RSL.id_or_op_from_id(
                         transltr_Name(SD.vdname(vd)))),
                     transltr_TypeExpr(SD.vdte(vd)))))))⟩)
```

transltr_VariableDecls: Call the sister function to do the dirty work, and convert the result to an id list and in particular a lidt of declaration.

```
 transltr_VariableDecls :
```

```
    SD.VariableDecls → RSL.id* × RSL.decl*
  transltr_VariableDecls(vdl) ≡
    let (idl, vdefl) = transltr_VariableDecls′(vdl) in
      (idl,
        if vdefl = ⟨⟩ then ⟨⟩
        else
          ⟨RSL.decl_from_variable_decl(
              RSL.mk_variable_decl(vdefl))⟩
        end)
    end,
```

transltr_VariableDecls': Recurse through the list of variable declarations and each entry to the RSL syntax.

```
  transltr_VariableDecls′ :
    SD.VariableDecls →
      RSL.id* × RSL.variable_def*
  transltr_VariableDecls′(vdl) ≡
    if vdl = ⟨⟩ then (⟨⟩, ⟨⟩)
    else
      let
        (idl, vdefl) = transltr_VariableDecl(hd vdl),
        (idl′, vdefl′) = transltr_VariableDecls′(tl vdl)
      in
        (idl ⌢ idl′, vdefl ⌢ vdefl′)
      end
    end,
```

transltr_VariableDecl: Convert a single Variable declaration from the Scheme Diagram to rsl. If the visibility given as parameter is set to private then the name of the variable is added to the returned id list. The returned variable list will always be one in length.

```
  transltr_VariableDecl :
    SD.VariableDecl × SD.Visibility →
      RSL.id* × RSL.variable_def*
  transltr_VariableDecl(vd, visi) ≡
    (private_idl(visi, SD.vdname(vd)),
      ⟨RSL.variable_def_from_single_variable_def(
          RSL.mk_single_variable_def(
              transltr_Name(SD.vdname(vd)),
              transltr_TypeExpr(SD.vdte(vd)),
              RSL.opt_init_none))⟩)

value
  transltr_ChannelDecls :
    SD.ChannelDecls → RSL.id* × RSL.decl*
  transltr_ChannelDecls(vdl) ≡
    let (idl, cdefl) = transltr_ChannelDecls′(vdl) in
      (idl,
        if cdefl = ⟨⟩ then ⟨⟩
        else
          ⟨RSL.decl_from_channel_decl(
              RSL.mk_channel_decl(cdefl))⟩
        end)
    end,

  transltr_ChannelDecls′ :
    SD.ChannelDecls →
      RSL.id* × RSL.channel_def*
  transltr_ChannelDecls′(cdl) ≡
    if cdl = ⟨⟩ then (⟨⟩, ⟨⟩)
```

```
    else
        let
            (idl, cdefl) = transltr_ChannelDecl(hd cdl),
            (idl′, cdefl′) = transltr_ChannelDecls′(tl cdl)
        in
            (idl ⌢ idl′, cdefl ⌢ cdefl′)
        end
    end,

transltr_ChannelDecl :
    SD.ChannelDecl × SD.Visibility →
        RSL.id* × RSL.channel_def*
transltr_ChannelDecl(cd, visi) ≡
    (private_idl(visi, SD.cdname(cd)),
        ⟨RSL.channel_def_from_single_channel_def(
                RSL.mk_single_channel_def(
                        transltr_Name(SD.cdname(cd)),
                        transltr_TypeExpr(SD.cdte(cd))))⟩)

value
transltr_AxiomDecls :
    SD.AxiomDecls → RSL.id* × RSL.decl*
transltr_AxiomDecls(adl) ≡
    if adl = ⟨⟩ then (⟨⟩, ⟨⟩)
    else
        (⟨⟩,
            ⟨RSL.decl_from_axiom_decl(
                    RSL.mk_axiom_decl(
                            ⟨transltr_AxiomDecl(ad) | ad in adl⟩))⟩
        )
    end,

transltr_AxiomDecl : SD.AxiomDecl → RSL.axiom_def
transltr_AxiomDecl(ad) ≡
    RSL.mk_axiom_def(
            RSL.opt_axiom_naming_from_axiom_naming(
                RSL.mk_axiom_naming(
                        transltr_Name(SD.adname(ad)))),
            RSL.ve_val_l(RSL.bool_literal(true)))
```

(contatenation): Concatenates two tuples both containing a list of id′s and a list of declarations.

```
⌢ :
    (RSL.id* × RSL.decl*) ×
    (RSL.id* × RSL.decl*) →
        (RSL.id* × RSL.decl*)
a ⌢ b ≡
    let (idl, d) = a, (idl′, d′) = b in
        (idl ⌢ idl′, d ⌢ d′)
    end,
```

private_idl: If the visibility is private the list returns a list containing only the name given as parameter; otherwise the returned list is empty.

```
private_idl : SD.Visibility × SD.Name → RSL.id*
private_idl(visi, n) ≡
    if visi = SD.Private then ⟨transltr_Name(n)⟩
    else ⟨⟩
    end
```

```
value
 transltr_TypeExpr : SD.TypeExpr → RSL.type_expr
 transltr_TypeExpr(te) ≡
     case te of
         SD.tl_Unit →
             RSL.type_expr_from_type_literal(RSL.tl_Unit),
         SD.tl_Bool →
             RSL.type_expr_from_type_literal(RSL.tl_Bool),
         SD.tl_Int →
             RSL.type_expr_from_type_literal(RSL.tl_Int),
         SD.tl_Nat →
             RSL.type_expr_from_type_literal(RSL.tl_Nat),
         SD.tl_Real →
             RSL.type_expr_from_type_literal(RSL.tl_Real),
         SD.tl_Text →
             RSL.type_expr_from_type_literal(RSL.tl_Text),
         SD.tl_Char →
             RSL.type_expr_from_type_literal(RSL.tl_Char),
         SD.TypeName(n, q) →
             RSL.type_expr_from_name(
                 transltr_QualifiedName(n, q)),
         SD.ProductTypeExpr(tel) →
             RSL.type_expr_from_product_type_expr(
                 RSL.mk_product_type_expr(
                     transltr_TypeExprList(tel))),
         SD.FiniteSetTypeExpr(te) →
             RSL.type_expr_from_set_type_expr(
                 RSL.set_type_expr_from_finite_set_type_expr(
                     RSL.mk_finite_set_type_expr(
                         transltr_TypeExpr(te)))),
         SD.InfiniteSetTypeExpr(te) →
             RSL.type_expr_from_set_type_expr(
                 RSL.set_type_expr_from_infinite_set_type_expr(
                     RSL.mk_infinite_set_type_expr(
                         transltr_TypeExpr(te)))),
         SD.FiniteListTypeExpr(te) →
             RSL.type_expr_from_list_type_expr(
                 RSL.list_type_expr_from_finite_list_type_expr(
                     RSL.mk_finite_list_type_expr(
                         transltr_TypeExpr(te)))),
         SD.InfiniteListTypeExpr(te) →
             RSL.type_expr_from_list_type_expr(
                 RSL.list_type_expr_from_infinite_list_type_expr(
                     RSL.mk_infinite_list_type_expr(
                         transltr_TypeExpr(te)))),
         SD.MapTypeExpr(tdom, trng) →
             RSL.type_expr_from_map_type_expr(
                 RSL.map_type_expr_from_finite_map_type_expr(
                     RSL.mk_finite_map_type_expr(
                         transltr_TypeExpr(tdom),
                         transltr_TypeExpr(trng)))),
         SD.FunctionTypeExpr(param_te, fa, rd) →
             RSL.type_expr_from_function_type_expr(
                 RSL.mk_function_type_expr(
                     transltr_TypeExpr(param_te),
                     transltr_Function_Arrow(fa),
                     transltr_ResultDescr(rd))),
```

```
        SD.BracketedTypeExpr(te) →
            RSL.type_expr_from_bracketed_type_expr(
                RSL.mk_bracketed_type_expr(
                    transltr_TypeExpr(te))),
        SD.SubtypeExpr(t, qn) →
            RSL.type_expr_from_subtype_expr(
                transltr_Subtype(t, qn))
    end,

transltr_TypeExprList :
    SD.TypeExpr* → RSL.type_expr*
transltr_TypeExprList(tel) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
        ⟨transltr_TypeExpr(hd tel)⟩ ⌢
        transltr_TypeExprList(tl tel)
    end,
```

transltr_Subtype: A subtype expression concsists of a type expression and a predicate. Since the Scheme Diagram does not include value expressions, the predicate must be the name of a value function. It is possible to specify the type and function but the binding that is given as parameter to the predicate is arbitrarily chosen.

```
transltr_Subtype :
    SD.TypeExpr × SD.QualifiedName → RSL.subtype_expr
transltr_Subtype(te, r) ≡
    let
        b =
            RSL.binding_from_id_or_op(
                RSL.id_or_op_from_id(″binding_name″))
    in
        RSL.mk_subtype_expr(
            RSL.mk_single_typing(b, transltr_TypeExpr(te)),
            RSL.mk_restriction(
                RSL.ve_name(transltr_QualifiedName(r))))
    end,

transltr_Function_Arrow :
    SD.FunctionArrow → RSL.function_arrow
transltr_Function_Arrow(fa) ≡
    case (fa) of
        SD.fa_total → RSL.fa_total,
        SD.fa_partial → RSL.fa_partial
    end,

transltr_ResultDescr :
    SD.ResultDescr → RSL.result_desc
transltr_ResultDescr(al, te) ≡
    RSL.mk_result_desc(
        (transltr_AccessDescr(al), transltr_TypeExpr(te))),

transltr_AccessDescr :
    SD.AccessDescr* → RSL.accss_desc*
transltr_AccessDescr(adl) ≡
    if adl = ⟨⟩ then ⟨⟩
    else
        let (am, al) = hd adl in
            ⟨RSL.mk_accss_desc(
                transltr_AccessMode(am),
                (⟨transltr_Access(a) | a in al⟩))⟩ ⌢
```

```
            transltr_AccessDescr(tl adl)
        end
    end,

transltr_AccessMode : SD.AccessMode → RSL.accss_mode
transltr_AccessMode(am) ≡
    case (am) of
        SD.am_read → RSL.am_read,
        SD.am_write → RSL.am_write,
        SD.am_in → RSL.am_in,
        SD.am_out → RSL.am_out
    end,

transltr_Access : SD.Access → RSL.accss
transltr_Access(ac) ≡
    case (ac) of
        SD.NameAccess((n, q)) →
            RSL.accss_from_acc_name(
                transltr_QualifiedName(n, q)),
        SD.EnumeratedAccess(al) →
            RSL.accss_from_enumerated_accss(
                RSL.mk_enumerated_accss(
                    transltr_AccessList(al))),
        SD.CompletedAccess(q) →
            RSL.accss_from_completed_accss(
                RSL.mk_completed_accss(
                    transltr_Qualification(q)))
    end,

transltr_AccessList : SD.Access* → RSL.accss*
transltr_AccessList(al) ≡
    if al = ⟨⟩ then ⟨⟩
    else
        ⟨transltr_Access(hd al)⟩ ⌢
        transltr_AccessList(tl al)
    end
```

transltr_QualifiedName: The use of 'name′ in the scheme diagram can be somewhat misleading. It represents a name of a scheme, object, declaration, etc. In RSL the corresponding type is called id. Translating a qualified name is translated by translating the qualification and name seperately.

```
transltr_QualifiedName : SD.QualifiedName → RSL.name
transltr_QualifiedName(n, q) ≡
    RSL.name_from_qualified_id(
        RSL.mk_qualified_id(
            transltr_Qualification(q), transltr_Name(n))),
```

transltr_Qualification: Translates the representation of qualification. In the Scheme Diagram qualification is represented by a list of names. In the RSL syntax it is a recursive data structure.

```
transltr_Qualification :
    SD.Qualification → RSL.opt_qualification
transltr_Qualification(nl) ≡
    if nl = ⟨⟩ then RSL.opt_qual_none
    else
        RSL.opt_qualification_from_qualification(
            RSL.mk_qualification(
                RSL.object_expr_from_object_name(
                    RSL.name_from_qualified_id(
```

```
                RSL.mk_qualified_id(
                    transltr_Qualification(
                        ⟨nl(i) |
                            i in ⟨1 .. len nl − 1⟩⟩),
                    transltr_Name(nl(len nl)))))))
    end,
```

transltr_Name: Translates the Name type of the Scheme Diagram to is corresponding type in the RSL syntax.

```
transltr_Name : SD.Name → RSL.id
transltr_Name(n) ≡ n,
```

transltr_formal_param: Determine the formal parameters for a given scheme. In the Scheme Diagram formal parameters are represented by associations of kind Parameter. It is however not only the assoications for the specified scheme but also for all its suppliers. There is no ordering of the parameters in the Scheme Diagram and the order in which they are written in RSL is not important. Hence an arbitrary ordering is chosen by the list 'assl'. If a formal parameter is a parameterised scheme then the actual parameters must be specified.

```
transltr_formal_param :
    RSL.module_decl* × SD.Model × SD.Name ~→
        RSL.opt_formal_scheme_parameter
transltr_formal_param(spec, mdl, scheme_name) ≡
    let
        assm = SD.associations_param(mdl, scheme_name),
        assl = SD.name_set2list(dom assm)
    in
        ⟨RSL.mk_formal_scheme_argument(
                RSL.mk_object_def(
                    transltr_Name(fp),
                    transltr_Multiplicity(SD.mul(assm(fp))),
                    RSL.class_expr_from_scheme_instantiation(
                            RSL.mk_scheme_instantiation(
                                RSL.name_from_qualified_id(
                                 RSL.mk_qualified_id(
                                    RSL.opt_qual_none,
                                    SD.supplier(assm(fp)))),
                                actual_param(
                                 spec, SD.supplier(assm(fp)),
                                 actual_parameters(assm(fp))))))) |
            fp in assl⟩
    end
pre scheme_name ∈ SD.schemes(mdl),
```

actual_parameters: Extracts the map from formal parameters to actual parameters from an association. Global associations do not have actual parameters since they are already instantiated, thus the returned map will always be empty.

```
actual_parameters :
    SD.Association → SD.ActualParameters
actual_parameters(a) ≡
    case SD.kind(a) of
        SD.Nested(_, ap) → ap,
        SD.Parameter(ap) → ap,
        SD.Global → [ ]
    end
```

C.3 Imperative Scheme Diagram

C.3.1 RSL Part

transltr, ../rslsyntax/exec/rslprint, wf_model, examples

After adding all elements of the diagram, run a post thing: − After adding an object the state of the object should be updated. Otherwise it will not be wellformed. −

```
hide RSL, SD, TL, mdl, spec, context, max_or_zero, new_rid in
class
object
    RSL : rslprint, SD : examples, TL : transltr(RSL, SD)
```

An empty model

```
    empty_mdl : SD.Model′ =
        SD.mk_Model′([ ], [ ], [ ], [ ], [ ])
variable
    texp : SD.TypeExpr := SD.tl_Int,
    texplist : SD.TypeExpr* := ⟨⟩,
    mdl : SD.Model′ := empty_mdl,
    spec : RSL.module_decl* := ⟨⟩,
    context : RSL.Context := [ ]
```

Resets the model to be empty

```
    reset : Unit → write mdl, spec, context Unit
    reset() ≡
        mdl := empty_mdl ; spec := ⟨⟩ ; context := [ ],

    wf_model : Unit → write mdl Bool
    wf_model() ≡
    let
        newobjs = [ n ↦ add_object_state(SD.objects(mdl)(n)) | n : SD.Name • n ∈ SD.objects(mdl) ]
    in
        mdl := SD.replace_objects(newobjs, mdl) ;
        SD.wf_model(mdl)
    end,
```

Adds state to objects.

```
    add_object_state : SD.Object → read mdl SD.Object
    add_object_state(obj) ≡
    let
        instance_of = SD.instance_of(obj),
        ap = SD.actual_parameters(obj),
        state_dom = if instance_of ∈ SD.modules(mdl)
then
  SD.state_domain(mdl, instance_of)
else {} end,
        state = [ qn ↦ ″″ | qn : SD.QualifiedName • qn ∈ state_dom ]
    in
        SD.mk_Object(instance_of, ap, state)
```

```
    end
```

Adds a scheme

```
    add_scheme : Text → write mdl Bool
    add_scheme(sn) ≡
        if sn ∈ SD.schemes(mdl) then false
        else
            mdl :=
            SD.replace_schemes(
                    SD.schemes(mdl) † [ sn ↦ SD.empty_scheme ], mdl) ;
            true
```

Adds an object

```
    add_object : Text × Text → write mdl Bool
    add_object(on, instance_of) ≡
        if on ∈ SD.objects(mdl) then false
        else
            mdl :=
            SD.replace_objects(
                    SD.objects(mdl) †
                    [ on ↦ SD.mk_Object(instance_of, [ ], [ ]) ], mdl) ;
            true
        end,
```

Adds actual parameter information to an object oname: name of the object for_par : the name of the formal parameter act_par : the name of the actual parameter

```
    add_object_ap : Text × Text × Text → write mdl Bool
    add_object_ap(oname, for_par, act_par) ≡
        if oname ∈ SD.objects(mdl)
        then
            let
                obj = SD.objects(mdl)(oname),
instance_of = SD.instance_of(obj),
par = SD.actual_parameters(obj),
state = SD.state(obj),
newpar = par † [ for_par ↦ (act_par, [ ]) ]
            in
                mdl :=
                SD.replace_objects(
                    SD.objects(mdl) †
                        [ oname ↦ SD.mk_Object(instance_of, newpar, state) ], mdl); true
            end
        else
            false
        end,
```

Adds fitting information to an actual parameter of an object (formal parameter is domain in map) oname : name of the object for_par : the name of the formal parameter to be changed fitdom : the name of the fitted element fitrng : the name of the new fitting of the above element

```
    add_object_ap_fit : Text × Text × Text × Text → write mdl Bool
    add_object_ap_fit(oname, for_par, fitdom, fitrng) ≡
        if oname ∈ SD.objects(mdl)
        then
            let
                obj = SD.objects(mdl)(oname),
instance_of = SD.instance_of(obj),
```

```
par = SD.actual_parameters(obj),
state = SD.state(obj),
                (act_par,fit) = par(for_par),
                newfit = fit † [ fitdom ↦ fitrng ],
                newpar = par † [ for_par ↦ (act_par, newfit) ]
            in
                mdl :=
                SD.replace_objects(
                    SD.objects(mdl) †
                        [ oname ↦ SD.mk_Object(instance_of, newpar, state) ], mdl); true
            end
        else
            false
        end
```

Adds an extend relation between two schemes

```
    add_extend : Text × Text → write mdl Nat
    add_extend(c, s) ≡
            let rid = new_rid(mdl) in
                mdl :=
                SD.replace_extends(
                        SD.extends(mdl) †
                        [ rid ↦ SD.mk_Extend(c, s) ], mdl) ;
                rid
            end,
```

Adds an implement relation between two schemes

```
    add_implement : Text × Text → write mdl Nat
    add_implement(c, s) ≡

            let rid = new_rid(mdl) in
                mdl :=
                SD.replace_implements(
                        SD.implements(mdl) †
                        [ rid ↦ SD.mk_Implement(c, s) ], mdl) ;
                rid
            end,
```

Adds a global association between two schemes

```
    add_global : Text × Text → write mdl Nat
    add_global(c, s) ≡
            let rid = new_rid(mdl) in
                mdl :=
                SD.replace_associations(
                        SD.associations(mdl) †
                        [ rid ↦
                         SD.mk_Association(
                                c, SD.Global, s, '''', SD.None) ], mdl) ;
                rid
            end,
```

Adds a nested association between two schemes

```
    add_nested : Text × Text × Text × Bool→ write mdl Nat
    add_nested(c, s, rolename, vis) ≡
            let
                rid = new_rid(mdl),
```

```
visib = if vis then SD.Public else SD.Private end
            in
                mdl :=
                SD.replace_associations(
                        SD.associations(mdl) †
                        [ rid ↦
                         SD.mk_Association(
                                c, SD.Nested(visib,[ ] ), s, rolename, SD.None) ], mdl) ;
                rid
```

Adds actual parameters on a nested association rid : rid of the nested association for_par : the name of the formal parameter act_par : the name of the actual parameter

```
    add_nested_ap : Int × Text × Text → write mdl Nat
    add_nested_ap(rid, for_par, act_par) ≡
        if rid ∈ rids(mdl)
        then
            let
                assoc = SD.associations(mdl)(rid),
                c = SD.client(assoc),
                s = SD.supplier(assoc),
rolename = SD.rolename(assoc),
                nkind = SD.kind(assoc)
            in
               case nkind of
                SD.Nested(vis,par) →
                    let
                       newpar = par † [ for_par ↦ (act_par, [ ]) ]
                    in
                     mdl :=
                         SD.replace_associations(
                             SD.associations(mdl) †
                             [ rid ↦
                                 SD.mk_Association(
                                     c, SD.Nested(vis,newpar ), s, rolename, SD.None) ], mdl) ;
                         rid
                    end
                    ,
                _ → 0
               end
            end
        else
            0
```

Adds fitting information to an actual parameter (formal parameter is domain in map) rid : rid of the nested association for_par : the name of the formal parameter to be changed fitdom : the name of the fitted element fitrng : the name of the new fitting of the above element

```
    add_nested_ap_fit : Int × Text × Text × Text → write mdl Nat
    add_nested_ap_fit(rid,  for_par, fitdom, fitrng) ≡
        if rid ∈ rids(mdl)
        then
            let
                assoc = SD.associations(mdl)(rid),
                c = SD.client(assoc),
                s = SD.supplier(assoc),
rolename = SD.rolename(assoc),
                nkind = SD.kind(assoc)
            in
               case nkind of
```

```
            SD.Nested(vis,par) →
                let
                (act_par,fit) = par(for_par),
                    newfit = fit † [ fitdom ↦ fitrng ],
                    newpar = par † [ for_par ↦ (act_par, newfit) ]
                in
                    mdl :=
                        SD.replace_associations(
                            SD.associations(mdl) †
                            [ rid ↦
                                SD.mk_Association(
                                    c, SD.Nested(vis,newpar ), s, rolename, SD.None) ], mdl) ;
                    rid
                end,
            _ → 0
            end
        end
    else
        0
```

Adds a parameter association between two schemes

```
add_parameter : Text × Text × Text→ write mdl Nat
add_parameter(c, s, rolename) ≡
    let rid = new_rid(mdl) in
        mdl :=
        SD.replace_associations(
                SD.associations(mdl) †
                [ rid ↦
                 SD.mk_Association(
                        c, SD.Parameter([ ]), s, rolename, SD.None) ], mdl) ;
        rid
```

Adds actual parameter information for a parameter association

```
add_parameter_ap : Int × Text × Text → write mdl Nat
add_parameter_ap(rid, domain, range) ≡
    if rid ∈ rids(mdl)
    then
        let
            assoc = SD.associations(mdl)(rid),
            c = SD.client(assoc),
            s = SD.supplier(assoc),
rolename = SD.rolename(assoc),
            nkind = SD.kind(assoc)
        in
            case nkind of
            SD.Parameter(par) →
                let
                    newpar = par † [ domain ↦ (range, [ ]) ]
                in
                    mdl :=
                        SD.replace_associations(
                            SD.associations(mdl) †
                            [ rid ↦
                                SD.mk_Association(
                                    c, SD.Parameter(newpar ), s, rolename, SD.None) ], mdl) ;
                    rid
                end,
            _ → 0
```

```
            end
        end
    else
        0
```

Adds fitting information for actual parameter for a parameter association rid : rid of the nested association for_par : the name of the formal parameter to be changed fitdom : the name of the fitted element fitrng : the name of the new fitting of the above element

```
add_parameter_ap_fit : Int × Text × Text × Text → write mdl Nat
add_parameter_ap_fit(rid,  for_par, fitdom, fitrng) ≡
    if rid ∈ rids(mdl)
    then
        let
            assoc = SD.associations(mdl)(rid),
            c = SD.client(assoc),
            s = SD.supplier(assoc),
rolename = SD.rolename(assoc),
            nkind = SD.kind(assoc)
        in
            case nkind of
            SD.Parameter(par) →
                let
                (act_par,fit) = par(for_par),
                   newfit = fit † [ fitdom ↦ fitrng ],
                   newpar = par † [ for_par ↦ (act_par, newfit) ]
                in
                    mdl :=
                        SD.replace_associations(
                            SD.associations(mdl) †
                            [ rid ↦
                                SD.mk_Association(
                                    c, SD.Parameter(newpar ), s, rolename, SD.None) ], mdl) ;
                    rid
                end,
            _ → 0
            end
        end
    else
        0
```

Adds a sort to a scheme sname: name of scheme where sort is added vis: visibility, true = public, false = private sort : the sort to be added

add_type_sort : Text × Bool × Text → write mdl Bool add_type_sort(sname, vis, sort) is if sname ∈ dom SD.schemes(mdl) then let ce = SD.schemes(mdl)(sname), typedecl = SD.types(ce), newtypedecl = if vis then typedecl ⌢⟨(SD.SortDef(sort),SD.Public)⟩ else typedecl ⌢⟨(SD.SortDef(sort),SD.Private)⟩ end, ce′ = SD.replace_types(newtypedecl, ce) in mdl := SD.replace_schemes(SD.schemes(mdl) † [sname ↦ ce′], mdl); true end else false end,

Adds an type def to a scheme, texp must be set prior to this it is not a sort. sname : scheme where type is added vis : visibility, true = public typename : name of new type

```
add_type : Text × Bool × Text × Bool  → read texp write mdl Bool
add_type(sname, vis, typename, sort) ≡
 if sname ∈ dom SD.schemes(mdl)
 then
```

```
        let
            ce = SD.schemes(mdl)(sname),
            typedecl = SD.types(ce),
            newtypedecl =
                if vis then
 if sort then
  typedecl ⌢⟨(SD.SortDef(typename),SD.Public)⟩
 else
                        typedecl ⌢⟨(SD.AbbreviationDef(typename,texp),SD.Public)⟩
 end
                else
                    if sort then
  typedecl ⌢⟨(SD.SortDef(typename),SD.Private)⟩
 else
                        typedecl ⌢⟨(SD.AbbreviationDef(typename,
  texp),SD.Private)⟩
                    end
                end,
            ce′ = SD.replace_types(newtypedecl, ce)
        in
            mdl :=
                SD.replace_schemes(
                    SD.schemes(mdl) †
                        [sname ↦ ce′], mdl);
            true
        end
    else
        false
    end,
```

Adds a value to a scheme, texp must be set prior to this sname: name of scheme where value is added vis: visibility, true = public, false = private vname : the value name to be added

```
    add_value : Text × Bool × Text → write mdl read texp Bool
    add_value(sname, vis, vname) ≡
    if sname ∈ dom SD.schemes(mdl)
    then
        let
            ce = SD.schemes(mdl)(sname),
            valdecls = SD.values(ce),
            newvaldecls =
                if vis then
                    valdecls ⌢⟨(SD.mk_ValueDecl(vname, texp),SD.Public)⟩
                else
                    valdecls ⌢⟨(SD.mk_ValueDecl(vname, texp),SD.Private)⟩
                end,
            ce′ = SD.replace_values(newvaldecls, ce)
        in
            mdl :=
                SD.replace_schemes(
                    SD.schemes(mdl) †
                        [sname ↦ ce′], mdl);
            true
        end
    else
        false
    end,
```

Adds a variable to a scheme, texp must be set prior to this sname: name of scheme where variable is added vis: visibility, true = public, false = private vname : the value name to be added

```
add_variable : Text × Bool × Text → write mdl read texp Bool
add_variable(sname, vis, vname) ≡
if sname ∈ dom SD.schemes(mdl)
then
  let
    ce = SD.schemes(mdl)(sname),
    vardecls = SD.variables(ce),
    newvardecls =
      if vis then
        vardecls ⌢⟨(SD.mk_VariableDecl(vname, texp),SD.Public)⟩
      else
        vardecls ⌢⟨(SD.mk_VariableDecl(vname, texp),SD.Private)⟩
      end,
    ce′ = SD.replace_variables(newvardecls, ce)
  in
    mdl :=
      SD.replace_schemes(
        SD.schemes(mdl) †
          [ sname ↦ ce′ ], mdl);
    true
  end
else
  false
end,
```

Adds a channel to a scheme, texp must be set prior to this sname: name of scheme where channel is added vis: visibility, true = public, false = private cname : the channel name to be added

```
add_channel : Text × Bool × Text → write mdl read texp Bool
add_channel(sname, vis, cname) ≡
if sname ∈ dom SD.schemes(mdl)
then
  let
    ce = SD.schemes(mdl)(sname),
    chdecls = SD.channels(ce),
    newchdecls =
      if vis then
        chdecls ⌢⟨(SD.mk_ChannelDecl(cname, texp),SD.Public)⟩
      else
        chdecls ⌢⟨(SD.mk_ChannelDecl(cname, texp),SD.Private)⟩
      end,
    ce′ = SD.replace_channels(newchdecls, ce)
  in
    mdl :=
      SD.replace_schemes(
        SD.schemes(mdl) †
          [ sname ↦ ce′ ], mdl);
    true
  end
else
  false
end,
```

Adds an axiom to a scheme, texp must be set prior to this sname: name of scheme where axiom is added aname : the axiom name to be added

```
add_axiom : Text × Text → write mdl read texp Bool
```

```
add_axiom(sname, aname) ≡
if sname ∈ dom SD.schemes(mdl)
then
    let
        ce = SD.schemes(mdl)(sname),
        axdecls = SD.axioms(ce),
        newaxdecls = axdecls ⌢⟨SD.mk_AxiomDecl(aname)⟩,
        ce′ = SD.replace_axioms(newaxdecls, ce)
    in
        mdl :=
            SD.replace_schemes(
                SD.schemes(mdl) †
                    [ sname ↦ ce′ ], mdl);
        true
    end
else
    false
end,
```

Creates a native type expression

```
temp_texp_type : Text → write texp Unit
temp_texp_type(literal) ≡
    case literal of
                                ″unit″→ texp := SD.tl_Unit,
                                ″bool″ → texp := SD.tl_Bool,
                                ″int″ → texp := SD.tl_Int,
                                ″nat″ → texp := SD.tl_Nat,
                                ″real″ → texp := SD.tl_Real,
                                ″text″ → texp := SD.tl_Text,
                                ″char″ → texp := SD.tl_Char,
                                _ → texp:= SD.TypeName(literal, ⟨⟩)
```

Creates a qualified typename

temp_texp_typename : Text → write texp Unit temp_texp_typename(tname) is texp := SD.TypeName(tname, ⟨⟩),

```
temp_texp_typename_qualification : Text → write texp Unit
temp_texp_typename_qualification(qual) ≡
case texp of
    SD.TypeName(name, oldqual) → texp := SD.TypeName(name, oldqual
    ⌢⟨qual⟩)
end,
```

Creates a list type expression

```
temp_texp_list : Bool → write texp Unit
temp_texp_list(finite) ≡
if finite then
    texp := SD.FiniteListTypeExpr(texp)
else
    texp := SD.InfiniteListTypeExpr(texp)
end,
```

Creates a set type expression

```
temp_texp_set : Bool → write texp Unit
temp_texp_set(finite) ≡
```

```
    if finite then
        texp := SD.FiniteSetTypeExpr(texp)
    else
        texp := SD.InfiniteSetTypeExpr(texp)
    end,
```

Creates a product type expression

```
    temp_texp_product : Unit → write texp,texplist Unit
    temp_texp_product() ≡
    texp := SD.ProductTypeExpr(texplist);
    texplist := ⟨⟩,
```

Creates a function type expression

```
    temp_texp_function : Unit → write texp,texplist Unit
    temp_texp_function() ≡
        texp := SD.FunctionTypeExpr(hd texplist, SD.fa_total,
            (⟨⟩, hd tl texplist));
        texplist := ⟨⟩
    pre len texplist = 2,
```

Adds the current type expression (texp) to a list of type expressions

```
add_texp2list : Unit → write texplist read texp Int
add_texp2list() ≡
texplist := texplist ⌢⟨texp⟩; len texplist
```

new_rid returns a new unique identifier

```
    new_rid : SD.Model′ → SD.RID
    new_rid(mdl) ≡
        1 +
        max_or_zero(
                dom SD.associations(mdl) ∪
                dom SD.extends(mdl) ∪ dom SD.implements(mdl)),

    max_or_zero : Nat-set → Nat
    max_or_zero(ints) ≡
        case card ints of
            0 → 0,
            1 → hd ints,
            _ →
                let i = hd ints, i′ = max_or_zero(ints \ {i}) in
                    if i > i′ then i else i′ end
                end
```

returns the set of relation identifiers present in the model

```
    rids : SD.Model′ → read mdl Int-set
    rids(mdl) ≡
    let
        assos = SD.associations(mdl)
    in
       dom assos
    end

value
    seperator : Text = ″;;″,
```

print_mdl prints the current model stored in the variable mdl

```
    print_mdl : Unit → read mdl write spec, context Text
    print_mdl() ≡
        spec := TL.transltr(mdl) ;
        context := TL.transltr_context(mdl) ;
        pmdl(spec),

    pmdl : RSL.module_decl* → read context Text
    pmdl(x) ≡
        case x of
            ⟨⟩ → "",
            ⟨a⟩ ⌢ ⟨⟩ →
                RSL.print_context(context, extract_module_id(a)) ⌢
                "\n" ⌢ RSL.print_module_decl(a),
            ⟨a⟩ ⌢ b →
                RSL.print_context(context, extract_module_id(a)) ⌢
                "\n" ⌢ RSL.print_module_decl(a) ⌢ seperator ⌢
                pmdl(b)
        end,

    extract_module_id : RSL.module_decl → Text
    extract_module_id(md) ≡
        case md of
            RSL.module_decl_from_scheme_decl(
                  RSL.mk_scheme_decl(sdl)) →
                case hd sdl of
                    RSL.mk_scheme_def(n, _, _) → n
                end,
                RSL.module_decl_from_object_decl(RSL.mk_object_decl(obl))
                →
                case hd obl of
                    RSL.mk_object_def(n, _, _) → n
                end
        end

end
```

C.3.2 SchemeDiagramInterface.java

Listing C.1: SchemeDiagramInterface.java

```
/* OK */
package rsl.esde.libsd;

public class SchemeDiagramInterface {

    /** Resets the model to be empty*/
    public native void reset();

    /** Checks if the model is wellformed*/
    public native boolean wfModel();

    /** Adds a scheme to the model*/
    public native boolean addScheme(String name);
    /** Adds an object to the model*/
    public native boolean addObject(String name, String instanceOf);
    /** Adds actual parameter information for nested association*/
    public native boolean addObjectAp(String oname, String for_par, String act_par)
```

```
/** Adds fitting information for an actual parameter of a nested association*/
public native boolean addObjectApFit(String oname, String for_par, String fit_dom,
        String fit_rng);

/** Adds an extend relation between to schemes*/
public native int addExtend(String client, String supplier);
/** Adds an implement relation between to schemes*/
public native int addImplement(String client, String supplier);
/** Adds a global association between to schemes*/
public native int addGlobal(String client, String supplier);

/** Adds a nested association between to schemes*/
public native int addNested(String client, String supplier, String rolename,
        boolean visibility);
/** Adds actual parameter information for nested association*/
public native int addNestedAp(int rid, String for_par, String act_par);
/** Adds fitting information for an actual parameter of a nested association*/
public native int addNestedApFit(int rid, String for_par, String fit_dom,
        String fit_rng);
/** Adds a parameter association between to schemes*/
public native int addParameter(String client, String supplier, String rolename);
/** Adds actual parameter information for parameter association*/
public native int addParameterAp(int rid, String for_par, String act_par);
/** Adds fitting information for an actual parameter of a nested association*/
public native int addParameterApFit(int rid, String for_par, String fit_dom,
        String fit_rng);

/** Adds a type to a scheme*/
public native boolean addType(String scheme, boolean visibility,
        String typename, boolean sort);
/** Adds a value, type expression must be created first*/
public native boolean addValue(String scheme, boolean visibility,
        String valuename);
/** Adds a variable, type expression must be created first*/
public native boolean addVariable(String scheme, boolean visibility,
        String variablename);
/** Adds a channel, type expression must be created first*/
public native boolean addChannel(String scheme, boolean visibility,
        String channelname);
/** Adds a axiom*/
public native boolean addAxiom(String scheme, String axiomname);

/** Creates a type expression with a type in RSL model*/
public native void tempTexpType(String literal);
/** Adds a qualification to a type expression */
public native void tempTexpTypenameQualification(String qualification);
/** Creates a list (finite or infinite) type expression in RSL model*/
public native void tempTexpList(boolean finite);
/** Creates a set (finite or infinite) type expression in RSL model*/
public native void tempTexpSet(boolean finite);

/** Creates a product type expression in RSL model*/
public native void tempTexpProduct();
/** Creates a function type expression in RSL model*/
public native void tempTexpFunction();
/** Adds a type expression to a list of type expressions in RSL model*/
public native int addTexp2List();
```

```
        /** Prints the model and returns the complete resulting RSL specification
         * as a string*/
    public native String printMdl();


    /** Loads the compiled RSL C++ library*/
    static { System.loadLibrary("sd"); }
}
```

C.3.3 rsl_esde_libsd_SchemeDiagramInterface.h

Listing C.2: rsl_esde_libsd_SchemeDiagramInterface.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class rsl_esde_libsd_SchemeDiagramInterface */

#ifndef _Included_rsl_esde_libsd_SchemeDiagramInterface
#define _Included_rsl_esde_libsd_SchemeDiagramInterface
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    reset
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_reset
  (JNIEnv *, jobject);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    wfModel
 * Signature: ()Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_wfModel
  (JNIEnv *, jobject);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addScheme
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addScheme
  (JNIEnv *, jobject, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addObject
 * Signature: (Ljava/lang/String;Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addObject
  (JNIEnv *, jobject, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addObjectAp
```

```
 * Signature: (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addObjectAp
  (JNIEnv *, jobject, jstring, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addObjectApFit
 * Signature: (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addObjectApFit
  (JNIEnv *, jobject, jstring, jstring, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addExtend
 * Signature: (Ljava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addExtend
  (JNIEnv *, jobject, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addImplement
 * Signature: (Ljava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addImplement
  (JNIEnv *, jobject, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addGlobal
 * Signature: (Ljava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addGlobal
  (JNIEnv *, jobject, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addNested
 * Signature: (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Z)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addNested
  (JNIEnv *, jobject, jstring, jstring, jstring, jboolean);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addNestedAp
 * Signature: (ILjava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addNestedAp
  (JNIEnv *, jobject, jint, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addNestedApFit
 * Signature: (ILjava/lang/String;Ljava/lang/String;Ljava/lang/String;)I
 */
```

```
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addNestedApFit
  (JNIEnv *, jobject, jint, jstring, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addParameter
 * Signature: (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addParameter
  (JNIEnv *, jobject, jstring, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addParameterAp
 * Signature: (ILjava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addParameterAp
  (JNIEnv *, jobject, jint, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addParameterApFit
 * Signature: (ILjava/lang/String;Ljava/lang/String;Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addParameterApFit
  (JNIEnv *, jobject, jint, jstring, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addType
 * Signature: (Ljava/lang/String;ZLjava/lang/String;Z)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addType
  (JNIEnv *, jobject, jstring, jboolean, jstring, jboolean);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addValue
 * Signature: (Ljava/lang/String;ZLjava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addValue
  (JNIEnv *, jobject, jstring, jboolean, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addVariable
 * Signature: (Ljava/lang/String;ZLjava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addVariable
  (JNIEnv *, jobject, jstring, jboolean, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addChannel
 * Signature: (Ljava/lang/String;ZLjava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addChannel
  (JNIEnv *, jobject, jstring, jboolean, jstring);
```

```
/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addAxiom
 * Signature: (Ljava/lang/String;Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addAxiom
  (JNIEnv *, jobject, jstring, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    tempTexpType
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpType
  (JNIEnv *, jobject, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    tempTexpTypenameQualification
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpTypenameQualifi
  (JNIEnv *, jobject, jstring);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    tempTexpList
 * Signature: (Z)V
 */
JNIEXPORT void JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpList
  (JNIEnv *, jobject, jboolean);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    tempTexpSet
 * Signature: (Z)V
 */
JNIEXPORT void JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpSet
  (JNIEnv *, jobject, jboolean);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    tempTexpProduct
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpProduct
  (JNIEnv *, jobject);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    tempTexpFunction
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpFunction
  (JNIEnv *, jobject);

/*
```

```
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    addTexp2List
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_addTexp2List
  (JNIEnv *, jobject);

/*
 * Class:     rsl_esde_libsd_SchemeDiagramInterface
 * Method:    printMdl
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_rsl_esde_libsd_SchemeDiagramInterface_printMdl
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

C.3.4 rsl_esde_libsd_SchemeDiagramInterface.cc

Listing C.3: rsl_esde_libsd_SchemeDiagramInterface.cc

```
#include <jni.h>
#include <stdio.h>
#include "rsl_esde_libsd_SchemeDiagramInterface.h"
#include "imperative.cc"
#include "convert.cc"



JNIEXPORT void JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_reset(JNIEnv *env, jobject obj)
{
  reset();
  return;
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_wfModel(JNIEnv *env, jobject obj)
{
 return wf_model();
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addScheme(JNIEnv *env, jobject obj,
jstring name)
{
  return add_scheme(jstr2rsl(env, name));
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addObject(JNIEnv *env, jobject obj,
jstring name, jstring instance_of) {
  return add_object(jstr2rsl(env, name), jstr2rsl(env, instance_of));
}
```

```
JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addObjectAp(JNIEnv *env, jobject obj,
jstring oname, jstring for_par, jstring act_par) {
  return add_object_ap(jstr2rsl(env, oname), jstr2rsl(env, for_par),jstr2rsl(env, act_par)
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addObjectApFit (JNIEnv *env,
jobject obj, jstring oname, jstring for_par, jstring fit_dom, jstring fit_rng)
{
  return add_object_ap_fit(jstr2rsl(env, oname), jstr2rsl(env, for_par),
                jstr2rsl(env, fit_dom), jstr2rsl(env, fit_rng));
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addExtend(JNIEnv *env, jobject obj,
jstring client, jstring supplier)
{
  return add_extend(jstr2rsl(env, client), jstr2rsl(env, supplier));
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addImplement(JNIEnv *env, jobject obj,
jstring client, jstring supplier)
{
  return add_implement(jstr2rsl(env, client), jstr2rsl(env, supplier));
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addGlobal(JNIEnv *env, jobject obj,
jstring client, jstring supplier)
{
  return add_global(jstr2rsl(env, client), jstr2rsl(env, supplier));
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addNested(JNIEnv *env, jobject obj,
jstring client, jstring supplier, jstring rolename, jboolean visibility)
{
  return add_nested(jstr2rsl(env, client), jstr2rsl(env, supplier), jstr2rsl(env, rolename
    visibility);
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addNestedAp(JNIEnv *env, jobject obj,
jint rid, jstring for_par, jstring act_par)
{
  return add_nested_ap(rid, jstr2rsl(env, for_par),jstr2rsl(env, act_par));
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addNestedApFit (JNIEnv *env, jobject obj,
jint rid, jstring for_par, jstring fit_dom, jstring fit_rng)
{
  return add_nested_ap_fit(rid, jstr2rsl(env, for_par),jstr2rsl(env, fit_dom),
    jstr2rsl(env, fit_rng));
}
```

```
JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addParameter(JNIEnv *env, jobject obj,
jstring client, jstring supplier, jstring rolename)
{
  return add_parameter(jstr2rsl(env, client), jstr2rsl(env, supplier), jstr2rsl(env
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addParameterAp(JNIEnv *env, jobject obj,
jint rid, jstring for_par, jstring act_par)
{
  return add_parameter_ap(rid, jstr2rsl(env, for_par),jstr2rsl(env, act_par));
}

JNIEXPORT jint JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addParameterApFit (JNIEnv *env, jobject
jint rid, jstring for_par, jstring fit_dom, jstring fit_rng)
{
  return add_parameter_ap_fit(rid, jstr2rsl(env, for_par),jstr2rsl(env, fit_dom),
    jstr2rsl(env, fit_rng));
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addType(JNIEnv *env, jobject obj,
jstring scheme, jboolean visibility, jstring tname, jboolean sort)
{
   return add_type(jstr2rsl(env, scheme),visibility, jstr2rsl(env, tname), sort);
}


JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addValue(JNIEnv *env, jobject obj,
jstring scheme, jboolean visibility, jstring vname )
{
   return add_value(jstr2rsl(env, scheme),visibility, jstr2rsl(env, vname));
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addVariable(JNIEnv *env, jobject obj,
jstring scheme, jboolean visibility, jstring vname )
{
   return add_variable(jstr2rsl(env, scheme),visibility, jstr2rsl(env, vname));
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addChannel(JNIEnv *env, jobject obj,
jstring scheme, jboolean visibility, jstring cname )
{
   return add_channel(jstr2rsl(env, scheme),visibility, jstr2rsl(env, cname));
}

JNIEXPORT jboolean JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addAxiom(JNIEnv *env, jobject obj,
jstring scheme, jstring aname )
{
   return add_axiom(jstr2rsl(env, scheme), jstr2rsl(env, aname));
}
```

```
JNIEXPORT void JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpType(JNIEnv *env, jobject obj,
    jstring literal)
{
   return temp_texp_type(jstr2rsl(env, literal));
}

JNIEXPORT void JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpTypenameQualification(JNIEnv *env,
    jobject obj, jstring qual)
{
   return temp_texp_typename_qualification(jstr2rsl(env, qual));
}

JNIEXPORT void JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpList(JNIEnv *env, jobject obj,
    jboolean finite)
{
   return temp_texp_list(finite);
}

JNIEXPORT void JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpSet(JNIEnv *env, jobject obj,
    jboolean finite)
{
   return temp_texp_set(finite);
}

JNIEXPORT void JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpProduct(JNIEnv *env, jobject obj)
{
   return temp_texp_product();
}

JNIEXPORT void JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_tempTexpFunction(JNIEnv *env, jobject obj)
{
   return temp_texp_function();
}

JNIEXPORT int JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_addTexp2List(JNIEnv *env, jobject obj)
{
   return add_texp2list();
}

JNIEXPORT jstring JNICALL
Java_rsl_esde_libsd_SchemeDiagramInterface_printMdl(JNIEnv *env, jobject obj)
{
  return rsl2jstr(env, print_mdl());
}
```

C.3.5 convert.cc

Listing C.4: convert.cc

```
// Convert RSL_string to jstring
jstring rsl2jstr(JNIEnv *env, RSL_string rstr) {
  return env->NewStringUTF(RSL_to_string(rstr).c_str());
}

// Convert jstring to RSL_string
RSL_string jstr2rsl(JNIEnv *env, jstring jstr) {
  return RSL_string(string(env->GetStringUTFChars(jstr, false)));
}
```

C.4 Test

C.4.1 Applicative

scheme sdtest =
  extend examples **with**
  class
    test_case
      [relations_____]
        relations(report_ex) = {0 .. 9},
      [suppliers_____]
        dom suppliers(report_ex, "B") = {"C", "A", "G"} ∧
        dom suppliers(report_ex, "D") = {},
      [associations_____]
        dom associations(report_ex, "B") =
          {"ce", "cp", "ge", "gp", "f1", "e", "d1"} ∧
        dom associations(report_ex, "E") = {} ∧
        dom associations(global_obj, "C") = {"OB"} ∧
        dom associations(global_obj, "OB") = {"OA"}

    value
      B2G : Model′ × Name × Qualification =
        (report_ex, "B", <."ge"⟩),
      B2D : Model′ × Name × Qualification =
        (report_ex, "B", <."ce", "d2"⟩)

    test_case
      [valid_qualification___]
        valid_qualification(B2G) ∧
        valid_qualification(B2D) ∧
        valid_qualification(report_ex, "B", <."f1"⟩) ∧
        ∼ valid_qualification(report_ex, "C", <."f2"⟩),
      [follow_qualification__]
        let n = follow_qualification(B2G) **in** n = "G" **end** ∧
        let n = follow_qualification(B2D) **in** n = "D" **end**

    test_case
      [path_____]
        path(report_ex, "B", "A") ∧
        path(report_ex, "B", "C") ∧
        path(report_ex, "B", "G") ∧
        path(report_ex, "B", "D") ∧
        path(report_ex, "B", "F") ∧
        path(report_ex, "B", "E") ∧
        path(report_ex, "C", "D") ∧
        path(report_ex, "G", "F") ∧

```
    path(report_ex, "A", "E") ∧
    path(report_ex, "A", "D") ∧
    path(report_ex, "A", "F") ∧
    ∼ path(report_ex, "A", "B") ∧
    ∼ path(report_ex, "C", "B") ∧
    ∼ path(report_ex, "G", "B") ∧
    ∼ path(report_ex, "D", "B") ∧
    ∼ path(report_ex, "F", "B") ∧
    ∼ path(report_ex, "E", "B") ∧
    ∼ path(report_ex, "D", "C") ∧
    ∼ path(report_ex, "F", "G") ∧
    ∼ path(report_ex, "E", "A") ∧
    ∼ path(report_ex, "D", "A") ∧
    ∼ path(report_ex, "F", "A"),
  [cyclic_____________]
```

In the cyclic example it is possible for any given state to go to all other states. Thus it is of course cyclic.

```
    (∀ s : Name •
      s ∈ modules(cyclic_mdl) ⇒
        (∀ s′ : Name •
          s′ ∈ modules(cyclic_mdl) ⇒
            path(cyclic_mdl, s, s′))) ∧
    ∼ wf_non_cyclic(cyclic_mdl) ∧ wf_non_cyclic(empty_mdl)

test_case
  [maximal_type_____________]
    maximal_type(empty_mdl, ⟨⟩, "", tl_Unit) = tl_Unit ∧
    maximal_type(empty_mdl, ⟨⟩, "", tl_Bool) = tl_Bool ∧
    maximal_type(empty_mdl, ⟨⟩, "", tl_Int) = tl_Int ∧
    maximal_type(empty_mdl, ⟨⟩, "", tl_Nat) = tl_Int ∧
    maximal_type(empty_mdl, ⟨⟩, "", tl_Real) = tl_Real ∧
    maximal_type(empty_mdl, ⟨⟩, "", tl_Text) =
      InfiniteListTypeExpr(tl_Char) ∧
    maximal_type(empty_mdl, ⟨⟩, "", tl_Char) = tl_Char ∧
    maximal_type(
      empty_mdl, ⟨⟩, "", BracketedTypeExpr(tl_Int)) =
      BracketedTypeExpr(tl_Int) ∧
    maximal_type(
      empty_mdl, ⟨⟩, "", BracketedTypeExpr(tl_Nat)) =
      BracketedTypeExpr(tl_Int),
  [wf_type_expr_____________]
    let
      te =
        FunctionTypeExpr(
          ProductTypeExpr(⟨tl_Int, tl_Nat⟩),
          fa_total,
          (⟨⟩,
            MapTypeExpr(
              tl_Nat,
              ProductTypeExpr(⟨tl_Char, tl_Real⟩)))))
    in
      wf_type_expr(report_ex, "B", te)
    end

test_case
  [wf_scheme_decl_expr___]
    wf_scheme_decl_expr(stack_mdl, "Stack"),
  [wf_class_expr_________]
```

wf_class_expr(
stack_mdl, ″Stack", schemes(stack_mdl)("Stack″)),
[overloading_____]
wf_class_expr(
test_overload_mdl, ″ok1″,
schemes(test_overload_mdl)(″ok1″)) ∧
∼ wf_class_expr(
test_overload_mdl, ″not1″,
schemes(test_overload_mdl)(″not1″))

test_case
[maximal_class_1_____]
maximal_class(
report_ex, ″A", schemes(report_ex)("A″)) =
schemes(report_ex)(″A″),

[signature_____]
signature(stack_mdl, ″Stack")= (mk_ClassExpr(<.(SortDef("Elem″),Public)⟩,
⟨(mk_ValueDecl(″push",FunctionTypeExpr(TypeName("Elem″,⟨⟩),fa_partial,
(⟨(am_write,⟨NameAccess((″el″,⟨⟩))⟩⟩,tl_Unit))),Public)⟩,
⟨(mk_VariableDecl(″el",InfiniteListTypeExpr(TypeName("Elem″,⟨⟩))),Public)⟩,
⟨⟩,⟨⟩),[ ])

test_case
[static_implement_1_cli]
maximal_class(static_impl_mdl, ″S1″, S1) =
mk_ClassExpr(⟨(AbbreviationDef(″T",tl_Int),Public).>,<.(mk_ValueDecl("x″,tl_Int),Public)
(mk_ValueDecl(″y″,tl_Int),Public)⟩,⟨⟩,⟨⟩,⟨⟩),

[static_implement_1_sup]
maximal_class(static_impl_mdl, ″S", S)=mk_ClassExpr(<.(SortDef("T″),Public)⟩,
⟨(mk_ValueDecl(″x",TypeName("T",<..>)),Public),(mk_ValueDecl("y",TypeName("T″,
⟨⟩,⟨⟩,⟨⟩),

[static_implement_types]
static_implement_types(
schemes(static_impl_mdl)(″S1″),
schemes(static_impl_mdl)(″S″)) ∧
∼ static_implement_types(
schemes(static_impl_mdl)(″S2″),
schemes(static_impl_mdl)(″S″)) ∧
static_implement_types(
schemes(static_impl_mdl)(″S1″),
schemes(static_impl_mdl)(″S2″)) ∧
static_implement_types(
schemes(static_impl_mdl)(″S3″),
schemes(static_impl_mdl)(″S″)) ∧
static_implement_types(
schemes(static_impl_mdl)(″S4″),
schemes(static_impl_mdl)(″S″)) ∧
static_implement_types(
schemes(static_impl_mdl)(″S5″),
schemes(static_impl_mdl)(″S″)) ∧
static_implement_types(
schemes(static_impl_mdl)(″S5″),
schemes(static_impl_mdl)(″S1″)) ∧
static_implement_types(
schemes(static_impl_mdl)(″S5″),

```
        schemes(static_impl_mdl)(″S2″)) ∧
    static_implement_types(
        schemes(static_impl_mdl)(″S5″),
        schemes(static_impl_mdl)(″S4″))

test_case
  [static_implement_param]
    static_implement_param(static_impl_mdl, ″S1", "S″) ∧
    static_implement_param(static_impl_mdl, ″S2", "S″) ∧
    static_implement_param(static_impl_mdl, ″S1", "S2″) ∧
    static_implement_param(static_impl_mdl, ″S3", "S″) ∧
    static_implement_param(static_impl_mdl, ″S4", "S″) ∧
    static_implement_param(static_impl_mdl, ″S5", "S″) ∧
    static_implement_param(static_impl_mdl, ″S5", "S1″) ∧
    static_implement_param(static_impl_mdl, ″S5", "S2″) ∧
    static_implement_param(static_impl_mdl, ″S5", "S4″)

test_case
  [static_implement_____]
    static_implement(static_impl_mdl, ″S1", "S″) ∧
    ∼ static_implement(static_impl_mdl, ″S2", "S″) ∧
    static_implement(static_impl_mdl, ″S1", "S2″) ∧
    static_implement(static_impl_mdl, ″S3", "S″) ∧
    ∼ static_implement(static_impl_mdl, ″S4", "S″) ∧
    static_implement(static_impl_mdl, ″S5", "S″) ∧
    static_implement(static_impl_mdl, ″S5", "S1″) ∧
    static_implement(static_impl_mdl, ″S5", "S2″) ∧
    static_implement(static_impl_mdl, ″S5", "S4″)

test_case
  [state_domain_____]
    state_domain(stack_mdl, ″Stack") = {("el″, ⟨⟩)} ∧
    state_domain(test_var_mdl, ″A″) =
      {(″j", <."d".>), ("k", <."d", "c2″⟩),
       (″i", <."d", "c2".>), ("k", <."c″⟩),
       (″i", <."c".>), ("i", <..>), ("n″, ⟨⟩)}

test_case
  [wf_schemes_____]
    wf_schemes(empty_mdl) ∧ wf_schemes(report_ex) ∧
    wf_schemes(global_obj) ∧ wf_schemes(stack_mdl),
  [wf_implements_____]
    wf_implements(empty_mdl) ∧
    wf_implements(report_ex) ∧
    wf_implements(global_obj) ∧
    wf_implements(stack_mdl),
  [wf_extends_____]
    wf_extends(empty_mdl) ∧ wf_extends(report_ex),
  [wf_objects_____]
    wf_objects(empty_mdl) ∧ wf_objects(report_ex) ∧
    wf_objects(global_obj),
  [wf_associations_____]
    wf_associations(empty_mdl) ∧
    wf_associations(report_ex) ∧
    wf_associations(global_obj),
  [wf_model_____]
    wf_model(empty_mdl) ∧ wf_model(report_ex) ∧
    wf_model(global_obj) ∧ ∼ wf_model(cyclic_mdl)
```

C.4.2 Imperative: printed .rsl files

global_object.esde

Scheme2

scheme Scheme1(s2 : Scheme2) =
  class type type1 **variable** var1 : type1 **end**

scheme Scheme2 = **class end**

Scheme1, Object4

object Object3 : Scheme1(Object4)

Scheme2

object Object4 : Scheme2

qualification.esde

Scheme2

scheme Scheme1(S2 : Scheme2) =
  hide Type2 **in class type** Type2 = S2.S1.S0.Type1 **end**

Scheme3

scheme Scheme2 = **extend** Scheme3 **with class end**

Scheme4

scheme Scheme3 = **class object** S1 : Scheme4 **end**

Scheme5

scheme Scheme4 = **class object** S0 : Scheme5 **end**

scheme Scheme5 = **class type** Type1 **end**

types.esde

scheme Scheme1 =
  hide Type5, val2, var2, channel2 **in**
  class

```
    type
        Type1,
        Type2 = Int,
        Type3 = Type2* → Type1-set,
        Type4 = Type1ω × Type1-infset × Real,
        Type5 = Bool → Nat,
        type6 = Type3

    value
        val1 : Type1 → Bool,
        val2 : Type2 → Char*

    variable var1 : Type1, var2 : Int → Char

    channel channel1 : Unit, channel2 : Int × Nat

    axiom
        [ axiomname ]
            true
end
```

Appendix D

RSL specifications for the RSC

Contents

D.1 RSC syntax 251
D.1.1 RSC_types.rsl 251
D.1.2 RSC_wf.rsl 253
D.2 RSC semantics for one chart 271
D.2.1 RSC_semantics.rsl 271
D.3 RSC collections 287
D.3.1 lsc/LSC_collection.rsl 287
D.4 Test 294
D.4.1 Test of wellformedness conditions 294
D.4.2 Test of semantics 316
D.4.3 Test of collections 327
D.5 CSP and LSC 329
D.5.1 Example 1 329
D.5.2 Example 2 343
D.6 Applicative RSC 346
D.6.1 Types 346
D.6.2 Type object 347
D.6.3 Semantics 347
D.6.4 Semantics formal parameter 357
D.6.5 Account example 357

D.1 RSC syntax

D.1.1 RSC_types.rsl

```
scheme RSC_types =
  class
    type
```

A RSC has a name, prechart, mainchart and a set of created instances. A RSC is modelled consisting of a pre− and mainchart since their use is fundamentally different, even though their syntax is the same. Creations must be the instance names of the instances that are created in the main chart.

RSC′ ::
  name : RSC_Name
  prechart : Chart
  mainchart : Chart
  creations : Inst_Name-**set**,

A chart is a map from instance names to a list of locations in order to ease look up of locations. We will also use the index on the list in order to model the state in the semantics.

Chart = Inst_Name $\overrightarrow{m}$ Location*,
Location :: temp : HotCold event : Event,

An event may be one of the following: action, message input/output, condition, coregion or subchart begin/end. Coregion and subcharts have been included as events in order to simplify the specifications.

Event ==

An action event has a name and an id. It is further unspecified as we want to specify it in RSL later on.

  mk_ActionEvent(Action_Name, aid : ID) |

An input event has an id and an source. The message name is given in the corresponding, uniquely identifiable message output event.

  mk_InputEvent(inmsgid : ID, isender : Address) |

An output event has an id, a name and a destination. No parameters are included as this is later done via variables in section 6.7. The temperature is not included since we only consider hot messages in the semantics.

  mk_OutputEvent(
    outmsgid : ID,
    outpid : Msg_Name,
    dest : Address) |

A condition event has a name, an id, a temperature and a list of instance names. The list denotes which instances share the condition.

  mk_ConditionEvent(
    conname : Cond_Name,
    cid : ID,
    cetemp : HotCold,
    ceshare : Inst_Name-**set**) |

A coregion event has a location list. It must be the sublist of events on the instance that contains the message events that may happen in random order.

```
        mk_CoregionEvent(crlocl : Location*) |
```

A subchart event has a name, an id, a list of instance names, a multiplicity and a location list. The instance names denote the instances among which the subchart is shared. The multiplicity denotes the maximum number of repetitions of the chart.The location* is the part of the instance′s locations that the subchart encloses.

```
        mk_Subchart(
            scname : Subchart_Name,
            scid : ID,
            scshare : Inst_Name-set,
            mult : Multiplicity,
            sclocl : Location*) |
```

A endsubchart event has an id. It is used to denote the end of a subchart with the given id.

```
        mk_EndSubchart(escid : ID) |
```

A stop event is included as the last event on an instance.

```
        StopEvent,
```

Temperature may be hot or cold.

```
    HotCold == Hot | Cold,
```

A multiplicity is a positive number.

```
    Multiplicity = {| n : Nat • n>0 |},
```

An address is Environment or an instance name.

```
    Address == Environment | mk_Address(name : Inst_Name),
```

Identifiers are texts or integers.

```
    Action_Name = Text,
    Cond_Name = Text,
    Inst_Name = Text,
    RSC_Name = Text,
    ID = Int,
    Msg_Name = Text,
    Subchart_Name = Text
end
```

D.1.2 RSC_wf.rsl

RSC_types

scheme RSC_wf =
extend RSC_types **with**
class
type

A RSC is a subtype of RSC' which is wellformed.

RSC = {| l : RSC′ • wf_RSC(l) |},

Auxilliary types for checking for acyclicness.

Order :: id1 : ID id2 : ID, IDp :: id : ID occ : **Nat**

value

Check if a RSC' is wellformed. We must check prechart, mainchart and one wellformedness condition applicable to a RSC as a whole. Remember, prechart and mainchart are two separate charts. The last conditions apply to a RSC as a whole.

wf_RSC : RSC′ → **Bool**
wf_RSC(rsc) ≡
wf_chart(mainchart(rsc)) ∧
wf_chart(prechart(rsc)) ∧
wf_creation(rsc) ∧
wf_prechart_condition(prechart(rsc)) ,

Checks whether a chart is wellformed. A chart can be a prechart or a mainchart.

wf_chart : Chart → **Bool**
wf_chart(chart) ≡
wf_ids_unique(chart) ∧ wf_message_match(chart) ∧
wf_mess_cond_acyclic(chart) ∧ wf_condition_share(chart) ∧
wf_subchart_locations(chart) ∧ wf_subchart_ordered(chart) ∧
wf_subchart_coherent(chart) ∧ wf_subchart_end(chart) ∧
wf_subchart_conditions(chart) ∧ wf_subchart_messages(chart) ∧
wf_subchart_subchart(chart) ∧ wf_coregion_locations(chart) ∧
wf_coregion_messages(chart) ∧ wf_cold_subchart(chart) ∧
wf_cold_mainchart(chart) ∧ wf_last(chart),

1. Messages, conditions and subcharts have IDs that must be unique. Checked by extracting all the ids of the chart and comparing the number of occurences of each id to what it is supposed to be. Fx. the occurence of a message ID must be two. This is done since ID′s must uniquely identify events.

wf_ids_unique : Chart → **Bool**
wf_ids_unique(chart) ≡
let idplist = extract_idps_chart(chart) **in**
(∀ e : IDp • e ∈ **elems** idplist ⇒ occ(e) =
count_ids(idplist, id(e), 0))
end,

2.Messages consist of an output and input. These pairs of events must have the same message ID and have the correct origin−destination addresses. This says very much about messages, including that messages only reference declared instances or environment. Their corresponding locations must have the same temperature.

wf_message_match : Chart → **Bool**
wf_message_match(chart) ≡
(∀ i : Inst_Name •
  i ∈ **dom** chart ⇒
  (∀ ie : Event •
    ie ∈ inputEvents(chart(i)) ∧ isender(ie) ≠ Environment ⇒
    (∃! i2 : Inst_Name •
      i2 ∈ **dom** chart ∧ isender(ie) = mk_Address(i2) ∧
      (∃! oe : Event •
        oe ∈ outputEvents(chart(i2)) ∧ dest(oe) = mk_Address(i) ∧
        outmsgid(oe) = inmsgid(ie) ∧
        event_loc_temp(chart(i), ie) = event_loc_temp(chart(i2), oe)))) ∧
  (∀ oe : Event •
    oe ∈ outputEvents(chart(i)) ∧ dest(oe) ≠ Environment ⇒
    (∃! i2 : Inst_Name •
      i2 ∈ **dom** chart ∧ dest(oe) = mk_Address(i2) ∧
      (∃! ie : Event •
        ie ∈ inputEvents(chart(i2)) ∧ isender(ie) = mk_Address(i) ∧
        inmsgid(ie) = outmsgid(oe) ∧
        event_loc_temp(chart(i2), ie) = event_loc_temp(chart(i), oe))))),

3. The following wellformedness condition is to ensure that no deadlocks occur. We do this by checking that the transitive closure of the bidirectional connection graph is acyclic. property includes for example that a message output is not causally dependent on its corresponding message input, directly and indirectly trough other messages (se figure 6.14. However this property is not enough since messages are synchronous and thus will deadlock if the destination is not ready to accept the input. An example can be seen in figure 6.15, where the directed connection graph is not cyclic and the bidirectional is. Messages are a synchronization barrier, thus the order introduced is given by the synchronization points on the instances. As condition events also represent synchronization points they must also be considered. The order is established by creating the tuples of ID′s that happen in order, i.e. the tuples AB (Type Order) which denotes that the event with ID A happens before the event with ID B (on an instance). Now by ensuring that this order is not cyclic, the desired property is achieved. Even though subcharts are synchronization points as well, they must not be considered, since messages, conditions etc. in subcharts are contained within the subchart and can thus not introduce a cyclic order due to the other welformedness conditions. If the specification was not to be translated, the above condition could easily be specified as follows:

wf_mess_cond_acyclic : Chart → Bool wf_mess_cond_acyclic(chart) is let orders = po_instances(chart) in ∼ (exists ol : Order* • (all j : Nat • j ∈ inds ol ⇒ ol(j) ∈ orders) ∧ (all i : Nat • i > 0 ∧ i < len ol ⇒ id2(ol(i)) = id1(ol(i + 1))) ∧ id1(ol(1)) = id2(ol(len ol))) end,

For all given orders on a chart this functions checks weather the undirected connection graph is acyclic or not by trying to traverse the IDs reachable from each order present. Sorting is actually not necessary, but done in order to speed up the check in the resulting C−code.

wf_mess_cond_acyclic : Chart → **Bool**
wf_mess_cond_acyclic(chart) ≡

```
    let
        orders = po_instances(chart),
        orderl = insertionSort(set_to_list(orders), ⟨⟩)
    in
            ∀ o : Order • o ∈ elems orderl ⇒
                acyclic({id1(o), id2(o)}, id2(o), orderl)
    end,
```

4. A shared condition must appear in each instance among those it is shared. Including that the share must only use instances that are present. The temperature of the corresponding locations of the condition events must have the same temperature. This defines that conditions must be consistent across instances.

```
wf_condition_share : Chart → Bool
wf_condition_share(chart) ≡
    (∀ i : Inst_Name •
            i ∈ dom chart ⇒
                (∀ ce : Event •
                    ce ∈ conditionEvents(chart(i)) ⇒
                        (∀ i′ : Inst_Name •
                                i′ ∈ ceshare(ce) ⇒
                                    i′ ∈ dom chart ∧
                                    ce ∈ elems conditionEvents(chart(i′)) ∧
                                    event_loc_temp(chart(i), ce) =
                                     event_loc_temp(chart(i′), ce)))),
```

5. Locations in subcharts are locations on the RSC chart and vice versa in the correct order. It checks recursively on the location*, since quantified recursion is not in the subset of RSL that is translatable. We do this in order to ease the wellformedness conditions regarding messages. This also allows for easy definition of the state, as it can be done using a pointer, identifying the exact position. The same applies for subcharts.

```
wf_subchart_locations : Chart → Bool
wf_subchart_locations(chart) ≡
    (∀ i : Inst_Name •
            i ∈ dom chart ⇒
            wf_subchart_locations(chart(i))),

wf_subchart_locations : Location* → Bool
wf_subchart_locations(locl) ≡
    if locl = ⟨⟩ then true
    else
        case event(hd locl) of
            mk_Subchart(_, _, _,_, slocl) →
                let scpos = eventposition(locl, event(hd locl), 1) in
                        loclmatch(locl, slocl, 1, scpos)
                end,
            _ → true
        end
        ∧ wf_subchart_locations(tl locl)
    end,
```

6. All subcharts must appear in the same order on instances that share more than one subchart with each other. Checked separately, since acyclic does not treat subchart beginnings/endings, due to before mentioned

restrictions on them.

```
wf_subchart_ordered : Chart → Bool
wf_subchart_ordered(chart) ≡
    (∀ i : Inst_Name •
          i ∈ dom chart ⇒
              wf_subchart_ordered(chart,i, chart(i))),
wf_subchart_ordered : Chart × Inst_Name × Location* → Bool
wf_subchart_ordered(chart, i,  locl) ≡
    if locl = ⟨⟩ then true
    else
       case event(hd locl) of
        mk_Subchart(_, _, _,_, slocl) →
            (∀ sc2 : Event •
                sc2 ∈ elems subchartEvents(locl)
                ∧ event(hd locl) ≠ sc2 ⇒
                (∀ i′ : Inst_Name • i′ ∈
                   scshare(event(hd locl))∧ i′ ≠ i
                   ∧ i′ ∈ scshare(sc2) ⇒
                   (∃ sc′ : Event • sc′ ∈
                       subchartEvents(chart(i′))
                       ∧ scid(event(hd locl)) = scid(sc′) ∧
                       (∃ sc2′ : Event • sc2′ ∈
                           subchartEvents(chart(i′)) ∧ scid(sc2′) =
                           scid(sc2) ∧
                           let
                               r = eventposition(chart(i′), sc′, 1),
                               s = eventposition(chart(i′), sc2′, 1)
                           in
                               r < s
                           end)))),
            _ → true
        end
        ∧ wf_subchart_ordered(chart, i, tl locl)
    end,
```

7. All instances defined in a subchart share must contain that named subchart and they must be present on RSC. The temperature of the corresponding locations must be the same. This ensures that a subchart is consistent across instances.

```
wf_subchart_coherent : Chart → Bool
wf_subchart_coherent(chart) ≡
   (∀ i : Inst_Name •
       i ∈ dom chart ⇒
        (∀ sc : Event •
            sc ∈ elems subchartEvents(chart(i)) ⇒
               (∀ i′ : Inst_Name •
                   i′ ∈ scshare(sc) ⇒
                    (i′ ∈ dom chart ∧
                       (∃ sc′ : Event •
                        sc′ ∈ elems subchartEvents(chart(i′)) ∧
                        scname(sc) = scname(sc′) ∧
                        scid(sc) = scid(sc′) ∧
                        scshare(sc) = scshare(sc′) ∧
                        mult(sc) = mult(sc) ∧
```

```
            event_loc_temp(chart(i), sc) =
              event_loc_temp(chart(i′), sc′))))))),
```

8. A subchart has one endsubchart token after all the subchart locations. The temperature of the two corresponding locations must be the same. This is included in order to be able to tell when a given subchart ends, which is useful when it is executable.

```
wf_subchart_end : Chart → Bool
wf_subchart_end(chart) ≡
  (∀ i : Inst_Name •
    i ∈ dom chart ⇒
      (∀ sc : Event •
        sc ∈ elems subchartEvents(chart(i)) ⇒
          (∃! esc : Event •
              esc ∈ endsubchartEvents(chart(i))
              ∧ scid(sc) = escid(esc) ∧
              (let
                  scstart = eventposition(chart(i), sc, 1),
                  scend = eventposition(chart(i), esc, 1),
                  diff = len sclocl(sc)
               in
                  scstart + diff + 1 = scend
               end) ∧ event_loc_temp(chart(i), sc)
                        = event_loc_temp(chart(i), esc)))),
```

9. Conditions in subcharts must be contained within the subchart. This means a condition may not ″`cross`″ the boundaries of a subchart.

```
wf_subchart_conditions : Chart → Bool
wf_subchart_conditions(chart) ≡
  (∀ i : Inst_Name •
    i ∈ dom chart ⇒
      (∀ sc : Event •
        sc ∈ subchartEvents(chart(i)) ⇒
         (∀ ce : Event •
            ce ∈ conditionEvents(sclocl(sc)) ⇒
              (∀ i′ : Inst_Name •
                i′ ∈ ceshare(ce) ⇒
                 (∃ sc′ : Event •
                    sc′ ∈ elems subchartEvents(chart(i′))
                    ∧ scid(sc) = scid(sc′) ∧
                    (∃! ce′ : Event •
                       ce′ ∈ conditionEvents(sclocl(sc′))
                       ∧ cid(ce) = cid(ce′)))))))),
```

10. Messages emanating/ending in a subchart must have their corresponding message output/input in subchart. Same argument as above.

```
wf_subchart_messages : Chart → Bool
wf_subchart_messages(chart) ≡
  (∀ i : Inst_Name •
```

```
i ∈ dom chart ⇒
(∀ sc : Event •
    sc ∈ elems subchartEvents(chart(i)) ⇒
    (∀ oe : Event •
        oe ∈ outputEvents(sclocl(sc))
        ∧ dest(oe) ≠ Environment ⇒
        (∃ i′ : Inst_Name •
            i′ ∈ dom chart ∧
            (∃ sc′ : Event •
                sc′ ∈ subchartEvents(chart(i′))
                ∧ scid(sc) = scid(sc′) ∧
                    mk_InputEvent(outmsgid(oe), mk_Address(i)) ∈
                    elems inputEvents(sclocl(sc′))))) ∧
(∀ ie : Event •
    ie ∈ inputEvents(sclocl(sc))
    ∧ isender(ie) ≠ Environment ⇒
    (∃ i′ : Inst_Name •
        i′ ∈ dom chart ∧
        (∃ sc′ : Event •
            sc′ ∈ subchartEvents(chart(i′))
            ∧ scid(sc) = scid(sc′) ∧
            (∃ oe : Event •
                oe ∈ outputEvents(sclocl(sc′))
                ∧ inmsgid(ie) = outmsgid(oe) ∧
                dest(oe) = mk_Address(i))))))),
```

11. Subcharts starting in a subchart must also end in the same subchart. Sub−subcharts may also only use instances covered by the main−subchart. Same argument as above.

```
wf_subchart_subchart : Chart → Bool
wf_subchart_subchart(chart) ≡
    (∀ i : Inst_Name •
        i ∈ dom chart ⇒
            (∀ sc : Event •
                sc ∈ elems subchartEvents(chart(i)) ⇒
                (∀ sc′ : Event •
                    sc′ ∈ subchartEvents(sclocl(sc)) ⇒
                        (∃ esc′ : Event •
                            esc′ ∈ endsubchartEvents(sclocl(sc))
                            ∧ scid(sc′) = escid(esc′))
                                    ∧
                        (∀ i′ : Inst_Name •
                            i′ ∈ scshare(sc′) ⇒ i′ ∈ scshare(sc))))),
```

12. All coregion locations are locations on RSC chart. Most of the syntax′es we encounter handles this differently.See nr.5.

```
wf_coregion_locations : Chart → Bool
wf_coregion_locations(chart) ≡
    (∀ i : Inst_Name •
        i ∈ dom chart ⇒
            (∀ cr : Event •
                cr ∈ elems coregionEvents(chart(i)) ⇒
                (∀ l : Location •
```

```
        l ∈ crlocl(cr) ⇒
          l ∈ chart(i) ∧
            let crpos = eventposition(chart(i), cr, 1) in
              loclmatch(chart(i), crlocl(cr), 1, crpos)
            end))),
```

The last part of the above wellformedness condition could very easily be written as follows if it where not to be translatable. (exists s : Location*, t : Location* • (s ⌢ crlocl(cr)⌢ t = chart(i)))

13. All events in coregions are message events, as they are only defined with messages.

```
wf_coregion_messages : Chart → Bool
wf_coregion_messages(chart) ≡
  (∀ i : Inst_Name •
    i ∈ dom chart ⇒
      (∀ cr : Event •
        cr ∈ elems coregionEvents(chart(i)) ⇒
        (∀ l : Location •
          l ∈ elems crlocl(cr) ⇒
            (event(l) ∈ inputEvents(chart(i))
            ∨ event(l) ∈ outputEvents(chart(i))))))),
```

14. All locations after a cold location in a subchart are cold. For sub−subcharts the end subchart token may be an exception, which is same temperature as sub−subchart start event. This is to ensure that the ordering of hot and cold locations in subcharts is correct.

```
wf_cold_subchart : Chart → Bool
wf_cold_subchart(chart) ≡
  (∀ i : Inst_Name •
    i ∈ dom chart ⇒
      (∀ sc : Event •
        sc ∈ elems subchartEvents(chart(i)) ⇒
        (∀ j : Int •
          j ∈ inds sclocl(sc) ∧ temp(sclocl(sc)(j)) = Cold ⇒
          (∀ j′ : Int •
            j′ ∈ inds sclocl(sc) ∧ j′ > j ⇒
            (temp(sclocl(sc)(j′)) = Cold)
              ∨ (∃ sc′ : Event •
                  sc′ ∈ subchartEvents(sclocl(sc)) ∧
                  sclocl(sc)(j′) ∈ sclocl(sc′))
              ∨ (case event(sclocl(sc)(j′)) of
                    mk_EndSubchart(_) → true,
                    _ → false
                  end))))),
```

15. All locations after a cold location in a RSC are cold. Except for cold locations in subcharts, which may be followed by hot locations on surrounding (main− or sub−) chart. Subcharts are checked in wf nr. 14, same argument as above. In three parts due to some bug in the C♯ translator.

```
wf_cold_mainchart : Chart → Bool
```

```
wf_cold_mainchart(chart) ≡
    (∀ i : Inst_Name •
        i ∈ dom chart ⇒
            (∀ j : Int •
                j ∈ inds chart(i) ∧ temp(chart(i)(j)) = Cold ⇒
(

(∀ k : Int • k ∈ inds chart(i) ∧
 k > j ⇒ (temp(chart(i)(k)) = Cold))
∨
                    (case event(chart(i)(j)) of
                        mk_EndSubchart(_) → true,
                        _→ false
  end)
                ∨
            (∃ sc : Event •
                    sc ∈ subchartEvents(chart(i)) ∧ chart(i)(j)
                            ∈ elems sclocl(sc))


)))
,
```

16. The last event is a stop event. This is included exactly as in [26]. Initially in [7] a stop event as only used on created messages. It is however convenient when creating an executable version to have a stop event.

```
wf_last : Chart → Bool
wf_last(chart) ≡
    (∀ i : Inst_Name •
            i ∈ dom chart ⇒
                (event(chart(i)(len chart(i))) = StopEvent)),
```

17. A created instance may not be present in prechart as it is created in the main chart. In [7] a creation had a special arrow denoting creation. This has been omitted since the semantics of this arrow matched the semantics of a message. Therefore the first event must be a message. The list of creations in a RSC may be used to graphically determine which instances are created.

```
wf_creation: RSC′ → Bool
wf_creation(rsc) ≡
(∀ iname : Inst_Name • iname ∈ creations(rsc) ⇒
iname ∉ dom prechart(rsc)
∧ iname ∈ dom mainchart(rsc) ∧
        case event(mainchart(rsc)(iname)(1)) of
            mk_InputEvent(_,_) → true,
            _ → false
        end),
```

18. All conditions on a prechart must be hot. This is due to the fact that a cold condition exit is normally an accepted exit from a chart. This is not the case in precharts, where is form of acceptable exit is not wanted [7].

```
wf_prechart_condition: Chart → Bool
wf_prechart_condition(chart) ≡
(∀ iname : Inst_Name • iname ∈ dom chart ⇒
        (∀ ce : Event • ce ∈ conditionEvents(chart(iname)) ⇒
            (cetemp(ce) = Hot))),


-——————————————————————————————————-
-————————————- AUXILIARY FUNCTIONS ——————————————
-——————————————————————————————————-
```

po_instances finds the order of messages and conditions on instances of a chart.

```
po_instances : Chart → Order-set
po_instances(chart) ≡
    if chart = [ ] then {}
    else let i = hd chart in po_instance(chart(i), {}) ∪ po_instances(chart \ {i}) end
    end,
```

po_instance finds the order of messages and conditions on an instance (i.e. Location*). It creates all the orders introduced by the instance by creating tuples of ID, e.g. AB which denotes that the message or condition with ID A happens before the message or condition with ID B. It calls itself recursively in order to treat each Location depending on its event. It maintains a set of ID$'$s (idset) which holds the IDs that happen before the location at the beginning of the current Location*.

```
po_instance : Location* × ID-set → Order-set
po_instance(locl, idset) ≡
   if locl = ⟨⟩ then {}
   else
    case event(hd locl) of
```

Messages and condition events have their ID extracted and an Order-set is created by using the extracted ID and the preceeding ids given by idset. It proceeds with the rest of the list and adds the extracted ids to idset, which now holds the ID$'$s of messages and conditions that have happened.

```
        mk_InputEvent(inmsgid, _) →
            append(idset, {inmsgid})
            ∪
            po_instance(tl locl, idset ∪ {inmsgid}),
        mk_OutputEvent(outmsgid, _, _) →
            append(idset, {outmsgid})
            ∪
            po_instance(tl locl, idset ∪ {outmsgid}),
        mk_ConditionEvent(_, cid, _, _) →
            append(idset, {cid})
            ∪
            po_instance(tl locl, idset ∪ {cid}),
```

Coregionevents do not introduce ordering among themselves, only in relation to events happening before and after the coregion. All the ids of messages in coregion are extracted. Since a coregionevent spans more than one location all the locations of the coregionevent are removed in order to proceed.

```
        mk_CoregionEvent(clocl) →
            let cids = coregion_ids(clocl), newlocl = reduce_list((len clocl) + 1, locl) in
                append(idset, cids) ∪ po_instance(newlocl, idset ∪ cids)
            end,
```

All other events are not relevant for ordering. Not even subcharts, since there are well formedness conditions that guarantee that messages and conditions contained in subcharts do solely occur inside them.

```
        _ → po_instance(tl locl, idset)
    end
end,
```

Given a set of already traversed IDs, the current last traversed ID and a list of orders, acyclic checks whether the connection graph is acyclic or not. This is done by finding the next possible orders given the current last traversed ID.

```
acyclic : Int-set × Int × Order* → Bool
acyclic(seenIDs, lastid, orderl) ≡
    let fitting = extractfitting(lastid, orderl) in acyclic_fit(seenIDs, fitting, orderl) end,
```

Given a set of already traversed ID′s, a list of possible next orders and a list of all orders, acyclic_fit checks weather the connection graph is acyclic or not.

```
acyclic_fit : Int-set × Order* × Order* → Bool
acyclic_fit(seenIDs, fitting, orderl) ≡
    case fitting of
        ⟨⟩ → true,
        ⟨a⟩ ⌢ ⟨⟩ → id2(a) ∉ seenIDs
    ∧ acyclic(seenIDs ∪ {id2(a)}, id2(a), orderl),
        ⟨a⟩ ⌢ b →
            id2(a) ∉ seenIDs
    ∧ acyclic(seenIDs ∪ {id2(a)}, id2(a), orderl)
    ∧ acyclic_fit(seenIDs, b, orderl)
    end,
```

extractfitting extracts the orders of a list (as a list) with a given id as id1.

```
extractfitting : Int × Order* → Order*
extractfitting(id, orderl) ≡
    case orderl of
        ⟨⟩ → ⟨⟩,
        ⟨a⟩ ⌢ ⟨⟩ → if id1(a) = id then ⟨a⟩ else ⟨⟩ end,
        ⟨a⟩ ⌢ b →
            if id1(a) = id then ⟨a⟩ ⌢ extractfitting(id, b)
            else extractfitting(id, b) end
    end,
```

reduce_list reduces a Location* with i first elements.

```
reduce_list : Int × Location* ~→ Location*
reduce_list(i, locl) ≡ ⟨locl(j) | j in ⟨(i + 1) .. len locl⟩⟩
pre i ≤ len locl,
```

append creates an order-set of AB, where A is in the first argument set and B in the second. Uses two functions in order to recurse over the 2 sets. This could have been done very easily using a comprehended set, but that is unfortunately not translatable.

```
append : ID-set × ID-set → Order-set
append(idset1, idset2) ≡
    if idset1 = {} then {}
    else let i = hd idset1
in append_ID(i, idset2)
  ∪
  append(idset1 \ {i}, idset2)
end
    end,
```

append_ID creates Orders by using the first argument together with each element in the second argument.

```
append_ID : ID × ID-set → Order-set
append_ID(id, ids) ≡
    if ids = {} then {}
    else
        let i = hd ids
        in
            {mk_Order(id, i)}
            ∪
            append_ID(id, ids \ {i})
        end
    end,
```

eventlist extracts the list of events on an Instance.

```
eventlist : Location* → Event*
eventlist(ll) ≡ ⟨event(l) | l in ll⟩,
```

- -functions needed in order to make specification——
- ———— translatable————————————

onputEvents returns all the input events on a given Instance.

```
inputEvents : Location* → Event*
inputEvents(ll) ≡
    if ll = ⟨⟩ then ⟨⟩
    else
        case event(hd ll) of
            mk_InputEvent(_, _) → ⟨event(hd ll)⟩,
            _ → ⟨⟩
        end ⌢ inputEvents(tl ll)
```

```
    end,
```

outputEvents returns all the output events on a given Instance.

```
outputEvents : Location* → Event*
outputEvents(ll) ≡
    if ll = ⟨⟩ then ⟨⟩
    else
        case event(hd ll) of
            mk_OutputEvent(_, _, _) → ⟨event(hd ll)⟩,
            _ → ⟨⟩
        end ⌢ outputEvents(tl ll)
    end,
```

subchartEvents returns all the subchart events on a given Instance.

```
subchartEvents : Location* → Event*
subchartEvents(ll) ≡
    if ll = ⟨⟩ then ⟨⟩
    else
        case event(hd ll) of
            mk_Subchart(_, _, _, _, _) → ⟨event(hd ll)⟩,
            _ → ⟨⟩
        end ⌢ subchartEvents(tl ll)
    end,
```

endsubchartEvents returns all the endsubchart events on a given Instance.

```
endsubchartEvents : Location* → Event*
endsubchartEvents(ll) ≡
    if ll = ⟨⟩ then ⟨⟩
    else
        case event(hd ll) of
            mk_EndSubchart(_) → ⟨event(hd ll)⟩,
            _ → ⟨⟩
        end ⌢ endsubchartEvents(tl ll)
    end,
```

conditionEvents returns all the condition events on a given Instance.

```
conditionEvents : Location* → Event*
conditionEvents(ll) ≡
    if ll = ⟨⟩ then ⟨⟩
    else
        case event(hd ll) of
            mk_ConditionEvent(_, _, _, _) → ⟨event(hd ll)⟩,
            _ → ⟨⟩
        end ⌢ conditionEvents(tl ll)
    end,
```

coregionEvents returns all the coregion events on a given Instance.

```
coregionEvents : Location* → Event*
coregionEvents(ll) ≡
    if ll = ⟨⟩ then ⟨⟩
    else
        case event(hd ll) of
            mk_CoregionEvent(_) → ⟨event(hd ll)⟩,
            _ → ⟨⟩
        end ⌢ coregionEvents(tl ll)
    end,
```

coregion_locls extracts all the location*s used for coregions.

```
coregion_locls : Location* → (Location*)*
coregion_locls(ll) ≡
    if ll = ⟨⟩ then ⟨⟩
    else
        case event(hd ll) of
            mk_CoregionEvent(locl) → ⟨locl⟩,
            _ → ⟨⟩
        end ⌢ coregion_locls(tl ll)
    end,
```

coregion_ids extracts IDs used by messages in a given Location* which must only contain message−event.

```
coregion_ids : Location* ~→ ID-set
coregion_ids(locl) ≡
    if locl = ⟨⟩ then {}
    else
        case event(hd locl) of
            mk_InputEvent(inmsgid, _) →
                    {inmsgid} ∪ coregion_ids(tl locl),
            mk_OutputEvent(outmsgid, _, _) →
                    {outmsgid} ∪ coregion_ids(tl locl)
        end
    end
pre
    (∀ e : Event •
        e ∈ eventlist(locl) ⇒
            e ∈ inputEvents(locl) ∨ e ∈ outputEvents(locl)),
```

all_ids returns all the ids used in the chart.

```
all_ids : Chart → ID-set
all_ids(chart) ≡
    if chart = [ ] then {}
    else
```

```
        let i = hd chart
        in
            all_ids_instance(chart(i))
            ∪
            all_ids(chart \ {i})
        end
    end,
```

all_ids_instance returns all the ids from an instance.

```
all_ids_instance : Location* → ID-set
all_ids_instance(locl) ≡
    if locl = ⟨⟩ then {}
    else
        case event(hd locl) of
            mk_ActionEvent(_, id) → {id},
            mk_ConditionEvent(_, id, _, _) → {id},
            mk_InputEvent(id, addr) → {id},
            mk_OutputEvent(id, _, addr) → {id},
            mk_Subchart(_, id, _, _, _) → {id},
            _ → {}
        end ∪ all_ids_instance(tl locl)
    end,
```

all_addresses returns a set of all adresses possible in the chart, which is given by the instance names.

```
all_addresses : Chart → Address-set
all_addresses(chart) ≡
    if chart = [ ] then {Environment}
    else
        let i = hd chart
        in
            {mk_Address(i)}
            ∪
            all_addresses(chart \ {i})
        end
    end,
```

all_temp returns a set with all the temperatures.

```
all_temp : HotCold-set = {Hot, Cold},
```

all_outpids returns a set with all the message names of a chart.

```
all_outpids : Chart → Msg_Name-set
all_outpids(chart) ≡
    if chart = [ ] then {}
    else
        let i = hd chart
```

```
        in
            all_outpids_instance(chart(i))
            ∪
            all_outpids(chart \ {i})
        end
    end,
```

all_outpids_instance returns a set with all the message names of an instance.

```
all_outpids_instance : Location* → Msg_Name-set
all_outpids_instance(locl) ≡
    if locl = ⟨⟩ then {}
    else
        (case event(hd locl) of
              mk_OutputEvent(_, outpid, _) → {outpid},
              _ → {}
         end) ∪ all_outpids_instance(tl locl)
    end,
```

all_sc_names returns a set with all the subchart names of an instance.

```
all_sc_names : Location* → Subchart_Name-set
all_sc_names(locl) ≡
    if locl = ⟨⟩ then {}
    else
        case event(hd locl) of
            mk_Subchart(scname, _, _, _, _) → {scname},
            _ → {}
        end ∪ all_sc_names(tl locl)
    end,
```

loclmatch checks whether a Location* is a part of another Location*. If−constructs, albeit not necessary, are included for better understanding.

```
loclmatch : Location* × Location* × Nat × Nat → Bool
loclmatch(locl, crlocl, currpos, crpos) ≡
    - - complete list matched
    if currpos > len crlocl then true
    else
        - - end of locl reached
        if currpos+ crpos > len locl then false
        else
            - - keep looking
            (locl(crpos + currpos) = crlocl(currpos)) ∧
            loclmatch(locl, crlocl, currpos + 1, crpos)
            end
    end ,
```

extract_idps_chart extracts ids from a chart. Recurses over the map domain.

```
extract_idps_chart : Chart → IDp*
extract_idps_chart(chart) ≡
    if chart = [ ] then ⟨⟩
    else
        let i = hd chart in extract_idps_instance(chart(i)) ⌢ extract_idps_chart(chart \ {i}) end
    end,
```

extract_idps_instance extracts ids from a Location list and IDp′s for relevant events. Fx. the occurence of a message ID which message does not originate or end in environment must be 2.

```
extract_idps_instance : Location* → IDp*
extract_idps_instance(locl) ≡
    if locl = ⟨⟩ then ⟨⟩
    else
        (case event(hd locl) of
                mk_ConditionEvent(_, id, _, ceshare) →
  ⟨mk_IDp(id, card ceshare)⟩,
                mk_InputEvent(id, addr) →
                    if addr ≠ Environment then ⟨mk_IDp(id, 2)⟩
 else ⟨mk_IDp(id, 1)⟩ end,
                mk_OutputEvent(id, _, addr) →
                    if addr ≠ Environment then ⟨mk_IDp(id, 2)⟩
 else ⟨mk_IDp(id, 1)⟩ end,
                mk_Subchart(_, id, scshare, _, _) →
 ⟨mk_IDp(id, card scshare)⟩,
                mk_ActionEvent(_,id) → ⟨mk_IDp(id, 1)⟩,
                _ → ⟨⟩
            end) ⌢ extract_idps_instance(tl locl)
    end,
```

count_ids counts the number of occurences of an ID in an given list with IDp′s.

```
count_ids : IDp* × ID × Nat → Nat
count_ids(idplist, id, count) ≡
    if idplist = ⟨⟩ then count
    else
        if id(hd idplist) = id then count_ids(tl idplist, id, count + 1)
        else count_ids(tl idplist, id, count)
        end
    end,
```

event_loc−temp determines the temperature of the location of a given event.

```
event_loc_temp : Location* × Event → HotCold
event_loc_temp(locl, event) ≡
let ind = eventposition(locl, event, 1)
in temp(locl(ind)) end,
```

eventposition determines which indice a given event′s location has on a Location*.

```
eventposition : Location* × Event × Nat ~→ Nat
eventposition(locl, event, counter) ≡
    if event(locl(counter)) = event then counter
    else eventposition(locl, event, counter + 1) end
pre event ∈ elems eventlist(locl),
```

set_to_list transforms an Order-set to an Order*.

```
set_to_list : Order-set → Order*
set_to_list(os) ≡
    if os = {} then ⟨⟩
    else let e = hd os in ⟨e⟩ ⌢ set_to_list(os \ {e}) end end,
```

insertionSort sorts a list of orders. Modified version of insertionsort. It sorts first on id1 of the orders, then on id2. 2 Lists are used, one for the unsorted and one for the sorted elements in order to make an easier algorithm. Insertion−sort chosen since it is the fastest of the o(n2) sorting algorithms (only small amounts are to be ordered) and very easy to write.

```
insertionSort : Order* × Order* → Order*
insertionSort(unsorted, sorted) ≡
    case unsorted of
        ⟨⟩ → ⟨⟩,
        ⟨a⟩ ⌢ ⟨⟩ → insert(sorted, ⟨⟩, a),
        ⟨a⟩ ⌢ b → insertionSort(b, insert(sorted, ⟨⟩, a))
    end,

insert : Order* × Order* × Order → Order*
insert(unpassed, passed, number) ≡
    if unpassed = ⟨⟩ then passed ⌢ ⟨number⟩
    else
        if id1(hd unpassed) > id1(number)
              then passed ⌢ ⟨number⟩ ⌢ unpassed
        else
            if id1(hd unpassed) = id1(number)
            then
                if id2(hd unpassed) ≥ id2(number)
  then passed ⌢ ⟨number⟩ ⌢ unpassed
                else insert(tl unpassed, passed ⌢ ⟨hd unpassed⟩, number)
                end
            else insert(tl unpassed, passed ⌢ ⟨hd unpassed⟩, number)
            end
        end
    end

end - -class end
```

D.2 RSC semantics for one chart

D.2.1 RSC_semantics.rsl

RSC_wf

scheme RSC_semantics =
extend RSC_wf **with**
class
type

New types introduced in order to capture information about state and enabled events.

Position info gives information about the current state of one instance.

PosInfo :: pointer : **Nat** $\leftrightarrow$ incr aux : AuxInfo iteration :
Int* $\leftrightarrow$ alter,

Holds information about an instances state if it is currently in a coregion.

AuxInfo == None | CoRegion(idset : ID-**set**),

The state of a chart maps from instance names to a position info.

State = Inst_Name $\overrightarrow{m}$ PosInfo,

A trace is a list of states.

Trace = State*,

Traces are sets of a trace.

Traces = Trace-**set**,

EnableEvents are used in order to describe which events are currently enabled. The type depends on which kind of enabled event it is.

EnabledEvent ==

Enabled message. Instances as sets, as they may be empty due to messages from/to environment.

EnabledMessage(Inst_Name-**set**, Inst_Name-**set**, ID) |

Enabled (true) condition.

EnabledCondition(Inst_Name-**set**, Cond_Name, ID) |

Enabled action.

EnabledAction(Inst_Name, Action_Name, ID) |

Enabled message in coregion.

EnabledCoregion(Inst_Name-**set**, Inst_Name-**set**, ID, **Int**) |

Enabled subchart entry.

EnabledEnterSubchart(Inst_Name-**set**, ID, **Int**) |

Enabled subchart end.

EnabledEndSubchart(ID, Inst_Name-**set**) |

Enabled cold (false) condition exit from a subchart. First set are instances in subchart, next set are instances sharing condition.

EnabledExitSubchart(Inst_Name-**set**, ID, Inst_Name-**set**, Cond_Name) |

Enabled cold (false) condition exit from a mainchart.

EnabledExitMainchart(Inst_Name-**set**, Cond_Name) |

Denotes that an instance has stopped.

EnabledStopped |

Used when an instance has no enabled event.

NotEnabled
value

semantics gives all the possible traces for a given chart.

semantics : Chart → Traces
semantics(chart) ≡ eval(chart, {⟨initialize_chart(chart)⟩}),

eval recursively evaluates traces until no new are found.

```
eval : Chart × Traces → Traces
eval(chart, ts) ≡
    let ts′ = eval_traces(chart, ts) in
        if ts′ = ts then ts else eval(chart, ts′) end
    end,
```

eval_traces evaluates traces. There is a need for an extra function as recursively defined quantified expressions are not in the subset of RSL that is translatable to C++.

```
eval_traces : Chart × Traces → Traces
eval_traces(chart, ts) ≡
    --converting -set-set to -set
    convert_tss2s({eval_trace(chart, t) | t : Trace • t ∈ ts}),
```

eval_trace evaluates a trace by finding the next possible states given the current state, and creating the resulting new traces.

```
eval_trace : Chart × Trace ~→ Traces
eval_trace(chart, t) ≡
    let evalss = eval_state(chart, hd t) in
        if evalss = {} ∨ (∀ ns : State • ns ∈ evalss ⇒ ns =
        hd t ∨ (if len t > 1 then hd t = (hd (tl t)) else false end))
        then {t}
        else
            -- some of the instances may be stopped, they are not added
            -- since check ns ~= hd t
            if (∃ ns : State • ns ∈ evalss ∧ ns = [ ])
            then
                {⟨ns⟩ ⌢ t | ns : State • ns ∈ evalss ∧ ns ≠ hd t ∧ ns ≠ [ ]} ∪
                {⟨hd t⟩ ⌢ t}
            else {⟨ns⟩ ⌢ t | ns : State • ns ∈ evalss ∧ ns ≠ hd t}
            end
        end
    end
    pre t ≠ ⟨⟩ ,
```

eval_state: given a chart and a stats the enabled events are found and the corresponding next states are found.

```
eval_state : Chart × State → State-set
eval_state(chart, curState) ≡
    {step_event(chart, x, curState) | x : EnabledEvent • x ∈ get_enabled_events(chart, curState)},
```

step_event takes a step with an enabled event and returns the resulting state..

```
step_event : Chart × EnabledEvent × State → State
step_event(chart, eevent, curState) ≡
    case (eevent) of
        -- Not an enabled event.
```

```
NotEnabled → curState,
- - Instance has stopped
EnabledStopped → curState,
- - main chart is exited
EnabledExitMainchart(_, _) → end_state(chart),
- - exit from subchart: all instances in subchart are advanced
- - to next endsubchart token
EnabledExitSubchart(inames, _, _, _) →
    curState †
    [ iname ↦ next_position_exitsc(chart, curState, iname) |
       iname : Inst_Name • iname ∈ inames ],
- - simple proceed for for example messages
EnabledAction(iname,_,_) →
    curState †
    [ iname ↦ next_position(chart, curState, iname) ],
EnabledMessage(inames, inames2, _) →
    curState †
    [ iname ↦ next_position(chart, curState, iname) | iname :
    Inst_Name • iname ∈ (inames ∪ inames2) ],
EnabledCondition(inames, _, _) →
    curState †
    [ iname ↦ next_position(chart, curState, iname) | iname :
    Inst_Name • iname ∈ inames ],
EnabledEnterSubchart(inames, _, iteration) →
    curState †
    [ iname ↦ next_position_entersc(chart, curState, iname, iteration) | iname :
    Inst_Name • iname ∈ inames ],
- - a complete subchart has finished and may iterate depending on multiplicity
EnabledEndSubchart(scid, inames) →
    (if (repeat_subchart(curState, inames))
       then
          curState †
          [ iname ↦ next_position_subback(chart, curState, iname, scid) |
           iname : Inst_Name • iname ∈ inames ]
       else
          curState †
          [ iname ↦ next_position_subend(chart, curState, iname) |
           iname : Inst_Name • iname ∈ inames ]
       end),
- - event involving coregion messages, 12 types of
- - different situations:
- -Type Origin Destination last event in coregion
- - 1 Coregion Instance no
- - 2 Coregion Instance yes
- - 3 Instance Coregion no
- - 4 Instance Coregion yes
- - 5 Coregion Coregion no, no
- - 6 Coregion Coregion yes, no
- - 7 Coregion Coregion no, yes
- - 8 Coregion Coregion yes, yes
- - 9 Coregion Environment no
- -10 Coregion Environment yes
- -11 Environment Coregion no
- -12 Environment Coregion yes

EnabledCoregion(iname, iname2, msgid, typeevent) →
    case typeevent of
       1 →
```

```
- - coregion origin (not last event), instance destination
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, false),
      hd iname2 ↦ next_position(chart, curState, hd iname2) ],
2 →
- - coregion origin (last event), instance destination
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, true),
      hd iname2 ↦ next_position(chart, curState, hd iname2) ],
3 →
- - instance origin, coregion destination (not last event)
    curState †
    [ hd iname ↦ next_position(chart, curState, hd iname),
      hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, false) ],
4 →
- - instance origin, coregion destination (last event)
    curState †
    [ hd iname ↦ next_position(chart, curState, hd iname),
      hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, true) ],
5 →
- - coregion origin (not last event), coregion destination
- - (not last event)
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, false),
      hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, false) ],
6 →
- - coregion origin (last event), coregion destination
- - (not last event)
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, true),
      hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, false) ],
7 →
- - coregion origin (not last event), coregion destination
- - (last event)
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, false),
      hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, true) ],
8 →
- - coregion origin (last event), coregion destination
- - (lastevent)
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, true),
      hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, true) ],
9 →
- - coregion origin (not last event), Environment destination
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, false) ],
10 →
- - coregion origin (last event), Environment destination
    curState †
    [ hd iname ↦ next_position_cor(chart, curState, hd iname, msgid, true) ],
11 →
- - Environment origin, Coregion destination (not last event)
    curState †
    [ hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, false) ],
12 →
- - Environment origin, Coregion destination (last event)
    curState †
```

```
                [ hd iname2 ↦ next_position_cor(chart, curState, hd iname2, msgid, true) ]
        end
    end,
```

next_position finds the next positioninfo for a state. Used for most enabled events.

```
next_position : Chart × State × Inst_Name → PosInfo
next_position(chart, curState, iname) ≡
    let pi = curState(iname), nextEvent = eventlist(chart(iname))(pointer(pi) + 1) in
        create_positioninfo(chart, nextEvent, pi, 1)
    end,
```

next_position_entersc finds the next positioninfo for an instance entering a subchart.

```
next_position_entersc : Chart × State × Inst_Name × Int → PosInfo
next_position_entersc(chart, curState, iname, iteration) ≡
    let pi = curState(iname), nextEvent = eventlist(chart(iname))(pointer(pi) + 1) in
        create_positioninfo(chart, nextEvent, alter(⟨iteration⟩^(iteration(pi)),pi), 1)
    end,
```

next_position_subend finds the next positioninfo for an instance leaving a subchart.

```
next_position_subend : Chart × State × Inst_Name → PosInfo
next_position_subend(chart, curState, iname) ≡
    let
        pi = curState(iname),
        nextEvent = eventlist(chart(iname))(pointer(pi) + 1)
    in
        create_positioninfo(chart, nextEvent, alter(tl iteration(pi),pi), 1)
    end,
```

next_position_exitsc finds the next positioninfo when the current location denotes a subchart exit. This is necessary since the next event position is not yet known.

```
next_position_exitsc : Chart × State × Inst_Name → PosInfo
next_position_exitsc(chart, curState, iname) ≡
    let
        oldpointer = pointer(curState(iname)),
        newpointer = return_next_endsc(chart, curState, iname, pointer(curState(iname)))
    in
        -- creating positioninfo based on newpointer
        -- StopEvent provided in order to fall trough in
        -- create_positioninfo case construct, does not mean end of
        -- instance. +1 one in order to avoid endsubchart token
        create_positioninfo(chart, StopEvent, curState(iname), newpointer − oldpointer+1)
    end,
```

next_position_subback finds the next positioninfo when the current location denotes a repeating of a subchart. This is necessary since the next event (on subchart beginning) is not yet known.

```
next_position_subback : Chart × State × Inst_Name × ID → PosInfo
next_position_subback(chart, curState, iname, scid) ≡
    let
        pi = curState(iname),
        sc : Event • sc ∈ subchartEvents(chart(iname)) ∧ scid = scid(sc),
        length = len sclocl(sc),
        oldptr = pointer(pi),
        nextEvent = eventlist(chart(iname))(oldptr−length)
    in
        -- creating new pi, event is subchart, -length since compensation
        --for all events in subchart, then subchart will directly be
        --enabled again
        create_positioninfo(chart, nextEvent, alter(⟨(hd iteration(pi))−1⟩⌢(tl iteration(pi)),pi), −(length))
    end,
```

next_position_cor finds the next positioninfo when the current location denotes a coregion.

```
next_position_cor : Chart × State × Inst_Name × ID × Bool → PosInfo
next_position_cor(chart, curState, iname, msgid, lastevent) ≡
    case lastevent of
        true →
            let
                pi = curState(iname),
                curEvent = eventlist(chart(iname))(pointer(pi)),
                diff = len crlocl(curEvent),
                nextEvent = eventlist(chart(iname))(pointer(pi) + diff +
                1),
                newpi = mk_PosInfo(pointer(pi), None, iteration(pi))
            in
                create_positioninfo(chart, nextEvent, newpi, diff + 1)
            end,
        false →
            let
                pi = curState(iname),
                oldai = aux(pi),
                oldmsgids  = idset(oldai),
                newmsgids = oldmsgids \ {msgid},
                ai = CoRegion(newmsgids),
                newpos = mk_PosInfo(pointer(pi), ai, iteration(pi))
            in
                newpos
            end
    end,
```

create_positioninfo creates the actual positioninfo.

```
create_positioninfo : Chart × Event × PosInfo × Int → PosInfo
create_positioninfo(chart, nextEvent, pi, diff) ≡
    case nextEvent of
        mk_CoregionEvent(crlocl) →
```

```
            let
                newmsgids = all_ids_instance(crlocl),
                ai = CoRegion(newmsgids), newpos =
                mk_PosInfo(pointer(pi) + diff, ai, iteration(pi))
            in
                newpos
            end,

        _ → incr(pointer(pi) + diff, pi)
    end,
```

get_enabled_events finds the enabled events in a given state.

```
get_enabled_events : Chart × State → EnabledEvent-set
get_enabled_events(chart, curState) ≡
    let inames = dom curState in
        convert_ees2es(
                {get_enabled_event(chart, curState, iname) | iname : Inst_Name • iname ∈ inames})
    end,
```

Same as above, splitted for convenience.

```
get_enabled_event : Chart × State × Inst_Name → EnabledEvent-set
get_enabled_event(chart, curState, iname) ≡
    let pi = curState(iname), curEvent = eventlist(chart(iname))(pointer(pi)) in
        get_enabled_event_cur(chart, curState, iname, curEvent)
    end,
```

The following part deals with identifying enabled events.

get_enabled_event_cur gets the enabled events depending on the current event.

```
get_enabled_event_cur : Chart × State × Inst_Name × Event → EnabledEvent-set
get_enabled_event_cur(chart, curState, iname, curEvent) ≡
    case curEvent of
     mk_InputEvent(inmsgid, inaddr) → {get_ee_inputevent(chart,
     curState, iname, inmsgid, inaddr)},
     mk_OutputEvent(outmsgid, outpid, outaddr) →
         {get_ee_outputevent(chart, curState, iname, outmsgid, outaddr)},
     -- may branch on condition, therefore set
     mk_ConditionEvent(conname, cid, cetemp, share) →
         get_ee_con(chart, curState, iname,  curEvent),
     mk_CoregionEvent(locl) → get_ee_coregion(chart, curState, iname, locl),
     mk_Subchart(scname, scid, inames, mult, sclocl) →
         {get_ee_subchart(chart, curState, curEvent)},
     mk_EndSubchart(scid) → {get_ee_endsc(chart, curState, scid)},
     StopEvent → {EnabledStopped},
     mk_ActionEvent(aname,id) → {EnabledAction(iname,aname, id)}
    end,
```

get_ee_inputevent finds enabled events on an input event if the message comes from Environment. Otherwise outputevents will check if the corresponding input event is ready.

```
get_ee_inputevent : Chart × State × Inst_Name × ID × Address → EnabledEvent
get_ee_inputevent(chart, curState, iname, inmsgid, inaddr) ≡
if inaddr = Environment
then
    EnabledMessage({}, {iname}, inmsgid)
else
    NotEnabled
end,
```

get_ee_outputevent finds enabled event if current is an output event.

```
get_ee_outputevent : Chart × State × Inst_Name × ID × Address → EnabledEvent
get_ee_outputevent(chart, curState, iname, outmsgid, outaddr) ≡
    -- destination is environment, instance may proceed
    if (outaddr = Environment)
    then
        EnabledMessage({iname}, {}, outmsgid)
    -- destination is other instance
    else
    (if
        -- destination is a simple message reception on instance
        (∃ iname2 : Inst_Name •
             iname2 ∈ dom curState ∧
             (∃ inaddr : Address •
                    inaddr ∈ all_addresses(chart)
 ∧ inaddr = mk_Address(iname) ∧
                    eventlist(chart(iname2))(pointer(curState(iname2))) =
                        mk_InputEvent(outmsgid, inaddr)))
     then
        let
         inaddr = mk_Address(iname),
         iname2 : Inst_Name •
            iname2 ∈ dom curState ∧
            eventlist(chart(iname2))(pointer(curState(iname2))) =
             mk_InputEvent(outmsgid, inaddr) in
             EnabledMessage({iname}, {iname2}, outmsgid) end
    else
    (if
            -- destination is a message reception in coregion
            (∃ iname2 : Inst_Name • iname2 ∈ dom curState  ∧
                (∃ locl : Location* • locl ∈
                coregion_locls(chart(iname2)) ∧
                eventlist(chart(iname2))(pointer(curState(iname2)))
                = mk_CoregionEvent(locl)
∧ mk_InputEvent(outmsgid, mk_Address(iname))
 ∈ elems eventlist(locl)))
        then
         let
            inaddr = mk_Address(iname),
            iname2 : Inst_Name •
```

```
            iname2 ∈ dom curState ∧
            (∃ locl : Location* • locl ∈
            coregion_locls(chart(iname2)) ∧
                eventlist(chart(iname2))(pointer(curState(iname2))) =
  mk_CoregionEvent(locl) ∧
                mk_InputEvent(outmsgid, inaddr) ∈ elems eventlist(locl)),
        pi = curState(iname2),
        ai = aux(pi),
        oldmsgids = idset(ai),
        newmsgids = oldmsgids \ {outmsgid} in
        if (newmsgids = {})
        then  - - coregion: last event, type 4 transfer
            EnabledCoregion({iname}, {iname2}, outmsgid, 4)
        else  - - coregion: not last event, type 3 transfer
            EnabledCoregion({iname}, {iname2}, outmsgid, 3)
        end
      end
      - - message receiver not ready yet
      else NotEnabled end)
   end)
  end,
```

get_ee_con finds enabled event if current is a condtion event.

```
get_ee_con : Chart × State × Inst_Name × Event → EnabledEvent-set
get_ee_con(chart, curState, iname, curEvent) ≡
    if
        cetemp(curEvent) = Hot
    then
        - - hot condition, must be satisfied, may proceed
        {get_ee_con_satisfied(chart, curState, iname, curEvent)}
    else  - - cold condition, may evaluate to true or false
        get_ee_con_cold(chart, curState, iname, curEvent)
    end,
```

get_ee_con_cold finds enabled event if current is a cold condition event.

```
get_ee_con_cold :
    Chart × State × Inst_Name × Event → EnabledEvent-set
get_ee_con_cold(chart, curState, iname, curEvent) ≡
    if  - - determines if condition is in a subchart
                (∃ sc : Event •
                    sc ∈ subchartEvents(chart(iname)) ∧
                    curEvent ∈ elems eventlist(sclocl(sc)))
    then
        let
      - - event = eventlist(chart(iname))(pointer(curState(iname))),
            sc : Event • sc ∈ subchartEvents(chart(iname)) ∧
            curEvent ∈ elems eventlist(sclocl(sc)) ∧
            closest_subchart(chart(iname),sc, curEvent)
        in
            - - 2 possible next states: true and cold condition exit from
            - - subchart
            {get_ee_con_satisfied(chart, curState, iname, curEvent),
                EnabledExitSubchart(scshare(sc), cid(curEvent),
```

```
ceshare(curEvent), conname(curEvent))}
        end
    else  - - condition is not in subchart
    - - 2 possible next states: true and cold condition exit from
    - - main chart
        {get_ee_con_satisfied(chart, curState, iname, curEvent),
            EnabledExitMainchart(ceshare(curEvent), conname(curEvent))}
    end,
```

get_ee_con_satisfied finds enabled event if current is a true condition event.

```
get_ee_con_satisfied :
    Chart × State × Inst_Name × Event→ EnabledEvent
get_ee_con_satisfied(chart, curState, iname, curEvent) ≡
    if ceshare(curEvent) = {}
    then  - - condition not shared, instance may proceed
        EnabledCondition({iname}, conname(curEvent), cid(curEvent))
    else
    - - Checking if all instances are at condition.
        if
            (∀ iname2 : Inst_Name •
                    iname2 ∈ dom curState ∧ iname2 ∈ ceshare(curEvent) ⇒
                            eventlist(chart(iname2))(pointer(curState(iname2))) =
                                curEvent)
        then EnabledCondition(ceshare(curEvent), conname(curEvent), cid(curEvent))
        else NotEnabled
        end
    end,
```

get_ee_subchart finds enabled event if current is a subchart event.

```
get_ee_subchart :
    Chart × State × Event → EnabledEvent
get_ee_subchart(chart, curState, curEvent) ≡
    if
    (∀ iname2 : Inst_Name •
        iname2 ∈ scshare(curEvent) ⇒
        case eventlist(chart(iname2))(pointer(curState(iname2)))
        of
            mk_Subchart(scname2, scid2, scshare2, mult2, _) →
             scname2 = scname(curEvent) ∧ scid2 = scid(curEvent)
             ∧ scshare2 = scshare(curEvent) ∧ mult2 =
             mult(curEvent),
            _ → false
            end)
    then EnabledEnterSubchart(scshare(curEvent), scid(curEvent), mult(curEvent))
else NotEnabled
end,
```

get_ee_endsc finds enabled event if current is a end subchart event.

```
get_ee_endsc : Chart × State × ID → EnabledEvent
get_ee_endsc(chart, curState, scid) ≡
    let
     inames =
        {iname2 |
         iname2 : Inst_Name •
            iname2 ∈ dom curState
            ∧ mk_EndSubchart(scid) ∈ eventlist(chart(iname2))}
    in
     if - - checking if all instances are finished in subchart
        (∀ iname2 : Inst_Name •
            iname2 ∈ inames ⇒
             eventlist(chart(iname2))(pointer(curState(iname2))) =
                 mk_EndSubchart(scid))
     then
        EnabledEndSubchart(scid, inames) - - subchart complete
     else NotEnabled - - subchart not complete, may not proceed
     end
    end,
```

get_ee_coregion finds enabled event if current is a coregion event.

```
get_ee_coregion : Chart × State × Inst_Name × Location* → EnabledEvent-set
get_ee_coregion(chart, curState, iname, locl) ≡
    let
        pi = curState(iname),
        ai = aux(pi),
        msgids = idset(ai),
        outmsgids = outmsgids(locl) ∩ msgids,
        inmsgids = inmsgids(locl) ∩ msgids
    in
        {get_ee_coregion_output(chart, curState, iname, msgid) |
            msgid : ID • msgid ∈ outmsgids}
            ∪
        {get_ee_coregion_input(chart, curState, iname, msgid) |
            msgid : ID • msgid ∈ inmsgids}
    end,
```

get_ee_coregion_output finds enabled event if current event is a coregion output event.

```
get_ee_coregion_output : Chart × State × Inst_Name × ID → EnabledEvent
get_ee_coregion_output(chart, curState, iname, msgid) ≡
    let
     event = mk_InputEvent(msgid, mk_Address(iname)),
     iname2 : Inst_Name • iname2 ∈ dom curState ∧ event ∈ eventlist(chart(iname2))
    in
     - - check destination
     if
        (∃ locl : Location* •
        locl ∈ elems coregion_locls(chart(iname2)) ∧
        event ∈ elems eventlist(locl))
     then
        - - destination is coregion
        get_ee_coregion_coregion(chart, curState, iname, iname2, msgid)
```

```
    else
      if
          (∃ iname2 : Inst_Name • iname2 ∈ dom chart ∧ event
          ∈ elems eventlist(chart(iname2)))
      then
       -- destination is instance
       get_ee_coregion_instance(chart, curState, iname, iname2, msgid)
      else
       -- destination is Environment
       get_ee_coregion_environment(chart, curState, iname, msgid)
      end
    end
   end,
```

get_ee_coregion_input finds enabled event if current event is a coregion output event.

```
get_ee_coregion_input : Chart × State × Inst_Name × ID → EnabledEvent
get_ee_coregion_input(chart, curState, iname, msgid) ≡
if
   (∃ event : Event • event ∈ elems eventlist(chart(iname)) ∧ event =
   mk_InputEvent(msgid, Environment))
then
   -- input event from Environment
   get_ee_environment_coregion(chart, curState, iname, msgid)
else
   -- input from elsewhere, do nothing
   NotEnabled
end,
```

get_ee_environment_coregion finds enabled event if current event is inputevent in coregion from environment

```
get_ee_environment_coregion : Chart × State × Inst_Name × ID → EnabledEvent
get_ee_environment_coregion(chart, curState, iname, msgid) ≡
    if (get_rest_msgids(curState, iname, msgid) = {})
    then -- coregion: last event, type 12 transfer
        EnabledCoregion({}, {iname}, msgid, 12)
    else -- coregion: not last event, type 11 transfer
        EnabledCoregion({}, {iname}, msgid, 11)
    end,
```

get_ee_coregion_environment finds enabled event if current event is outputevent in coregion to environment

```
get_ee_coregion_environment : Chart × State × Inst_Name × ID → EnabledEvent
get_ee_coregion_environment(chart, curState, iname, msgid) ≡
    if (get_rest_msgids(curState, iname, msgid) = {})
    then -- coregion: last event, type 10 transfer
        EnabledCoregion({iname}, {}, msgid, 10)
    else -- coregion: not last event, type 9 transfer
        EnabledCoregion({iname}, {}, msgid, 9)
    end,
```

get_ee_coregion_instance finds enabled event if current is a coregion and the destination of the message event is an instance (not coregion).

```
get_ee_coregion_instance : Chart × State × Inst_Name × Inst_Name × ID → EnabledEvent
get_ee_coregion_instance(chart, curState, iname, iname2, msgid) ≡
    if
     (eventlist(chart(iname2))(pointer(curState(iname2))) =
         mk_InputEvent(msgid, mk_Address(iname)))
    then
     if (get_rest_msgids(curState, iname, msgid) = {})
     then  - - coregion: last event, type 2 transfer
         EnabledCoregion({iname}, {iname2}, msgid, 2)
     else  - - coregion: not last event, type 1 transfer
         EnabledCoregion({iname}, {iname2}, msgid, 1)
     end
    else  - - instance not ready to receive message
        NotEnabled
    end,
```

get_ee_coregion_coregion finds enabled event if current is a coregion and the destination of the message event is a coregion (not instance).

```
get_ee_coregion_coregion :
Chart × State × Inst_Name × Inst_Name × ID → EnabledEvent
get_ee_coregion_coregion(chart, curState, iname, iname2, msgid) ≡
    if
     (∃ event : Event •
         event ∈ eventlist(chart(iname2)) ∧
         (case eventlist(chart(iname2))(pointer(curState(iname2))) of
             mk_CoregionEvent(locl) →
                 event = mk_InputEvent(msgid, mk_Address(iname)) ∧
                 event ∈ elems eventlist(locl),
             _ → false
            end))
    then  - - coregion ready to receive
    - - origin has last event (type 6, 8)
     if (get_rest_msgids(curState, iname, msgid) = {})
     then
        if (get_rest_msgids(curState, iname2, msgid) = {})
        then  - -destination has last event (type 8)
            EnabledCoregion({iname}, {iname2}, msgid, 8)
        else  - -destination not last event (type 6)
            EnabledCoregion({iname}, {iname2}, msgid, 6)
        end
     else  - - origin not last event (type 5, 7)
        if (get_rest_msgids(curState, iname2, msgid) = {})
        then  - - destination has last event (type 7)
            EnabledCoregion({iname}, {iname2}, msgid, 7)
        else  - - destination not last event
            EnabledCoregion({iname}, {iname2}, msgid, 5)
        end
     end
    else  - - coregion not reached yet
        NotEnabled
    end,
```

repeat_subchart checks whether a subchart should be repeated or not.

```
repeat_subchart : State × Inst_Name-set → Bool
repeat_subchart(state, names) ≡
let
    name = hd names,
    posinfo = state(name),
    iter = iteration(posinfo)
in
    hd iter > 1
end,
```

initialize_chart returns the initial state a RSC is in.

```
initialize_chart : Chart → State
initialize_chart(chart) ≡
    step_event(chart, EnabledCondition(dom chart,////, 0), [ i ↦
    mk_PosInfo(0, None, ⟨⟩) | i : Inst_Name • i ∈ dom chart ]),
```

end_state calculates the end state a RSC is in when it has performed all possible events.

```
end_state : Chart → State
end_state(chart) ≡
    [ i ↦ mk_PosInfo(len chart(i), None, ⟨⟩) |
    i : Inst_Name • i ∈ dom chart ],
```

The following part are auxiliary functions needed for translation to C♯.

Converts a set of traces to traces with the same elements.

```
convert_tss2s : Traces-set → Traces
convert_tss2s(tss) ≡
    if tss = {} then {}
    else let element = hd tss in element ∪ convert_tss2s(tss \ {element}) end
    end,
- - would be much simpler if it should not be translatable:
- - t | t : Trace, ts : Traces :- t isin ts / ts isin tss,
```

convert_ees2es converts a set-set of enabled events to a set of enabled events with the same elements.

```
convert_ees2es : (EnabledEvent-set)-set → EnabledEvent-set
convert_ees2es(adss) ≡
    if adss = {} then {}
```

```
    else let element = hd adss in element ∪ convert_ees2es(adss \ {element}) end
    end,
```

get_rest_msgids removes an id from the set of ids that denote the messages not yet sent/received in a coregion.

```
get_rest_msgids : State × Inst_Name × ID → ID-set
get_rest_msgids(curState, iname, msgid) ≡
    let
        pi = curState(iname),
        ai = aux(pi),
        oldmsgids = idset(ai),
        newmsgids = oldmsgids \ {msgid}
    in
        newmsgids
    end,
```

closest_subchart determines if the given subchart is the ″innermost″ subchart of the event given.

```
closest_subchart : Location* × Event × Event → Bool
closest_subchart(inst, sc, event) ≡
    ∼(∃ sc2 : Event • sc2 ∈ elems subchartEvents(sclocl(sc)) ∧
                      event ∈ elems eventlist(sclocl(sc2))),
```

outmsgids returns the ids of all the outputevents of a location*.

```
outmsgids : Location* → ID-set
outmsgids(locl) ≡
if locl = ⟨⟩ then {} else
    case event(hd locl) of
        mk_OutputEvent(outmsgid,_,_) → {outmsgid},
        _ → {}
    end ∪ outmsgids(tl locl)
end,
```

inmsgids returns the ids of all the inputevents of a location*.

```
inmsgids : Location* → ID-set
inmsgids(locl) ≡
if locl = ⟨⟩ then {} else
    case event(hd locl) of
        mk_InputEvent(inmsgid, _) → {inmsgid},
        _ → {}
    end ∪ inmsgids(tl locl)
end,
```

return_next_endsc is an auxiliary function for finding the next endsubchart token given the current pointer.

```
return_next_endsc : Chart × State × Inst_Name × Int → Int
return_next_endsc(chart, curState, iname, pointer) ≡
    let curEvent = eventlist(chart(iname))(pointer) in
        if
            (case curEvent of
                    mk_EndSubchart(_) → true,
                    _ → false
                end)
        then pointer
        else return_next_endsc(chart, curState, iname, pointer + 1)
        end
    end

end - - class end
```

D.3 RSC collections

D.3.1 lsc/LSC_collection.rsl

RSC_semantics

scheme RSC_collection =
extend RSC_semantics **with**
class

The current version does not support reactivation. It only shows how to step through and perform all events once in a RSC collection.

- ————————- NEW TYPES FOR COLLECTIONS————————

Types for collecting information about events and their corresponding instance.

type

Ties together an event to an instance.

```
    EventInst :: eievent : Event   einame : Inst_Name,
```

A RSC in a collectin. The first denotes a finished RSC. The second an activated with the current state. The last denotes a not activated LSC along with the possible states of the prechart.

```
    ColRSC ==
        mk_done(doneRSC : RSC) |
        mk_act(actRSC : RSC, actState : State) |
        mk_not_act(notactRSC : RSC, posStates : State-set),
    Collection = Inst_Name  m→  ColRSC
```

value

```
- ———————— EXECUTION PART ————————————

Executes a given collection by performing one step. This is possible as the system is discrete.

execute_collection : Collection ~→ Collection
execute_collection(col) ≡
let

We must deterministically choose an active RSC that we will advance.

    curIname = choose_active_RSC(col),
    - - curRSC : RSC :- curRSC = actRSC(col(curIname)),
    - - gets the possible advances of the current RSC

On the chosen RSC we find the enabled events and choose one to perform.

    eevents = get_enabled_events(mainchart(actRSC(col(curIname))), actState(col(curIname))),
    eevent = choose_event(eevents),

The event may also be enabled in other RSC's. They must be advanced as well. It is checked which RSC's are
affected

    affected =
        [ iname ↦ col(iname) |
            iname : Inst_Name •
                iname ∈ dom col ∧
                (case col(iname) of
                    mk_not_act(actRSC, _) → is_visible(actRSC, eevent),
                    _ → false
                 end) ],

The event may be the event that finally fulfills a prechart. The corresponding mainchart must then activated.

    activated =
        [ iname ↦ col(iname) |
            iname : Inst_Name •
                iname ∈ dom affected ∧
                is_lsc_activated(notactRSC(col(iname)), posStates(col(iname)), eevent) ]
in
    col †

The currently executed RSC is updated by performing the chosen event.

    [ curIname ↦ execute_active(col(curIname), eevent) ] †

The activated RSC′s are updated by performing the chosen event. They now have an active mainchart.

    [ iname ↦ mk_act(notactRSC(col(iname)), initialize_chart(mainchart(notactRSC(col(iname))))) |
      iname : Inst_Name • iname ∈ dom activated ] †
```

The affected, not active RSC's are updated by performing the chosen event.

```
    [ iname ↦
        mk_not_act(
            notactRSC(col(iname)),
            possible_states(notactRSC(col(iname)), posStates(col(iname)), eevent)) |
        iname : Inst_Name • iname ∈ dom affected ∧ iname ∉ dom activated ]
end
pre active_present(col),
```

Executes the currently active RSC.

```
execute_active : ColRSC × EnabledEvent ~→ ColRSC
execute_active(colRSC, adv) ≡
let
    newstate = step_event(mainchart(actRSC(colRSC)), adv, actState(colRSC)),
    newadvs = get_enabled_events(mainchart(actRSC(colRSC)), newstate)
in
    if newadvs = {EnabledStopped} then mk_done(actRSC(colRSC))
    else mk_act(actRSC(colRSC), newstate)
    end
end
pre is_active(colRSC),
```

choose_event chooses an enabled event from a set. This should be nondeterministic. As it is executable we must explicitly do this as done with the play−engine in [26].

```
choose_event : EnabledEvent-set → EnabledEvent
choose_event(advs) ≡ test_get_event(advs),

test_get_event : EnabledEvent-set → EnabledEvent
test_get_event (ass) ≡
if ass = {} then NotEnabled else
let
    el = hd ass
in
    case el of
        NotEnabled →  test_get_event(ass \ {el}),
        EnabledStopped →  test_get_event(ass \ {el}),
        _ → el
    end
end
end,
```

choose_active_RSC chooses an active RSC from the collection. Again, this must be chosen deterministically if several active RSC's are present.

```
choose_active_RSC : Collection ~→ Inst_Name
choose_active_RSC(col) ≡
let el = hd dom col in
    case col(el) of
```

```
        mk_act(_, _) → el,
        _ → choose_active_RSC(col \ {el})
    end
end
pre active_present(col),
```

active_present checks if there is an active RSC in a collection.

```
active_present : Collection → Bool
active_present(col) ≡
    if col = [ ] then false
    else let el = hd dom col in is_active(col(el)) ∨ active_present(col \ {el}) end
    end,
```

is_lsc_activated checks if an RSC is activated by a given advancement. The state-set denotes the states to which the prechart may have proceeded.

```
is_lsc_activated : RSC × State-set × EnabledEvent → Bool
is_lsc_activated(lsc, pcstates, adv) ≡
- - checks whether there exists a state that completes the
- - prechart of lsc by adv given a set of states the prechart
- - may be in
(∃ st : State •
    st ∈ pcstates ∧
    let nextstate = step_event(prechart(lsc), adv, st) in
        get_enabled_events(prechart(lsc), nextstate) = {EnabledStopped}
    end) pre is_visible(lsc, adv),
```

is_visible checks whether the event described by adv is visible on a lsc or not.

```
is_visible : RSC × EnabledEvent → Bool
is_visible(lsc, adv) ≡ adv_id(adv) ∈ lsc_ids(lsc),
```

possible_states returns the set of possible states a prechart may be given possible previous states and an advancement. The state set must be updated for each advancement concerning visible events of the given \lsc.

```
possible_states : RSC × State-set × EnabledEvent ~→ State-set
possible_states(lsc, states, adv) ≡
if states = {} then {initialize_chart(prechart(lsc))}
else
    let state = hd states
    in
        if adv ∈ get_enabled_events(prechart(lsc), state)
        then {step_event(prechart(lsc), adv, state)}
        else {}
        end
        ∪ possible_states(lsc, states \ {state}, adv)
    end
end
```

pre is_visible(lsc, adv),

- ———— WELLFORMEDNESS CONDITIONS————————————-

wf_collection checks whether an collection of LSC's is wellformed or not.

wf_collection : Collection → **Bool**
wf_collection(col) ≡
  wf_col_active(col) ∧
  wf_col_domain(col) ∧
  wf_col_consistent(col),

wf_col_active ensusures that at least one RSC is active or all are done.

wf_col_active : Collection → **Bool**
wf_col_active(col) ≡
(active_present(col) ∨
(∀ iname : Inst_Name •
  iname ∈ **dom** col ⇒
    case col(iname) **of**
     mk_done(_) → **true**,
     _ → **false**
    end)),

The instance names of the collection domain must match the names of the instances. This is redundant information, but practical as we also work on individual LSC's.

wf_col_domain : Collection → **Bool**
wf_col_domain(col) ≡
(∀ iname1 : Inst_Name •
  iname1 ∈ **dom** col ⇒
  (iname1 = name(return_lsc(col(iname1))))),

All the events in the collection must be consistent with regards to event ID′s. If an event ID occurs in two LSC's, it must denote the same event.

wf_col_consistent : Collection → **Bool**
wf_col_consistent(col) ≡
(∀ iname1 : Inst_Name •
  iname1 ∈ **dom** col ⇒
  (∀ iname2 : Inst_Name •
    iname2 ∈ **dom** col ⇒
      (col_lscs_consistent(return_lsc(col(iname1)), return_lsc(col(iname2)))))),
- ——————-AUXILIARY FUNCTIONS FOR WF CONDITIONS————————-

col_lscs_consistent checks whether 2 LSC's are consistent with regards to visible events and their ID′s, ie. that events with the same ID match.

col_lscs_consistent : RSC × RSC → **Bool**
col_lscs_consistent(lsc1, lsc2) ≡

```
let inter_ids = lsc_ids(lsc1) ∩ lsc_ids(lsc2)
in
    event_id_match(lsc1, lsc2, inter_ids) ∧
    event_id_match(lsc2, lsc1, inter_ids)
end,
```

event_id_match checks whether the intersection of ID′s in two LSC's represent the same events (this intersection also denotes the set of visible events to each other.

```
event_id_match : RSC × RSC × ID-set → Bool
event_id_match(lsc1, lsc2, ids) ≡
let eventsinst1 = eventsInst_lsc(lsc1), eventsinst2 = eventsInst_lsc(lsc2) in
   (∀ ei : EventInst •
       ei ∈ eventsinst1 ∧
       case eievent(ei) of
           mk_ActionEvent(_, id) → id ∈ ids,
           mk_InputEvent(id, _) → id ∈ ids,
           mk_OutputEvent(id, _, _) → id ∈ ids,
           mk_ConditionEvent(_, id, _, _) → id ∈ ids,
           mk_Subchart(_, id, _, _, _) → id ∈ ids,
           _ → false
       end ⇒
        case eievent(ei) of
```

Subchart events may differ between lsc1 and lsc2 since they may not contain the same elements.

```
           mk_Subchart(scname, id, scshare, mult, _) →
            (∃ ei2 : EventInst •
                ei2 ∈ eventsinst2 ∧ einame(ei) = einame(ei2) ∧
                case eievent(ei2) of
                    mk_Subchart(scname2, id2, scshare2, mult2, _) →
                        scname2 = scname ∧ id2 = id ∧ scshare2 = scshare ∧ mult2 = mult,
                    _ → false
                end),
           - - rest of events must be in lsc2 as they are
           _ → ei ∈ eventsinst2
        end)
end,

- ———————— AUXILIARY FUNCTIONS ———————————————————————-
```

return_lsc returns the RSC of an collection RSC.

```
return_lsc : ColRSC → RSC
return_lsc(collsc) ≡
case collsc of
    mk_done(doneRSC) → doneRSC,
    mk_act(actRSC, _) → actRSC,
    mk_not_act(notactRSC, _) → notactRSC
end,
```

lsc_ids finds all the ID′s present on a LSC.

```
lsc_ids : RSC → ID-set
lsc_ids(lsc) ≡ all_ids(prechart(lsc)) ∪ all_ids(mainchart(lsc)),
```

eventsInst_lsc finds the pairs of events and corresponding ID′s on a LSC.

```
eventsInst_lsc : RSC → EventInst-set
eventsInst_lsc(lsc) ≡ eventsInst_chart(prechart(lsc)) ∪ eventsInst_chart(mainchart(lsc)),
```

eventsInst_chart finds the pairs of events and corresponding ID′s on a chart.

```
eventsInst_chart : Chart → EventInst-set
eventsInst_chart(chart) ≡
    if chart = [ ] then {}
    else
        let el = hd chart in
            eventsInst_instance(chart(el), el) ∪ eventsInst_chart(chart \ {el})
        end
    end,
```

eventsInst_instance finds the pairs of events and corresponding ID′s on an instance.

```
eventsInst_instance : Location* × Inst_Name → EventInst-set
eventsInst_instance(locl, iname) ≡
    if locl = ⟨⟩ then {}
    else {mk_EventInst(event(hd locl), iname)} ∪ eventsInst_instance(tl locl, iname)
    end,
```

adv_id returns the ID of an EnabledEvent.

```
adv_id : EnabledEvent ~→ ID
adv_id(ee) ≡
case ee of
    EnabledMessage(_,_, id) → id,
    EnabledAction(_,_, id) → id,
    EnabledCondition(_,_, id) → id,
    EnabledEnterSubchart(_, id,_) → id,
    EnabledCoregion(_, _, id, _) → id,
    EnabledEndSubchart(id, _) → id,
    EnabledExitSubchart(_, id,_,_) → id
end,
```

is_active checks if an collection RSC is active.

```
is_active : ColRSC → Bool
is_active(colRSC) ≡
case colRSC of
    mk_act(_, _) → true,
```

```
    _ → false
end

end - -class end
```

D.4 Test

D.4.1 Test of wellformedness conditions

RSC_collection

```
scheme RSC_test1 =
    extend RSC_collection with
    class

    value
```

Test of wellformedness condition wf_ids_unique (nr. 1).

test with messages

```
wf1_m1out : Event = mk_OutputEvent(11, "Msg1", mk_Address("B")),
wf1_m1in : Event = mk_InputEvent(11, mk_Address("A")),
wf1_m2out : Event = mk_OutputEvent(11, "Msg2", mk_Address("A")),
wf1_m2in : Event = mk_InputEvent(21, mk_Address("B")),
wf1_m3out : Event = mk_OutputEvent(21, "Msg2", mk_Address("A")),
wf1_m3in : Event = mk_InputEvent(11, mk_Address("B")),
wf1_m4out : Event = mk_OutputEvent(11, "Msg2", mk_Address("A")),
wf1_m4in : Event = mk_InputEvent(11, mk_Address("B")),
wf1_m5out : Event = mk_OutputEvent(21, "Msg2", mk_Address("A")),
wf1_m5in : Event = mk_InputEvent(21, mk_Address("B")),

wf1_ia1 : Location* =
    ⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf1_m2in)⟩,
wf1_ib1 : Location* =
    ⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_m2out)⟩,
wf1_ch1 : Chart = [ "A" +> wf1_ia1, "B" ↦ wf1_ib1 ],

wf1_ia2 : Location* =
    ⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf1_m3in)⟩,
wf1_ib2 : Location* =
    ⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_m3out)⟩,
wf1_ch2 : Chart = [ "A" +> wf1_ia2, "B" ↦ wf1_ib2 ],

wf1_ia3 : Location* =
    ⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf1_m4in)⟩,
wf1_ib3 : Location* =
    ⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_m4out)⟩,
wf1_ch3 : Chart = [ "A" +> wf1_ia3, "B" ↦ wf1_ib3 ],
```

wf1_ia4 : Location* =
⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf1_m5in)⟩,
wf1_ib4 : Location* =
⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_m5out)⟩,
wf1_ch4 : Chart = [″A" +> wf1_ia4, "B″ ↦ wf1_ib4],

test with message and condition

wf1_c1 : Event = mk_ConditionEvent(″Cond1", 11, Hot, {"B″,
″C″}),
wf1_ia5 : Location* =
⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf1_c1)⟩,
wf1_ib5 : Location* =
⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_c1)⟩,
wf1_ch5 : Chart = [″A" +> wf1_ia5, "B″ ↦ wf1_ib5],

test with message and action event

wf1_tact : Event = mk_ActionEvent(″Action″, 11),
wf1_ia7 : Location* =
⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf1_tact)⟩,
wf1_ib7 : Location* =
⟨mk_Location(Hot, wf1_m1in)⟩,
wf1_ch7 : Chart = [″A" +> wf1_ia7, "B″ ↦ wf1_ib7],

test with message and subchart

wf1_tscla : Location* = ⟨mk_Location(Cold, wf1_m1out)⟩,
wf1_tsclb : Location* = ⟨mk_Location(Cold, wf1_m1in)⟩,
wf1_tsca : Event = mk_Subchart(″Subchart", 11, {"A", "B″},3, wf1_tscla),
wf1_tscb : Event = mk_Subchart(″Subchart", 11, {"A″,
″B″},3, wf1_tsclb),
wf1_scstop : Event = mk_EndSubchart(11),

wf1_ia8 : Location* =
⟨mk_Location(Hot, wf1_tsca), mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf1_scstop)⟩,
wf1_ib8 : Location* =
⟨mk_Location(Hot, wf1_tsca), mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_scstop)⟩,
wf1_ch8 : Chart = [″A" +> wf1_ia8, "B″ ↦ wf1_ib8]

test_case

except for test 4 all tests must return false, therefore negated output

[wf_cond_id_unique_1_____]
∼wf_ids_unique(wf1_ch1),
[wf_cond_id_unique_2_____]
∼wf_ids_unique(wf1_ch2),
[wf_cond_id_unique_3_____]

```
    ~wf_ids_unique(wf1_ch3),
  [ wf_cond_id_unique_4________ ]
  wf_ids_unique(wf1_ch4),
  [ wf_cond_id_unique_5________ ]
  ~wf_ids_unique(wf1_ch5),
  [ wf_cond_id_unique_6________ ]
  ~wf_ids_unique(wf1_ch7),
  [ wf_cond_id_unique_7________ ]
  ~wf_ids_unique(wf1_ch7)
```

test of wellformedness condition wf_message_match (nr. 2)

```
value
    wf2_m1out : Event = mk_OutputEvent(11, "Msg1", mk_Address("B")),
    wf2_m1in : Event = mk_InputEvent(11, mk_Address("A")),
    wf2_m2out : Event = mk_OutputEvent(21, "Msg2", mk_Address("A")),
    wf2_m2in : Event = mk_InputEvent(21, mk_Address("B")),
    wf2_m3out : Event = mk_OutputEvent(21, "Msg2", mk_Address("A")),
    wf2_m3in : Event = mk_InputEvent(11, mk_Address("B")),
    wf2_m4out : Event = mk_OutputEvent(11, "Msg2", mk_Address("A")),
    wf2_m4in : Event = mk_InputEvent(21, mk_Address("B")),

    wf2_ia1 : Location* =
      ⟨mk_Location(Hot, wf2_m1out), mk_Location(Hot, wf2_m2in)⟩,
    wf2_ib1 : Location* =
      ⟨mk_Location(Hot, wf2_m1in)⟩,
    wf2_ch1 : Chart = [ "A" +> wf2_ia1, "B" ↦ wf2_ib1 ],

    wf2_ia2 : Location* =
      ⟨mk_Location(Hot, wf2_m1out)⟩,
    wf2_ib2 : Location* =
      ⟨mk_Location(Hot, wf2_m1in), mk_Location(Hot, wf2_m2out)⟩,
    wf2_ch2 : Chart = [ "A" +> wf2_ia2, "B" ↦ wf2_ib2 ],

    wf2_ia3 : Location* =
      ⟨mk_Location(Hot, wf2_m1out), mk_Location(Hot, wf2_m3in)⟩,
    wf2_ib3 : Location* =
      ⟨mk_Location(Hot, wf2_m1in), mk_Location(Hot, wf2_m3out)⟩,
    wf2_ch3 : Chart = [ "A" +> wf2_ia3, "B" ↦ wf2_ib3 ],

    wf2_ia4 : Location* =
      ⟨mk_Location(Hot, wf2_m1out), mk_Location(Hot, wf2_m4in)⟩,
    wf2_ib4 : Location* =
      ⟨mk_Location(Hot, wf2_m1in), mk_Location(Hot, wf2_m4out)⟩,
    wf2_ch4 : Chart = [ "A" +> wf2_ia4, "B" ↦ wf2_ib4 ]
```

all must return false

```
test_case
  [ wf_message_match_1_________ ]
  ~wf_ids_unique(wf2_ch1),
  [ wf_message_match_2_________ ]
```

∼wf_ids_unique(wf2_ch2),
[wf_message_match_3_____]
∼wf_ids_unique(wf2_ch3),
[wf_message_match_4_____]
∼wf_ids_unique(wf2_ch4)

test of wellformedness condition wf_ (nr. 3)

value

testing cyclic via overtaking

wf3_m1out : Event = mk_OutputEvent(11, "Msg1", mk_Address("B")),
wf3_m1in : Event = mk_InputEvent(11, mk_Address("A")),
wf3_m2out : Event = mk_OutputEvent(21, "Msg2", mk_Address("A")),
wf3_m2in : Event = mk_InputEvent(21, mk_Address("B")),

wf3_ia1 : Location* =
⟨mk_Location(Hot, wf3_m1out), mk_Location(Hot, wf3_m2out)⟩,
wf3_ib1 : Location* =
⟨mk_Location(Hot, wf2_m2in), mk_Location(Hot, wf2_m1in)⟩,
wf3_ch1 : Chart = ["A" +> wf3_ia1, "B" ↦ wf3_ib1],

testing cyclic via other messages

wf3_m3out : Event = mk_OutputEvent(11, "Msg1", mk_Address("A")),
wf3_m3in : Event = mk_InputEvent(11, mk_Address("B")),
wf3_m4out : Event = mk_OutputEvent(21, "Msg2", mk_Address("B")),
wf3_m4in : Event = mk_InputEvent(21, mk_Address("C")),
wf3_m5out : Event = mk_OutputEvent(31, "Msg3", mk_Address("C")),
wf3_m5in : Event = mk_InputEvent(31, mk_Address("A")),

wf3_ia2 : Location* =
⟨mk_Location(Hot, wf3_m5in), mk_Location(Hot, wf3_m3out)⟩,
wf3_ib2 : Location* =
⟨mk_Location(Hot, wf3_m3in), mk_Location(Hot, wf3_m4out)⟩,
wf3_ic2 : Location* =
⟨mk_Location(Hot, wf3_m4in), mk_Location(Hot, wf3_m5out)⟩,
wf3_ch2 : Chart = ["A" +> wf3_ia2, "B" +> wf3_ib2, "C" ↦ wf3_ic2],

testing cyclic via condition and message

wf3_c1 : Event = mk_ConditionEvent("Cond1", 51, Hot, {"B",
"C"}),
wf3_ia3 : Location* =
⟨mk_Location(Hot, wf3_m1out), mk_Location(Hot, wf3_c1)⟩,
wf3_ib3 : Location* =
⟨mk_Location(Hot, wf3_c1), mk_Location(Hot, wf3_m1in)⟩,

wf3_ch3 : Chart = ["A" +> wf3_ia3, "B" ↦ wf3_ib3],

testing cyclic with conditions via instances

wf3_c4 : Event = mk_ConditionEvent("Cond1", 51, Hot, {"B", "C"}),
wf3_c2 : Event = mk_ConditionEvent("Cond1", 52, Hot, {"B", "A"}),
wf3_c3 : Event = mk_ConditionEvent("Cond1", 53, Hot, {"A", "C"}),
wf3_ia4 : Location* =
⟨mk_Location(Hot, wf3_c3), mk_Location(Hot, wf3_c4)⟩,
wf3_ib4 : Location* =
⟨mk_Location(Hot, wf3_c4), mk_Location(Hot, wf3_c2)⟩,
wf3_ic4 : Location* =
⟨mk_Location(Hot, wf3_c2), mk_Location(Hot, wf3_c3)⟩,
wf3_ch4 : Chart = ["A" +> wf3_ia4, "B" +> wf3_ib4, "C" ↦ wf3_ic4]

test_case
[wf_mess_cond_acyclic_1_____]
∼wf_mess_cond_acyclic(wf3_ch1),
[wf_mess_cond_acyclic_2_____]
∼wf_mess_cond_acyclic(wf3_ch2),
[wf_mess_cond_acyclic_3_____]
∼wf_mess_cond_acyclic(wf3_ch3),
[wf_mess_cond_acyclic_4_____]
∼wf_mess_cond_acyclic(wf3_ch4)

test of wellformedness condition wf_condition_share (nr. 4)

value

testing with instance missing condition

wf4_c1 : Event = mk_ConditionEvent("Cond1", 51, Hot, {"A","B", "C"}),
wf4_ia1 : Location* =
⟨mk_Location(Hot, wf4_c1)⟩,
wf4_ib1 : Location* =
⟨mk_Location(Hot, wf4_c1), mk_Location(Hot, wf1_m1in)⟩,
wf4_ic1 : Location* =
⟨mk_Location(Hot, wf1_m1out)⟩,
wf4_ch1 : Chart = ["A" +> wf4_ia1, "B" +> wf4_ib1, "C" ↦ wf4_ic1],

testing with wrong temperature at one instance

wf4_c2 : Event = mk_ConditionEvent("Cond1", 51, Cold, {"A","B","C"}),
wf4_ia2 : Location* =

⟨mk_Location(Hot, wf4_c1)⟩,
wf4_ib2 : Location* =
⟨mk_Location(Hot, wf4_c1)⟩,
wf4_ic2 : Location* =
⟨mk_Location(Hot, wf4_c2)⟩,
wf4_ch2 : Chart = [″A" +> wf4_ia2, "B" +> wf4_ib2, "C″ ↦
wf4_ic2],

testing with wrong name at one instance

wf4_c3 : Event = mk_ConditionEvent(″Cond3", 51, Hot, {"A","B″}),
wf4_ia3 : Location* =
⟨mk_Location(Hot, wf4_c1)⟩,
wf4_ib3 : Location* =
⟨mk_Location(Hot, wf4_c1)⟩,
wf4_ic3 : Location* =
⟨mk_Location(Hot, wf4_c3)⟩,
wf4_ch3 : Chart = [″A" +> wf4_ia3, "B" +> wf4_ib3, "C″ ↦ wf4_ic3]

test_case
[wf_condition_share_1_____]
∼wf_condition_share(wf4_ch1),
[wf_condition_share_2_____]
∼wf_condition_share(wf4_ch2),
[wf_condition_share_3_____]
∼wf_condition_share(wf4_ch3)

test of wellformedness condition wf_subchart_locations (nr.5)

value

testing correct chart

wf5_scla : Location* = ⟨mk_Location(Hot, wf1_m1out),
mk_Location(Hot, wf4_c1)⟩,
wf5_sclb : Location* = ⟨mk_Location(Hot, wf1_m1in),
mk_Location(Hot, wf4_c1)⟩,
wf5_sca : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_scla),
wf5_scstop : Event = mk_EndSubchart(13),
wf5_scb : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_sclb),
wf5_ia1 : Location* =
⟨mk_Location(Hot, wf5_sca),mk_Location(Hot, wf1_m1out),mk_Location(Hot, wf4_c1) ⟩,
wf5_ib1 : Location* =
⟨mk_Location(Hot, wf5_scb),mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf4_c1)⟩,
wf5_ch1 : Chart = [″A" +> wf5_ia1, "B″ ↦ wf5_ib1],

testing missing event on instance outside subchart

wf5_ia2 : Location* =
⟨mk_Location(Hot, wf5_sca),mk_Location(Hot, wf1_m1out)⟩,

wf5_ib2 : Location* =
⟨mk_Location(Hot, wf5_scb),mk_Location(Hot, wf1_m1in)⟩,
wf5_ch2 : Chart = [″A" +> wf5_ia2, "B″ ↦ wf5_ib2],

testing wrong ordering of events

wf5_ia3 : Location* =
⟨mk_Location(Hot, wf5_sca), mk_Location(Hot, wf4_c1),mk_Location(Hot, wf1_m1out) ⟩,
wf5_ib3 : Location* =
⟨mk_Location(Hot, wf5_scb), mk_Location(Hot, wf4_c1),mk_Location(Hot, wf1_m1in)⟩,
wf5_ch3 : Chart = [″A" +> wf5_ia3, "B″ ↦ wf5_ib3],

testing ″missing″ event in subchart

wf5_scla2 : Location* = ⟨mk_Location(Hot, wf4_c1)⟩,
wf5_sclb2 : Location* = ⟨mk_Location(Hot, wf4_c1)⟩,
wf5_sca2 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_scla2),
wf5_scb2 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_sclb2),
wf5_ia4 : Location* =
⟨mk_Location(Hot, wf5_sca2),mk_Location(Hot, wf1_m1out),mk_Location(Hot, wf4_c1) ⟩,
wf5_ib4 : Location* =
⟨mk_Location(Hot, wf5_scb2),mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf4_c1)⟩,
wf5_ch4 : Chart = [″A" +> wf5_ia4, "B″ ↦ wf5_ib4],

test of subchart inside subchart

wf5_scla5 : Location* = ⟨mk_Location(Hot, wf1_m1out)⟩,
wf5_sclb5 : Location* = ⟨mk_Location(Hot, wf1_m1in)⟩,
wf5_sca5 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_scla5),
wf5_scb5 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_sclb5),
wf5_scstop5 : Event = mk_EndSubchart(13),

wf5_scla6 : Location* = ⟨mk_Location(Hot, wf5_sca5)⟩^wf5_scla5^⟨mk_Location(Hot, wf5_scstop5)⟩,
wf5_sclb6 : Location* = ⟨mk_Location(Hot, wf5_scb5)⟩^wf5_sclb5^⟨mk_Location(Hot, wf5_scstop5)⟩,
wf5_sca6 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf5_scla6),
wf5_scb6 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf5_sclb6),
wf5_scstop6 : Event = mk_EndSubchart(14),

wf5_ia5 : Location* =
⟨mk_Location(Hot, wf5_sca6),mk_Location(Hot, wf5_sca5),mk_Location(Hot, wf1_m1out),
mk_Location(Hot, wf5_scstop5),mk_Location(Hot, wf5_scstop6)⟩,
wf5_ib5 : Location* =
⟨mk_Location(Hot,
wf5_scb6),mk_Location(Hot,wf5_scb5),mk_Location(Hot, wf1_m1in),
mk_Location(Hot, wf5_scstop5),mk_Location(Hot, wf5_scstop6)⟩,
wf5_ch5 : Chart = [″A" +> wf5_ia5, "B″ ↦ wf5_ib5],

test of subchart inside subchart with subsubchart wrong ordering

wf5_scla7 : Location* = ⟨mk_Location(Hot, wf1_m1out),mk_Location(Hot, wf1_m5out)⟩,
wf5_sclb7 : Location* = ⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_m5in)⟩,
wf5_sca7 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_scla7),
wf5_scb7 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf5_sclb7),
wf5_scstop7 : Event = mk_EndSubchart(13),

wf5_scla8 : Location* = ⟨mk_Location(Hot, wf5_sca7)⟩^wf5_scla7^⟨mk_Location(Hot, wf5_scstop7)⟩,
wf5_sclb8 : Location* = ⟨mk_Location(Hot, wf5_scb7)⟩^wf5_sclb7^⟨mk_Location(Hot, wf5_scstop7)⟩,
wf5_sca8 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf5_scla8),
wf5_scb8 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf5_sclb8),
wf5_scstop8 : Event = mk_EndSubchart(14),

wf5_ia6 : Location* =
⟨mk_Location(Hot, wf5_sca8),mk_Location(Hot,wf5_sca7),mk_Location(Hot, wf1_m5out),
mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf5_scstop7),mk_Location(Hot, wf5_scstop8)⟩,
wf5_ib6 : Location* =
⟨mk_Location(Hot, wf5_scb8),mk_Location(Hot,wf5_scb7),mk_Location(Hot, wf1_m5in),
mk_Location(Hot, wf1_m1in),mk_Location(Hot, wf5_scstop7),mk_Location(Hot, wf5_scstop8)⟩,
wf5_ch6 : Chart = [″A" +> wf5_ia6, "B″ ↦ wf5_ib6]

test_case
[wf_subchart_locations_1____]
wf_subchart_locations(wf5_ch1),
[wf_subchart_locations_2____]
∼wf_subchart_locations(wf5_ch2),
[wf_subchart_locations_3____]
∼wf_subchart_locations(wf5_ch3),
[wf_subchart_locations_4____]
∼wf_subchart_locations(wf5_ch4),
[wf_subchart_locations_5____]
wf_subchart_locations(wf5_ch5),
[wf_subchart_locations_6____]
∼wf_subchart_locations(wf5_ch6)

test of wellformedness condition wf_subchart_ordered (nr. 6)

value

testing subcharts in correct order

wf6_scla1 : Location* = ⟨mk_Location(Hot, wf1_m1out)⟩,
wf6_sclb1 : Location* = ⟨mk_Location(Hot, wf1_m1in)⟩,
wf6_sca1 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf6_scla1),
wf6_scb1 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf6_sclb1),

wf6_scla2 : Location* = ⟨mk_Location(Hot, wf1_m5out)⟩,
wf6_sclb2 : Location* = ⟨mk_Location(Hot, wf1_m5in)⟩,
wf6_sca2 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf6_scla2),
wf6_scb2 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf6_sclb2),

wf6_ia1 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot,wf1_m1out),mk_Location(Hot, wf6_sca2),
mk_Location(Hot, wf1_m5out) ⟩,
wf6_ib1 : Location* =
⟨mk_Location(Hot, wf6_scb1),mk_Location(Hot, wf1_m1in),mk_Location(Hot, wf6_scb2),
mk_Location(Hot, wf1_m5in) ⟩,

wf6_ch1 : Chart = ["A" +> wf6_ia1, "B" ↦ wf6_ib1],

testing wrong order

wf6_ia2 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot, wf1_m1out),mk_Location(Hot, wf6_sca2),
mk_Location(Hot, wf1_m5out) ⟩,
wf6_ib2 : Location* =
⟨mk_Location(Hot, wf6_scb2),mk_Location(Hot, wf1_m1in),mk_Location(Hot, wf6_scb1),
mk_Location(Hot, wf1_m5in) ⟩,
wf6_ch2 : Chart = ["A" +> wf6_ia2, "B" ↦ wf6_ib2]

test_case
[wf_subchart_ordered_1_____]
wf_subchart_ordered(wf6_ch1),
[wf_subchart_ordered_2_____]
∼wf_subchart_ordered(wf6_ch2)

test of wellformedness condition wf_subchart_coherent (nr. 7)

value

testing missing subchart

wf7_ia1 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot,wf1_m1out),mk_Location(Hot, wf6_sca2),
mk_Location(Hot, wf1_m5out) ⟩,
wf7_ib1 : Location* =
⟨mk_Location(Hot, wf6_scb1),mk_Location(Hot, wf1_m1in),mk_Location(Hot, wf1_m5in) ⟩,
wf7_ch1 : Chart = ["A" +> wf7_ia1, "B" ↦ wf7_ib1],

testing wrong temperature

wf7_ia2 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot,wf1_m1out),mk_Location(Hot, wf6_sca2),
mk_Location(Hot, wf1_m5out) ⟩,
wf7_ib2 : Location* =
⟨mk_Location(Hot, wf6_scb1),mk_Location(Hot,wf1_m1in),mk_Location(Cold, wf6_scb2),
mk_Location(Cold, wf1_m5in) ⟩,
wf7_ch2 : Chart = ["A" +> wf7_ia2, "B" ↦ wf7_ib2]

test_case
[wf_subchart_coherent_1_____]
∼wf_subchart_coherent(wf7_ch1),
[wf_subchart_coherent_2_____]
∼wf_subchart_coherent(wf7_ch2)

test of wellformedness condition wf_subchart_end (nr. 8)

value

testing with subchart end

wf8_ia1 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot, wf1_m1out),mk_Location(Hot, mk_EndSubchart(13))⟩,
wf8_ib1 : Location* =
⟨mk_Location(Hot, wf6_scb1),mk_Location(Hot, wf1_m1in),mk_Location(Hot, mk_EndSubchart(13))⟩,
wf8_ch1 : Chart = ["A" +> wf8_ia1, "B" ↦ wf8_ib1],

testing without subchart end

wf8_ia2 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot, wf1_m1out) ⟩,
wf8_ib2 : Location* =
⟨mk_Location(Hot, wf6_scb1),mk_Location(Hot, wf1_m1in),mk_Location(Hot, mk_EndSubchart(13))⟩,
wf8_ch2 : Chart = ["A" +> wf8_ia2, "B" ↦ wf8_ib2],

testing with wrong subchart end

wf8_ia3 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot, wf1_m1out),mk_Location(Hot, mk_EndSubchart(14))⟩,
wf8_ib3 : Location* =
⟨mk_Location(Hot, wf6_scb1),mk_Location(Hot, wf1_m1in),mk_Location(Hot, mk_EndSubchart(14))⟩,
wf8_ch3 : Chart = ["A" +> wf8_ia3, "B" ↦ wf8_ib3],

testing with wrong temperature subchart end

wf8_ia4 : Location* =
⟨mk_Location(Hot, wf6_sca1),mk_Location(Hot, wf1_m1out),mk_Location(Cold, mk_EndSubchart(13))⟩,
wf8_ib4 : Location* =
⟨mk_Location(Hot, wf6_scb1),mk_Location(Hot, wf1_m1in),mk_Location(Cold, mk_EndSubchart(13))⟩,
wf8_ch4 : Chart = ["A" +> wf8_ia4, "B" ↦ wf8_ib4]

test_case

[wf_subchart_end_1__________]
wf_subchart_end(wf8_ch1),
[wf_subchart_end_2__________]
∼wf_subchart_end(wf8_ch2),
[wf_subchart_end_3__________]
∼wf_subchart_end(wf8_ch3),
[wf_subchart_end_4__________]
∼wf_subchart_end(wf8_ch4)

test of wellformedness condition wf_subchart_conditions (nr. 9)

value

testing with condition correct inside

```
wf9_c1 : Event = mk_ConditionEvent("Cond1", 51, Cold, {"A","B"}),
wf9_scla1 : Location* = ⟨mk_Location(Hot, wf9_c1)⟩,
wf9_sclb1 : Location* = ⟨mk_Location(Hot, wf9_c1)⟩,
wf9_sca1 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf9_scla1),
wf9_scb1 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf9_sclb1),

wf9_ia1 : Location* =
    ⟨mk_Location(Hot, wf9_sca1),mk_Location(Hot, wf9_c1) ⟩,
wf9_ib1 : Location* =
    ⟨mk_Location(Hot, wf9_scb1),mk_Location(Hot, wf9_c1) ⟩,
wf9_ch1 : Chart = ["A" +> wf9_ia1, "B" ↦ wf9_ib1 ],
```

testing with condition outside of subchart, horizontally (another instance). if condition outside vertically it will be caught via wf_subchart_coherent

```
wf9_c2 : Event = mk_ConditionEvent("Cond1", 51, Cold, {"A","B","C"}),
wf9_scla2 : Location* = ⟨mk_Location(Hot, wf9_c2)⟩,
wf9_sclb2 : Location* = ⟨mk_Location(Hot, wf9_c2)⟩,
wf9_sca2 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf9_scla2),
wf9_scb2 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf9_sclb2),

wf9_ia2 : Location* =
    ⟨mk_Location(Hot, wf9_sca2),mk_Location(Hot, wf9_c2) ⟩,
wf9_ib2 : Location* =
    ⟨mk_Location(Hot, wf9_scb2),mk_Location(Hot, wf9_c2) ⟩,
wf9_ic2 : Location* =
    ⟨mk_Location(Hot, wf9_c2)⟩,
wf9_ch2 : Chart = ["A" +> wf9_ia2, "B" +> wf9_ib2, "C" ↦ wf9_ic2]
```

test_case
[wf_subchart_conditions_1___]
wf_subchart_conditions(wf9_ch1),
[wf_subchart_conditions_2___]
∼wf_subchart_conditions(wf9_ch2)

test of wellformedness condition wf_subchart_messages (nr. 10)

value

testing output outside subchart

```
wf10_scla1 : Location* = ⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, wf4_c1)⟩,
wf10_sclb1 : Location* = ⟨mk_Location(Hot, wf4_c1)⟩,
wf10_sca1 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf10_scla1),
wf10_scb1 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf10_sclb1),
wf10_ia1 : Location* =
    ⟨mk_Location(Hot, wf10_sca1),mk_Location(Hot, wf1_m1out),mk_Location(Hot, wf4_c1) ⟩,
wf10_ib1 : Location* =
    ⟨mk_Location(Hot, wf10_scb1),mk_Location(Hot, wf4_c1)⟩,
wf10_ic1 : Location* = ⟨mk_Location(Hot,wf1_m1in)⟩,
wf10_ch1 : Chart = ["A" +> wf10_ia1, "B" ↦ wf10_ib1 ],
```

testing input outside subchart

wf10_scla2 : Location* = ⟨mk_Location(Hot, wf4_c1)⟩,
wf10_sclb2 : Location* = ⟨mk_Location(Hot, wf1_m1in),
mk_Location(Hot, wf4_c1)⟩,
wf10_sca2 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf10_scla2),
wf10_scb2 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf10_sclb2),
wf10_ia2 : Location* =
⟨mk_Location(Hot, wf10_sca2),mk_Location(Hot, wf4_c1) ⟩,
wf10_ib2 : Location* =
⟨mk_Location(Hot, wf10_scb2),mk_Location(Hot,
wf1_m1in),mk_Location(Hot, wf4_c1)⟩,
wf10_ic2 : Location* = ⟨mk_Location(Hot,wf1_m1in)⟩,
wf10_ch2 : Chart = [″A" +> wf10_ia2, "B″ ↦ wf10_ib2]

test_case
[wf_subchart_messages_1_____]
∼wf_subchart_messages(wf10_ch1),
[wf_subchart_messages_2_____]
∼wf_subchart_messages(wf10_ch2)

test of wellformedness condition wf_subchart_subchart (nr. 11)

value

test of subchart inside subchart

wf11_scla1 : Location* = ⟨mk_Location(Hot, wf1_m1out)⟩,
wf11_sclb1 : Location* = ⟨mk_Location(Hot, wf1_m1in)⟩,
wf11_sca1 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf11_scla1),
wf11_scb1 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf11_sclb1),
wf11_scstop1 : Event = mk_EndSubchart(13),

wf11_scla2 : Location* = ⟨mk_Location(Hot,wf11_sca1)⟩^wf11_scla1^
⟨mk_Location(Hot, wf11_scstop1)⟩,
wf11_sclb2 : Location* = ⟨mk_Location(Hot,wf11_scb1)⟩^wf11_sclb1^
⟨mk_Location(Hot, wf11_scstop1)⟩,
wf11_sca2 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf11_scla2),
wf11_scb2 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf11_sclb2),
wf11_scstop2 : Event = mk_EndSubchart(14),

wf11_ia1 : Location* =
⟨mk_Location(Hot, wf11_sca2),mk_Location(Hot,wf11_sca1),mk_Location(Hot, wf1_m1out),
mk_Location(Hot, wf11_scstop1),mk_Location(Hot, wf11_scstop2)⟩,
wf11_ib1 : Location* =
⟨mk_Location(Hot, wf11_scb2),mk_Location(Hot,wf11_scb1),mk_Location(Hot, wf1_m1in),
mk_Location(Hot, wf11_scstop1),mk_Location(Hot, wf11_scstop2)⟩,
wf11_ch1 : Chart = [″A" +> wf11_ia1, "B″ ↦ wf11_ib1],

test of sub−subchart that extends beyond main−subchart (instance−wise)

wf11_scla3 : Location* = ⟨mk_Location(Hot, wf1_m1out)⟩,
wf11_sclb3 : Location* = ⟨mk_Location(Hot, wf1_m1in)⟩,
wf11_sca3 : Event = mk_Subchart(″Subchart", 13, {"A", "B", "C″}, 3, wf11_scla3),
wf11_scb3 : Event = mk_Subchart(″Subchart", 13, {"A", "B", "C″}, 3, wf11_sclb3),
wf11_scstop3 : Event = mk_EndSubchart(13),

wf11_scla4 : Location* = ⟨mk_Location(Hot,wf11_sca3)⟩^wf11_scla3^
⟨mk_Location(Hot, wf11_scstop3)⟩,
wf11_sclb4 : Location* = ⟨mk_Location(Hot,wf11_scb3)⟩^wf11_sclb3^
⟨mk_Location(Hot, wf11_scstop3)⟩,
wf11_sca4 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf11_scla4),
wf11_scb4 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf11_sclb4),
wf11_scstop4 : Event = mk_EndSubchart(14),

wf11_ia2 : Location* =
⟨mk_Location(Hot, wf11_sca4),mk_Location(Hot,wf11_sca3),mk_Location(Hot, wf1_m1out),
mk_Location(Hot, wf11_scstop3),mk_Location(Hot, wf11_scstop4)⟩,
wf11_ib2 : Location* =
⟨mk_Location(Hot, wf11_scb4),mk_Location(Hot,wf11_scb3),mk_Location(Hot, wf1_m1in),
mk_Location(Hot, wf11_scstop3),mk_Location(Hot, wf11_scstop4)⟩,
wf11_ch2 : Chart = [″A" +> wf11_ia2, "B″ ↦ wf11_ib2],

test of sub−subchart that ends outside main− subchart

wf11_ia3 : Location* =
⟨mk_Location(Hot, wf11_sca2),mk_Location(Hot,wf11_sca1),mk_Location(Hot, wf1_m1out),
mk_Location(Hot, wf11_scstop2),mk_Location(Hot, wf11_scstop1)⟩,
wf11_ib3 : Location* =
⟨mk_Location(Hot, wf11_scb2),mk_Location(Hot,wf11_scb1),mk_Location(Hot, wf1_m1in),
mk_Location(Hot, wf11_scstop2),mk_Location(Hot, wf11_scstop1)⟩,
wf11_ch3 : Chart = [″A" +> wf11_ia3, "B″ ↦ wf11_ib3]

test_case
[wf_subchart_subchart_1_____]
wf_subchart_subchart(wf11_ch1) ∧ wf_subchart_end(wf11_ch1) ∧ wf_subchart_locations(wf11_ch1),
[wf_subchart_subchart_2_____]
∼wf_subchart_subchart(wf11_ch2),
[wf_subchart_subchart_3_____]
wf_subchart_subchart(wf11_ch3)

test of wellformedness condition wf_coregion_locations (nr. 12)

value

test coregion with same ordering on instance

wf12_cr1 : Event = mk_CoregionEvent(⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, wf1_m5out)⟩),
wf12_cr2 : Event = mk_CoregionEvent(⟨mk_Location(Hot, wf1_m1out),mk_Location(Hot,wf1_m5in)⟩),
wf12_ia1 : Location* =
⟨mk_Location(Hot, wf12_cr1),mk_Location(Hot, wf1_m1in),mk_Location(Hot, wf1_m5out)⟩,

wf12_ib1 : Location* =
⟨mk_Location(Hot, wf12_cr2),mk_Location(Hot, wf1_m1out),mk_Location(Hot, wf1_m5in)⟩,
wf12_ch1 : Chart = ["A" +> wf12_ia1, "B" ↦ wf12_ib1],

test coregion with different ordering on instance

wf12_ia2 : Location* =
⟨mk_Location(Hot, wf12_cr1),mk_Location(Hot, wf1_m5out),mk_Location(Hot, wf1_m1in)⟩,
wf12_ib2 : Location* =
⟨mk_Location(Hot, wf12_cr2),mk_Location(Hot, wf1_m1in),mk_Location(Hot, wf1_m1out)⟩,
wf12_ch2 : Chart = ["A" +> wf12_ia2, "B" ↦ wf12_ib2],

test coregion with missing event on instance

wf12_ia3 : Location* =
⟨mk_Location(Hot, wf12_cr1),mk_Location(Hot, wf1_m5out),mk_Location(Hot, wf1_m1in)⟩,
wf12_ib3 : Location* =
⟨mk_Location(Hot, wf12_cr2),mk_Location(Hot, wf1_m1in)⟩,
wf12_ch3 : Chart = ["A" +> wf12_ia3, "B" ↦ wf12_ib3],

test coregion with event placed wrongly on instance

wf12_ia4 : Location* =
⟨mk_Location(Hot, wf12_cr1),mk_Location(Hot, wf1_m1in),mk_Location(Hot, wf1_m5out)⟩,
wf12_ib4 : Location* =
⟨mk_Location(Hot, wf12_cr2),mk_Location(Hot,
wf1_m1out),mk_Location(Hot, wf1_m5out), mk_Location(Hot, wf1_m5in)⟩,
wf12_ch4 : Chart = ["A" +> wf12_ia4, "B" ↦ wf12_ib4]

test_case
[wf_coregion_locations_1____]
wf_coregion_locations(wf12_ch1),
[wf_coregion_locations_2____]
∼wf_coregion_locations(wf12_ch2),
[wf_coregion_locations_3____]
∼wf_coregion_locations(wf12_ch3),
[wf_coregion_locations_4____]
∼wf_coregion_locations(wf12_ch4)

test of wellformedness condition wf_coregion_message (nr. 13)

value

test with only messages in coregion (using chart from previous example)

test coregion with other event

```
wf13_cr1 : Event = mk_CoregionEvent(⟨mk_Location(Hot, wf11_scstop1), mk_Location(Hot, wf1_m5out)⟩),
wf13_cr2 : Event = mk_CoregionEvent(⟨mk_Location(Hot,wf1_m5in)⟩),
wf13_ia2 : Location* =
    ⟨mk_Location(Hot, wf13_cr1),mk_Location(Hot, wf11_scstop1),mk_Location(Hot, wf1_m5out)⟩,
wf13_ib2 : Location* =
    ⟨mk_Location(Hot, wf13_cr2),mk_Location(Hot, wf1_m5in)⟩,
wf13_ch2 : Chart = [ "A" +> wf13_ia2, "B" ↦ wf13_ib2 ]
```

test_case
[wf_coregion_messages_1_____]
wf_coregion_messages(wf12_ch1),
[wf_coregion_messages_2_____]
∼wf_coregion_messages(wf13_ch2)

test of wellformedness condition wf_cold_subchart (nr. 14)

value

test of hot locations in subchart using values from wf nr. 5

test of cold locations in subchart

```
wf14_scla1 : Location* = ⟨mk_Location(Cold, wf1_m1out),
mk_Location(Cold, wf4_c1)⟩,
wf14_sclb1 : Location* = ⟨mk_Location(Cold, wf1_m1in),
mk_Location(Cold, wf4_c1)⟩,
wf14_sca1 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf14_scla1),
wf14_scb1 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf14_sclb1),
wf14_ia1 : Location* =
    ⟨mk_Location(Hot, wf14_sca1),mk_Location(Cold, wf1_m1out),mk_Location(Cold, wf4_c1) ⟩,
wf14_ib1 : Location* =
    ⟨mk_Location(Hot, wf14_scb1),mk_Location(Cold, wf1_m1in),mk_Location(Cold, wf4_c1)⟩,
wf14_ch1 : Chart = [ "A" +> wf14_ia1, "B" ↦ wf14_ib1 ],
```

test of cold followed by hot location in subchart

```
wf14_scla2 : Location* = ⟨mk_Location(Cold, wf1_m1out),
mk_Location(Hot, wf4_c1)⟩,
wf14_sclb2 : Location* = ⟨mk_Location(Cold, wf1_m1in),
mk_Location(Hot, wf4_c1)⟩,
wf14_sca2 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf14_scla2),
wf14_scb2 : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 3, wf14_sclb2),
wf14_ia2 : Location* =
    ⟨mk_Location(Hot, wf14_sca2),mk_Location(Cold, wf1_m1out),mk_Location(Hot, wf4_c1) ⟩,
wf14_ib2 : Location* =
    ⟨mk_Location(Hot, wf14_scb2),mk_Location(Cold, wf1_m1in),mk_Location(Hot, wf4_c1)⟩,
wf14_ch2 : Chart = [ "A" +> wf14_ia2, "B" ↦ wf14_ib2 ],
```

test of subchart with cold event inside sub−subchart

wf14_scla3 : Location* = ⟨mk_Location(Cold, wf1_m1out)⟩,
wf14_sclb3 : Location* = ⟨mk_Location(Cold, wf1_m1in)⟩,
wf14_sca3 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf14_scla3),
wf14_scb3 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf14_sclb3),
wf14_scstop3 : Event = mk_EndSubchart(13),

wf14_scla4 : Location* = ⟨mk_Location(Hot, wf14_sca3)⟩^wf14_scla3^⟨mk_Location(Hot, wf14_scstop3)⟩,
wf14_sclb4 : Location* = ⟨mk_Location(Hot, wf14_scb3)⟩^wf14_sclb3^⟨mk_Location(Hot, wf14_scstop3)⟩,
wf14_sca4 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf14_scla4),
wf14_scb4 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf14_sclb4),
wf14_scstop4 : Event = mk_EndSubchart(14),

wf14_ia3 : Location* =
⟨mk_Location(Hot, wf14_sca4),mk_Location(Hot,wf14_sca3),mk_Location(Cold, wf1_m1out),
mk_Location(Hot, wf14_scstop3),mk_Location(Hot, wf14_scstop4)⟩,
wf14_ib3 : Location* =
⟨mk_Location(Hot, wf14_scb4),mk_Location(Hot,wf14_scb3),mk_Location(Cold, wf1_m1in),
mk_Location(Hot, wf14_scstop3),mk_Location(Hot, wf14_scstop4)⟩,
wf14_ch3 : Chart = [″A" +> wf14_ia3, "B″ ↦ wf14_ib3],

test of subchart with cold event followed by hot event inside subchart

wf14_scla5 : Location* = ⟨mk_Location(Cold, wf1_m1out),mk_Location(Hot, wf1_m5out)⟩,
wf14_sclb5 : Location* = ⟨mk_Location(Cold, wf1_m1in), mk_Location(Hot, wf1_m5in)⟩,
wf14_sca5 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf14_scla5),
wf14_scb5 : Event = mk_Subchart(″Subchart", 13, {"A", "B″}, 3, wf14_sclb5),
wf14_scstop5 : Event = mk_EndSubchart(13),

wf14_scla6 : Location* = ⟨mk_Location(Hot, wf14_sca5)⟩^wf14_scla5^⟨mk_Location(Hot, wf14_scstop5)⟩,
wf14_sclb6 : Location* = ⟨mk_Location(Hot, wf14_scb5)⟩^wf14_sclb5^⟨mk_Location(Hot, wf14_scstop5)⟩,
wf14_sca6 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf14_scla6),
wf14_scb6 : Event = mk_Subchart(″Subchart2", 14, {"A", "B″}, 1, wf14_sclb6),
wf14_scstop6 : Event = mk_EndSubchart(14),

wf14_ia4 : Location* =
⟨mk_Location(Hot,wf14_sca6)⟩^wf14_scla6^⟨mk_Location(Hot, wf14_scstop6)⟩,
wf14_ib4 : Location* =
⟨mk_Location(Hot,wf14_scb6)⟩^wf14_sclb6^⟨mk_Location(Hot, wf14_scstop6)⟩,
wf14_ch4 : Chart = [″A" +> wf14_ia4, "B″ ↦ wf14_ib4]

test_case
[wf_cold_subchart_1_____]
wf_cold_subchart(wf5_ch1),
[wf_cold_subchart_2_____]
wf_cold_subchart(wf14_ch1),
[wf_cold_subchart_3_____]
∼wf_cold_subchart(wf14_ch2),
[wf_cold_subchart_4_____]
wf_cold_subchart(wf14_ch3),
[wf_cold_subchart_5_____]
∼wf_cold_subchart(wf14_ch4)

test of wellformedness condition wf_cold_mainchart (nr. 15)

value

testing hot events followed by cold events

```
        wf15_scstop : Event = mk_EndSubchart(13),
        wf15_ia0 : Location* =
            ⟨mk_Location(Hot, wf1_m1out),
mk_Location(Hot, wf14_sca1),
mk_Location(Cold,wf1_m1out),
mk_Location(Cold,wf4_c1),
mk_Location(Cold, wf15_scstop),
mk_Location(Cold, wf1_m5out) ⟩,
        wf15_ib0 : Location* =
            ⟨mk_Location(Hot, wf1_m1in),
mk_Location(Cold, wf14_scb1),
mk_Location(Cold,wf1_m1in),
mk_Location(Cold,wf4_c1),
mk_Location(Cold, wf15_scstop),
mk_Location(Cold, wf1_m5in)⟩,
        wf15_ch0 : Chart = [″A" +> wf15_ia0, "B″ ↦ wf15_ib0],
```

testing hot mainchart with cold subchart events

```
        wf15_ia1 : Location* =
            ⟨mk_Location(Hot, wf1_m1out),
mk_Location(Hot, wf14_sca1),
mk_Location(Cold,wf1_m1out),
mk_Location(Cold,wf4_c1),
mk_Location(Hot, wf15_scstop),
mk_Location(Hot, wf1_m5out) ⟩,
        wf15_ib1 : Location* =
            ⟨mk_Location(Hot, wf1_m1in),
mk_Location(Hot, wf14_scb1),
mk_Location(Cold,wf1_m1in),
mk_Location(Cold,wf4_c1),
mk_Location(Hot, wf15_scstop),
mk_Location(Hot, wf1_m5in)⟩,
        wf15_ch1 : Chart = [″A" +> wf15_ia1, "B″ ↦ wf15_ib1],
```

testing hot mainchart with cold subchart

```
        wf15_ia2 : Location* =
            ⟨mk_Location(Hot, wf1_m1out),
mk_Location(Cold, wf14_sca1),
mk_Location(Cold,wf1_m1out),
mk_Location(Cold,wf4_c1),
mk_Location(Cold, wf15_scstop),
mk_Location(Hot, wf1_m5out) ⟩,
        wf15_ib2 : Location* =
            ⟨mk_Location(Hot, wf1_m1in),
```

mk_Location(Cold, wf14_scb1),
mk_Location(Cold,wf1_m1in),
mk_Location(Cold,wf4_c1),
mk_Location(Cold, wf15_scstop),
mk_Location(Hot, wf1_m5in)⟩,
wf15_ch2 : Chart = [″A" +> wf15_ia2, "B″ ↦ wf15_ib2],

testing cold mainchart with hot subchart

wf15_ia3 : Location* =
⟨mk_Location(Cold, wf1_m1out),
mk_Location(Cold, wf14_sca1),
mk_Location(Hot,wf1_m1out),
mk_Location(Hot,wf4_c1),
mk_Location(Hot, wf15_scstop),
mk_Location(Cold, wf1_m5out) ⟩,
wf15_ib3 : Location* =
⟨mk_Location(Cold, wf1_m1in),
mk_Location(Cold, wf14_scb1),
mk_Location(Hot, wf1_m1in),
mk_Location(Hot,wf4_c1),
mk_Location(Hot, wf15_scstop),
mk_Location(Cold, wf1_m5in)⟩,
wf15_ch3 : Chart = [″A" +> wf15_ia3, "B″ ↦ wf15_ib3],

testing cold events followed by hot event

wf15_ia4 : Location* =
⟨mk_Location(Cold, wf1_m1out),
mk_Location(Cold, wf14_sca1),
mk_Location(Cold,wf1_m1out),
mk_Location(Cold,wf4_c1),
mk_Location(Cold, wf15_scstop),
mk_Location(Hot, wf1_m5out) ⟩,
wf15_ib4 : Location* =
⟨mk_Location(Cold, wf1_m1in),
mk_Location(Cold, wf14_scb1),
mk_Location(Cold,wf1_m1in),
mk_Location(Cold,wf4_c1),
mk_Location(Cold, wf15_scstop),
mk_Location(Hot, wf1_m5in)⟩,
wf15_ch4 : Chart = [″A" +> wf15_ia4, "B″ ↦ wf15_ib4]

test_case
[wf_cold_mainchart_0_____]
wf_cold_mainchart(wf15_ch0),
[wf_cold_mainchart_1_____]
wf_cold_mainchart(wf15_ch1),
[wf_cold_mainchart_2_____]
∼wf_cold_mainchart(wf15_ch2),
[wf_cold_mainchart_3_____]
∼wf_cold_mainchart(wf15_ch3),
[wf_cold_mainchart_4_____]
∼wf_cold_mainchart(wf15_ch4)

test of wellformedness condition wf_last (nr. 16)

value

test with correct stop event

wf16_ia1 : Location* =
⟨mk_Location(Hot, wf1_m1out), mk_Location(Hot, StopEvent)⟩,
wf16_ib1 : Location* =
⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, StopEvent)⟩,
wf16_ch1 : Chart = ["A" +> wf16_ia1, "B" ↦ wf16_ib1],

test with missing stop event

wf16_ia2 : Location* =
⟨mk_Location(Hot, wf1_m1out)⟩,
wf16_ib2 : Location* =
⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, StopEvent)⟩,
wf16_ch2 : Chart = ["A" +> wf16_ia2, "B" ↦ wf16_ib2],

test with wrong position stop event

wf16_ia3 : Location* =
⟨mk_Location(Hot, StopEvent), mk_Location(Hot, wf1_m1out)⟩,
wf16_ib3 : Location* =
⟨mk_Location(Hot, wf1_m1in), mk_Location(Hot, StopEvent)⟩,
wf16_ch3 : Chart = ["A" +> wf16_ia3, "B" ↦ wf16_ib3]

test_case
[wf_last_1_____]
wf_last(wf16_ch1),
[wf_last_2_____]
∼wf_last(wf16_ch2),
[wf_last_3_____]
∼wf_last(wf16_ch3)

test of one rather large example TODO: henvis til FIGUR fra RAPPORT!

value

defining all the events

wfl_m3out : Event = mk_OutputEvent(31, "Msg3", mk_Address("B")),
wfl_m3in : Event = mk_InputEvent(31, mk_Address("A")),

```
wfl_m4out : Event = mk_OutputEvent(41, "Msg4", mk_Address("B")),
wfl_m4in : Event = mk_InputEvent(41, mk_Address("C")),
wfl_m5out : Event = mk_OutputEvent(51, "Msg5", Environment),
wfl_m6out : Event = mk_OutputEvent(61, "Msg6", mk_Address("B")),
wfl_m6in : Event = mk_InputEvent(61, mk_Address("A")),
wfl_m7out : Event = mk_OutputEvent(71, "Msg7", mk_Address("C")),
wfl_m7in : Event = mk_InputEvent(71, mk_Address("B")),

wfl_m8out : Event = mk_OutputEvent(81, "Create", mk_Address("D")),
wfl_m8in : Event = mk_InputEvent(81, mk_Address("C")),
wfl_m9out : Event = mk_OutputEvent(91, "Msg9", mk_Address("B")),
wfl_m9in : Event = mk_InputEvent(91, mk_Address("C")),

wfl_m8out2 : Event = mk_OutputEvent(81, "Create", Environment),

wfl_m10out : Event = mk_OutputEvent(101, "Msg10", mk_Address("C")),
wfl_m10in : Event = mk_InputEvent(101, mk_Address("B")),
wfl_act : Event = mk_ActionEvent("Action", 15),
wfl_cr1 : Event = mk_CoregionEvent(⟨mk_Location(Hot, wfl_m3in), mk_Location(Hot, wfl_m4in)⟩),
wfl_cr2 : Event = mk_CoregionEvent(⟨mk_Location(Hot, wfl_m7out), mk_Location(Hot, wfl_m9in)⟩),
wfl_cr3 : Event = mk_CoregionEvent(⟨mk_Location(Hot, wfl_m9out), mk_Location(Hot, wfl_m7in)⟩),
wfl_cond1 : Event = mk_ConditionEvent("Cond1", 12, Hot, {"B",
"C"}),
wfl_cond11 : Event = mk_ConditionEvent("Cond1", 12, Cold, {"B", "C"}),
wfl_cond2 : Event = mk_ConditionEvent("Cond2", 22, Hot, {"A", "B"}),
wfl_scla : Location* = ⟨mk_Location(Hot, wfl_cond2), mk_Location(Cold, wfl_m6out)⟩,
wfl_sclb : Location* = ⟨mk_Location(Hot, wfl_cond2), mk_Location(Cold, wfl_m6in)⟩,
wfl_sca : Event = mk_Subchart("Subchart", 13, {"A", "B"}, 2, wfl_scla),
wfl_scstop : Event = mk_EndSubchart(13),
wfl_scb : Event = mk_Subchart("Subchart", 13, {"A", "B"},
2, wfl_sclb),
```

defining main chart instances (location lists)

```
wfl_insta : Location* =
    ⟨mk_Location(Hot, wfl_m3out),
mk_Location(Hot, wfl_act),
        mk_Location(Hot, wfl_sca)⟩
⌢ wfl_scla ⌢
    ⟨mk_Location(Hot, wfl_scstop),
mk_Location(Hot, StopEvent)⟩,
wfl_instb : Location* =
    ⟨mk_Location(Hot, wfl_cr1),
mk_Location(Hot, wfl_m3in),
        mk_Location(Hot, wfl_m4in),
mk_Location(Hot, wfl_cond1),
mk_Location(Hot, wfl_scb)⟩
⌢ wfl_sclb ⌢
    ⟨mk_Location(Hot, wfl_scstop),
mk_Location(Hot, wfl_cr2),
mk_Location(Hot, wfl_m7out),
mk_Location(Hot, wfl_m9in),
        mk_Location(Hot, StopEvent)⟩,
wfl_instc : Location* =
    ⟨mk_Location(Hot, wfl_m4out),
mk_Location(Hot, wfl_cond1),
```

```
                mk_Location(Hot, wfl_m8out),
 mk_Location(Hot, wfl_cr3),
 mk_Location(Hot, wfl_m9out),
 mk_Location(Hot, wfl_m7in),
                mk_Location(Hot, StopEvent)⟩,
        wfl_instd : Location* =
            ⟨mk_Location(Hot, wfl_m8in),
                mk_Location(Hot, wfl_m5out),
 mk_Location(Hot, StopEvent)⟩,
```

defining prechart instances

```
        wfl_insta2 : Location* =
            ⟨mk_Location(Hot, wfl_m3out),
 mk_Location(Hot, wfl_act),
                mk_Location(Hot, wfl_sca)⟩
 ⌢ wfl_scla ⌢
            ⟨mk_Location(Hot, wfl_scstop),
 mk_Location(Hot, StopEvent)⟩,
        wfl_instb2 : Location* =
            ⟨mk_Location(Hot, wfl_cr1),
 mk_Location(Hot, wfl_m3in),
                mk_Location(Hot, wfl_m4in),
 mk_Location(Hot, wfl_cond1),
 mk_Location(Hot, wfl_scb)⟩
 ⌢ wfl_sclb ⌢
            ⟨mk_Location(Hot, wfl_scstop),
 mk_Location(Hot, wfl_m10out),
 mk_Location(Hot, StopEvent)⟩,
        wfl_instb21 : Location* =
            ⟨mk_Location(Hot, wfl_cr1),
 mk_Location(Hot, wfl_m3in),
                mk_Location(Hot, wfl_m4in),
 mk_Location(Hot, wfl_cond11),
 mk_Location(Hot, wfl_scb)⟩
 ⌢ wfl_sclb ⌢
            ⟨mk_Location(Hot, wfl_scstop),
 mk_Location(Hot, wfl_m10out),
 mk_Location(Hot, StopEvent)⟩,
        wfl_instc2 : Location* =
            ⟨mk_Location(Hot, wfl_m4out),
 mk_Location(Hot, wfl_cond1),
                mk_Location(Hot, wfl_m8out2),
 mk_Location(Hot, wfl_m10in),
 mk_Location(Hot, StopEvent)⟩,
```

defining charts

```
        wfl_mainch : Chart = [ ″A″ ↦ wfl_insta,
  ″B″ ↦ wfl_instb,
  ″C″ ↦ wfl_instc,
  ″D″ ↦ wfl_instd ],
        wfl_prech  : Chart = [ ″A″ ↦ wfl_insta2,
  ″B″ ↦ wfl_instb2,
```

$''\texttt{C}'' \mapsto$ wfl_instc2],

For testing wf 18, cold location in prechart

wfl_prech2 : Chart = [$''\texttt{A}'' \mapsto$ wfl_insta2,
$''\texttt{B}'' \mapsto$ wfl_instb21,
$''\texttt{C}'' \mapsto$ wfl_instc2],

defining RSC subtype

wfl_testlsc : RSC′ = mk_RSC′($''$`Test-RSC", wfl_prech, wfl_mainch, {"D`$''$}),

test of wellformedness condition wf_creation(nr. 19), done now since test is done on whole LSC rather than an individual chart

testing with created instance with first event that is not a message

wf19_testlsc1 : RSC′ = mk_RSC′($''$`Test-RSC2", [], wfl_mainch, {"B`$''$}),

testing with created instance in prechart

wf19_testlsc2 : RSC′ = mk_RSC′($''$`Test-RSC2", wfl_mainch, wfl_mainch, {"D`$''$}),

testing with created instance not in mainchart

wf19_testlsc3 : RSC′ = mk_RSC′($''$`Test-RSC2", wfl_mainch, wfl_prech, {"D`$''$})

test_case
[wfl_01wf_ids_unique_____]
wf_ids_unique(wfl_mainch),
[wfl_02wf_message_match_____]
wf_message_match(wfl_mainch),
[wfl_03wf_mess_cond_acyclic_]
wf_mess_cond_acyclic(wfl_mainch),
[wfl_04wf_condition_share___]
wf_condition_share(wfl_mainch),
[wfl_05wf_subchart_locations]
wf_subchart_locations(wfl_mainch),
[wfl_06wf_subchart_ordered__]
wf_subchart_ordered(wfl_mainch),
[wfl_07wf_subchart_coherent_]
wf_subchart_coherent(wfl_mainch),
[wfl_08wf_subchart_end_____]
wf_subchart_end(wfl_mainch),
[wfl_09wf_subchart_condition]
wf_subchart_conditions(wfl_mainch),
[wfl_10wf_subchart_messages_]

```
        wf_subchart_messages(wfl_mainch),
    [wfl_11wf_subchart_subchart_]
        wf_subchart_subchart(wfl_mainch),
    [wfl_12wf_coregion_locations]
        wf_coregion_locations(wfl_mainch),
    [wfl_13wf_coregion_messages_]
        wf_coregion_messages(wfl_mainch),
    [wfl_14wf_cold_subchart_____]
        wf_cold_subchart(wfl_mainch),
    [wfl_15wf_cold_mainchart____]
        wf_cold_mainchart(wfl_mainch),
    [wfl_16wf_last_______________]
        wf_last(wfl_mainch),
    [wfl_17wf_creation_1________]
        wf_creation(wfl_testlsc),
    [wfl_17wf_creation_2________]
        ~wf_creation(wf19_testlsc1),
    [wfl_17wf_creation_3________]
        ~wf_creation(wf19_testlsc2),
    [wfl_17wf_creation_4________]
        ~wf_creation(wf19_testlsc3),
    [wfl_18wf_prechart_condtion1]
         wf_prechart_condition(wfl_prech),
    [wfl_18wf_prechart_condtion2]
         ~wf_prechart_condition(wfl_prech2),
    [wfl_mainchart_wellformed___]
        wf_chart(wfl_mainch),
    [wfl_prechart_wellformed____]
        wf_chart(wfl_prech),
    [wfl_RSC_is_wellformed______]
        wf_RSC(wfl_testlsc)
end
```

D.4.2 Test of semantics

RSC_test1

```
scheme RSC_test2 =
    extend RSC_test1 with
    class
```

The tests of the semantics of a single chart have been automated using test functions. They are explained and defined below.

```
        value
```

The advancement chosen among the set of possibles is nondeterministical. However that is not implementable as is. Therefore this explicit function for chosing one of those advancements is given. Due to the translation in C‖, the same advancement will be returned every time with a specific argument, thus eliminating nondeterminism. This is practical when testing, as the expected result can be calculated at each step. In order to test different runs, we have provided two ways of chosing an advancement.

```
            sem_test_get_adv : EnabledEvent-set × Bool → EnabledEvent
            sem_test_get_adv (ass, run1) ≡
            if run1 then
            if ass = {} then NotEnabled else
            let
                el = hd ass
            in
                case el of
  NotEnabled → sem_test_get_adv(ass \ {el}, run1),
  EnabledStopped → sem_test_get_adv(ass \ {el}, run1),
  _ → el
end
            end
            end
            else
            if ass = {} then NotEnabled else
if card ass = 1 then
  hd ass
                 else
  let
    el = hd ass,
    el2 = hd (ass \ {el})
                    in
    case el2 of
      NotEnabled → sem_test_get_adv(ass \ {el2}, run1),
      EnabledStopped → sem_test_get_adv(ass \ {el2}, run1),
      _ → el2
    end
  end
end
            end
            end,
```

The following 3 functions have been provided in order to display traces, a single trace and a state, respectively, in a more human readable form. This enables the reader more easily to read the expected output from the tests.

Recurses over traces in order to conver them one at a time.

```
            sem_test_convert_traces : Traces → ((Int*)*)-set
            sem_test_convert_traces(traces) ≡
            if traces = {} then {} else
                let
 el = hd traces
in
 {sem_test_convert_trace(el)} ∪ sem_test_convert_traces(traces \{el})
end
            end,
```

Recurses over a trace in order to convert the individual states.

```
        sem_test_convert_trace : Trace → (Int*)*
        sem_test_convert_trace(trace) ≡
        if trace = ⟨⟩ then ⟨⟩ else
sem_test_convert_trace(tl trace)^ ⟨sem_test_convert_state(hd trace)⟩
        end,
```

It converts a state to an int*, so that f.x. ⟨1,1,2,2⟩ means that instance A and B are at Location 1 and C and D are at Location 2.

```
        sem_test_convert_state : State → Int*
        sem_test_convert_state(state) ≡
        ⟨pointer(state("A")), pointer(state("B")), pointer(state("C")), pointer(state("D"))⟩
```

type

Type needed for automatically running the tests. It records how many more steps to perform (i), the current state and the currently possible advancesteps.

```
        gStep :: i: Int gstate: State ads : EnabledEvent-set
```

value

Function for automatically "running" a chart. As long as the chart is not finished (all instances reached the last location) it progresses X steps, determined by a gStep value. It choses an enabled event to perform (using sem_test_get_adv) and uses the semantics function advance_state to return the new state. Based on this is uses the get_enabled_events semantics function in order to get the possible advancements in the new state. Thereafter it simply recurses.

```
        test_machine : Chart × gStep × Bool → gStep
        test_machine(chart, gstep, run1) ≡
        if i(gstep) = 0 then gstep else
        let
            newads = sem_test_get_adv(ads(gstep), run1)
        in
            if newads = NotEnabled then
 gstep
else
 let
  newstate = step_event(wfl_mainch, newads, gstate(gstep))
                  in
  if newstate ≠ [ ] then
    let
     newadss = get_enabled_events(wfl_mainch, newstate)
    in
     test_machine(chart,mk_gStep(i(gstep)−1, newstate,
newadss), run1)
    end
                      else
   gstep
  end
 end
```

end
end
end,

Start state and initial possible advancements. Provided in order to shorten the paramter*.

s0 : State = initialize_chart(wfl_mainch),
a0 : EnabledEvent-**set** = get_enabled_events(wfl_mainch, s0)

test_case
[sem_start_state_____]
initialize_chart(wfl_mainch)=
[″D"+>mk_PosInfo(1,None,<..>),"C″↦mk_PosInfo(1,None,⟨⟩),
″B"+>mk_PosInfo(1,CoRegion({41,31}),<..>),"A″↦mk_PosInfo(1,None,⟨⟩)],
[sem_chart_wf_____]
wf_chart(wfl_mainch),
[sem_state_____0]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(0, s0, a0),**true**))) =
⟨1,1,1,1⟩,
[sem_advance_steps_____0]
ads(test_machine(wfl_mainch, mk_gStep(0,s0,a0),**true**)) =
{NotEnabled,EnabledCoregion({″A"},{"B"},31,3),EnabledCoregion({"C"},{"B″},41,3)},
[sem_advance_step_chosen___0]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(0,s0,a0),**true**)),**true**) =
EnabledCoregion({″A"},{"B″},31,3),
[sem_state_____1]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(1,s0,a0),**true**))) =
⟨2,1,1,1⟩,
[sem_advance_steps_____1]
ads(test_machine(wfl_mainch, mk_gStep(1,s0,a0), **true**)) =
{NotEnabled,EnabledAction(″A","Action",15),EnabledCoregion({"C"},{"B″},41,4)},
[sem_advance_step_chosen___1]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(1,s0,a0), **true**)),**true**) =
EnabledAction(″A","Action″,15),
[sem_state_____2]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(2,s0,a0),**true**))) =
⟨3,1,1,1⟩,
[sem_advance_steps_____2]
ads(test_machine(wfl_mainch, mk_gStep(2,s0,a0),**true**)) =
{NotEnabled,EnabledCoregion({″C"},{"B″},41,4)},
[sem_advance_step_chosen___2]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(2,s0,a0),**true**)),**true**) =
EnabledCoregion({″C"},{"B″},41,4),
[sem_state_____3]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(3,s0,a0),**true**))) =
⟨3,4,2,1⟩,
[sem_advance_steps_____3]
ads(test_machine(wfl_mainch, mk_gStep(3,s0,a0),**true**)) =
{NotEnabled,EnabledCondition({″B","C"},"Cond1″,12)},
[sem_advance_step_chosen___3]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(3,s0,a0),**true**)),**true**) =
EnabledCondition({″B","C"},"Cond1″,12),
[sem_state_____4]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(4,s0,a0),**true**))) =
⟨3,5,3,1⟩,

[sem_advance_steps_____4]
ads(test_machine(wfl_mainch, mk_gStep(4,s0,a0),**true**)) =
{EnabledMessage({″C"},{"D"},81),EnabledEnterSubchart({"A","B″},13,2),NotEnabled},
[sem_advance_step_chosen___4]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(4,s0,a0),**true**)),**true**) =
EnabledMessage({″C"},{"D″},81),
[sem_state_____5]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(5,s0,a0),**true**))) =
⟨3,5,4,2⟩,
[sem_advance_steps_____5]
ads(test_machine(wfl_mainch, mk_gStep(5,s0,a0),**true**)) =
{NotEnabled,EnabledEnterSubchart({″A","B"},13,2),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen___5]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(5,s0,a0),**true**)),**true**) =
EnabledEnterSubchart({″A","B″},13,2),
[sem_state_____6]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(6,s0,a0),**true**))) =
⟨4,6,4,2⟩,
[sem_advance_steps_____6]
ads(test_machine(wfl_mainch, mk_gStep(6,s0,a0),**true**)) =
{NotEnabled,EnabledCondition({″A","B"},"Cond2",22),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen___6]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(6,s0,a0),**true**)),**true**) =
EnabledCondition({″A","B"},"Cond2″,22),
[sem_state_____7]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(7,s0,a0),**true**))) =
⟨5,7,4,2⟩,
[sem_advance_steps_____7]
ads(test_machine(wfl_mainch, mk_gStep(7,s0,a0),**true**)) =
{NotEnabled,EnabledMessage({″A"},{"B"},61),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen___7]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(7,s0,a0),**true**)),**true**) =
EnabledMessage({″A"},{"B″},61),
[sem_state_____8]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(8,s0,a0),**true**))) =
⟨6,8,4,2⟩,
[sem_advance_steps_____8]
ads(test_machine(wfl_mainch, mk_gStep(8,s0,a0),**true**)) =
{NotEnabled,EnabledEndSubchart(13,{″A","B"}),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen___8]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(8,s0,a0),**true**)),**true**) =
EnabledEndSubchart(13,{″A","B″}),
[sem_state_____9]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(9,s0,a0),**true**))) =
⟨4,6,4,2⟩,
[sem_advance_steps_____9]
ads(test_machine(wfl_mainch, mk_gStep(9,s0,a0),**true**)) =
{NotEnabled,EnabledCondition({″A","B"},"Cond2",22),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen___9]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(9,s0,a0),**true**)),**true**) =
EnabledCondition({″A","B"},"Cond2″,22),
[sem_state_____10]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(10,s0,a0),**true**))) =
⟨5,7,4,2⟩,
[sem_advance_steps_____10]
ads(test_machine(wfl_mainch, mk_gStep(10,s0,a0),**true**)) =
{NotEnabled,EnabledMessage({″A"},{"B"},61),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen__10]

sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(10,s0,a0),**true**)),**true**) =
EnabledMessage({″A"},{"B″},61),
[sem_state_____11]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(11,s0,a0),**true**))) =
⟨6,8,4,2⟩,
[sem_advance_steps_____11]
ads(test_machine(wfl_mainch, mk_gStep(11,s0,a0),**true**)) =
{NotEnabled,EnabledEndSubchart(13,{″A","B"}),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen__11]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(11,s0,a0),**true**)),**true**) =
EnabledEndSubchart(13,{″A","B″}),
[sem_state_____12]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(12,s0,a0),**true**))) =
⟨7,9,4,2⟩,
[sem_advance_steps_____12]
ads(test_machine(wfl_mainch, mk_gStep(12,s0,a0),**true**)) =
{EnabledCoregion({″C"},{"B″},91,5),NotEnabled,EnabledStopped,
EnabledCoregion({″B"},{"C"},71,5),EnabledMessage({"D″},{},51)},
[sem_advance_step_chosen__12]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(12,s0,a0),**true**)),**true**) =
EnabledCoregion({″C"},{"B″},91,5),
[sem_state_____13]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(13,s0,a0),**true**))) =
⟨7,9,4,2⟩,
[sem_advance_steps_____13]
ads(test_machine(wfl_mainch, mk_gStep(13,s0,a0),**true**)) =
{NotEnabled,EnabledStopped,EnabledCoregion({″B"},{"C″},71,8),
EnabledMessage({″D″},{},51)},
[sem_advance_step_chosen__13]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(13,s0,a0),**true**)),**true**) =
EnabledCoregion({″B"},{"C″},71,8),
[sem_state_____14]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(14,s0,a0),**true**))) =
⟨7,12,7,2⟩,
[sem_advance_steps_____14]
ads(test_machine(wfl_mainch, mk_gStep(14,s0,a0),**true**)) =
{EnabledStopped,EnabledMessage({″D″},{},51)},
[sem_advance_step_chosen__14]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(14,s0,a0),**true**)),**true**) =
EnabledMessage({″D″},{},51),
[sem_state_____15]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(15,s0,a0),**true**))) =
⟨7,12,7,3⟩,
[sem_advance_steps_____15]
ads(test_machine(wfl_mainch, mk_gStep(15,s0,a0),**true**)) =
{EnabledStopped},
[sem_advance_step_chosen__15]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(15,s0,a0),**true**)),**true**) =
NotEnabled,
[sem_state_____16]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(16, s0, a0),**true**))) =
⟨7,12,7,3⟩,
[sem_advance_steps_____16]
ads(test_machine(wfl_mainch, mk_gStep(16,s0,a0),**true**)) =
{EnabledStopped},
[sem_advance_step_chosen__16]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(16,s0,a0),**true**)),**true**) =
NotEnabled,

[sem_state_____17]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(17, s0, a0),**true**))) =
⟨7,12,7,3⟩,
[sem_advance_steps_____17]
ads(test_machine(wfl_mainch, mk_gStep(17,s0,a0),**true**)) =
{EnabledStopped},
[sem_advance_step_chosen__17]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(17,s0,a0),**true**)),**true**) =
NotEnabled,
[sem_state_____18]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(18, s0, a0),**true**))) =
⟨7,12,7,3⟩,
[sem_advance_steps_____18]
ads(test_machine(wfl_mainch, mk_gStep(18,s0,a0),**true**)) =
{EnabledStopped},
[sem_advance_step_chosen__18]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(18,s0,a0),**true**)),**true**) =
NotEnabled,
[sem2_state_____0]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(0, s0, a0),**false**))) =
⟨1,1,1,1⟩,
[sem2_advance_steps_____0]
ads(test_machine(wfl_mainch, mk_gStep(0,s0,a0),**false**)) =
{NotEnabled,EnabledCoregion({″A",{"B"},31,3),EnabledCoregion({"C"},{"B″},41,3)},
[sem2_advance_step_chosen__0]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(0,s0,a0),**false**)),**false**) =
EnabledCoregion({″A"},{"B″},31,3),
[sem2_state_____1]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(1,s0,a0),**false**))) =
⟨2,1,1,1⟩,
[sem2_advance_steps_____1]
ads(test_machine(wfl_mainch, mk_gStep(1,s0,a0),**false**)) =
{NotEnabled,EnabledAction(″A","Action",15),EnabledCoregion({"C"},{"B″},41,4)},
[sem2_advance_step_chosen__1]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(1,s0,a0),**false**)),**false**) =
EnabledAction(″A","Action″,15),
[sem2_state_____2]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(2, s0, a0),**false**))) =
⟨3,1,1,1⟩,
[sem2_advance_steps_____2]
ads(test_machine(wfl_mainch, mk_gStep(2,s0,a0),**false**)) =
{NotEnabled,EnabledCoregion({″C"},{"B″},41,4)},
[sem2_advance_step_chosen__2]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(2,s0,a0),**false**)),**false**) =
EnabledCoregion({″C"},{"B″},41,4),
[sem2_state_____3]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(3,s0,a0),**false**))) =
⟨3,4,2,1⟩,
[sem2_advance_steps_____3]
ads(test_machine(wfl_mainch, mk_gStep(3,s0,a0),**false**)) =
{NotEnabled,EnabledCondition({″B","C"},"Cond1″,12)},
[sem2_advance_step_chosen__3]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(3,s0,a0),**false**)),**false**) =
EnabledCondition({″B","C"},"Cond1″,12),
[sem2_state_____4]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(4, s0, a0),**false**))) =
⟨3,5,3,1⟩,
[sem2_advance_steps_____4]

ads(test_machine(wfl_mainch, mk_gStep(4,s0,a0),**false**)) =
{EnabledMessage({″C″},{"D"},81),EnabledEnterSubchart({"A","B″},13,2),NotEnabled},
[sem2_advance_step_chosen__4]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(4,s0,a0),**false**)),**false**) =
EnabledEnterSubchart({″A","B″},13,2),
[sem2_state_____5]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(5,s0,a0),**false**))) =
⟨4,6,3,1⟩,
[sem2_advance_steps_____5]
ads(test_machine(wfl_mainch, mk_gStep(5,s0,a0),**false**)) =
{EnabledMessage({″C″},{"D"},81),EnabledCondition({"A","B"},"Cond2″,22),NotEnabled},
[sem2_advance_step_chosen__5]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(5,s0,a0),**false**)),**false**) =
EnabledCondition({″A","B"},"Cond2″,22),
[sem2_state_____6]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(6, s0, a0),**false**))) =
⟨5,7,3,1⟩,
[sem2_advance_steps_____6]
ads(test_machine(wfl_mainch, mk_gStep(6,s0,a0),**false**)) =
{NotEnabled,EnabledMessage({″A"},{"B"},61),EnabledMessage({"C"},{"D″},81)},
[sem2_advance_step_chosen__6]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(6,s0,a0),**false**)),**false**) =
EnabledMessage({″A"},{"B″},61),
[sem2_state_____7]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(7,s0,a0),**false**))) =
⟨6,8,3,1⟩,
[sem2_advance_steps_____7]
ads(test_machine(wfl_mainch, mk_gStep(7,s0,a0),**false**)) =
{EnabledMessage({″C"},{"D"},81),EnabledEndSubchart(13,{"A","B″}),NotEnabled},
[sem2_advance_step_chosen__7]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(7,s0,a0),**false**)),**false**) =
EnabledEndSubchart(13,{″A","B″}),
[sem2_state_____8]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(8, s0, a0),**false**))) =
⟨4,6,3,1⟩,
[sem2_advance_steps_____8]
ads(test_machine(wfl_mainch, mk_gStep(8,s0,a0),**false**)) =
{EnabledMessage({″C"},{"D"},81),EnabledCondition({"A","B"},"Cond2″,22),NotEnabled},
[sem2_advance_step_chosen__8]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(8,s0,a0),**false**)),**false**) =
EnabledCondition({″A","B"},"Cond2″,22),
[sem2_state_____9]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(9,s0,a0),**false**))) =
⟨5,7,3,1⟩,
[sem2_advance_steps_____9]
ads(test_machine(wfl_mainch, mk_gStep(9,s0,a0),**false**)) =
{NotEnabled,EnabledMessage({″A"},{"B"},61),EnabledMessage({"C"},{"D″},81)},
[sem2_advance_step_chosen__9]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(9,s0,a0),**false**)),**false**) =
EnabledMessage({″A"},{"B″},61),
[sem2_state_____10]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(10, s0, a0),**false**))) =
⟨6,8,3,1⟩,
[sem2_advance_steps_____10]
ads(test_machine(wfl_mainch, mk_gStep(10,s0,a0),**false**)) =
{EnabledMessage({″C"},{"D"},81),EnabledEndSubchart(13,{"A","B″}),NotEnabled},
[sem2_advance_step_chosen_10]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(10,s0,a0),**false**)),**false**) =

EnabledEndSubchart(13,{″A","B″}),
[sem2_state_____11]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(11, s0, a0),**false**))) =
⟨7,9,3,1⟩,
[sem2_advance_steps_____11]
ads(test_machine(wfl_mainch, mk_gStep(11,s0,a0),**false**)) =
{NotEnabled,EnabledStopped,EnabledMessage({″C"},{"D″},81)},
[sem2_advance_step_chosen_11]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(11,s0,a0),**false**)),**false**) =
EnabledMessage({″C"},{"D″},81),
[sem2_state_____12]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(12, s0, a0),**false**))) =
⟨7,9,4,2⟩,
[sem2_advance_steps_____12]
ads(test_machine(wfl_mainch, mk_gStep(12,s0,a0),**false**)) =
{EnabledCoregion({″C"},{"B"},91,5),NotEnabled,EnabledStopped,EnabledCoregion({"B″},
{″C"},71,5),EnabledMessage({"D″},{},51)},
[sem2_advance_step_chosen_12]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(12,s0,a0),**false**)),**false**) =
EnabledCoregion({″B"},{"C″},71,5),
[sem2_state_____13]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(13, s0, a0),**false**))) =
⟨7,9,4,2⟩,
[sem2_advance_steps_____13]
ads(test_machine(wfl_mainch, mk_gStep(13,s0,a0),**false**)) =
{EnabledCoregion({″C"},{"B"},91,8),EnabledStopped,NotEnabled,EnabledMessage({"D″},{},51)},
[sem2_advance_step_chosen_13]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(13,s0,a0),**false**)),**false**) =
EnabledMessage({″D"},{},51),
[sem2_state_____14]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(14, s0, a0),**false**))) =
⟨7,9,4,3⟩,
[sem2_advance_steps_____14]
ads(test_machine(wfl_mainch, mk_gStep(14,s0,a0),**false**)) =
{NotEnabled,EnabledStopped,EnabledCoregion({″C"},{"B″},91,8)},
[sem2_advance_step_chosen_14]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(14,s0,a0),**false**)),**false**) =
EnabledCoregion({″C"},{"B″},91,8),
[sem2_state_____15]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(15, s0, a0),**false**))) =
⟨7,12,7,3⟩,
[sem2_advance_steps_____15]
ads(test_machine(wfl_mainch, mk_gStep(15,s0,a0),**false**)) =
{EnabledStopped},
[sem2_advance_step_chosen_15]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(15,s0,a0),**false**)),**false**) =
EnabledStopped,
[sem2_state_____16]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(16, s0, a0),**false**))) =
⟨7,12,7,3⟩,
[sem2_advance_steps_____16]
ads(test_machine(wfl_mainch, mk_gStep(16,s0,a0),**false**)) =
{EnabledStopped},
[sem2_advance_step_chosen_16]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(16,s0,a0),**false**)),**false**) =
EnabledStopped,
[sem2_state_____17]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(17, s0, a0),**false**))) =

⟨7,12,7,3⟩,
[sem2_advance_steps_____17]
ads(test_machine(wfl_mainch, mk_gStep(17,s0,a0),**false**)) =
{EnabledStopped},
[sem2_advance_step_chosen_17]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(17,s0,a0),**false**)),**false**) =
EnabledStopped,
[sem2_state_____18]
sem_test_convert_state(gstate(test_machine(wfl_mainch,mk_gStep(18, s0, a0),**false**))) =
⟨7,12,7,3⟩,
[sem2_advance_steps_____18]
ads(test_machine(wfl_mainch, mk_gStep(18,s0,a0),**false**)) =
{EnabledStopped},
[sem2_advance_step_chosen_18]
sem_test_get_adv(ads(test_machine(wfl_mainch,mk_gStep(18,s0,a0),**false**)),**false**) =
EnabledStopped
value

This function is used to test the main semantics function indirectly. It performs the same steps as the semantics function by calling eval_traces. This is done since it is not feasible to compute all the traces since the number of traces increases exponentially. Therefore only the traces up to a length of 6 are computed (se test_case section).

There is

testtracemachine : Chart × Traces × **Int** → Traces
testtracemachine(chart, traces, counter) ≡
if counter = 0 **then** traces **else**
if (**card** traces) > 1 **then**
let
el1 = **hd** traces,
el2 = **hd** (traces \ {el1}),
newt = {el1, el2}
in
--XX change traces/newt to newt/traces for smaller/bigger
-- thingy (el1 exchanged for traces
testtracemachine(wfl_mainch, eval_traces(chart,traces), counter−1)
end
else
let
el1 = **hd** traces,
newt = {el1}
in
testtracemachine(wfl_mainch, eval_traces(chart, traces), counter−1)
end
end
end,

The initial trace only consisting of the start state.

starttrace: Trace-**set** = {⟨initialize_chart(wfl_mainch)⟩}

test_case
[sem_traces_____0]
starttrace
={⟨["D"+>mk_PosInfo(1,None,<..>),"C"↦mk_PosInfo(1,None,⟨⟩),
"B"+>mk_PosInfo(1,CoRegion({41,31}),<..>),"A"↦mk_PosInfo(1,None,⟨⟩)]⟩},
[sem_traces_____1]
sem_test_convert_traces(testtracemachine(wfl_mainch,starttrace,1))=
{⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩⟩},
[sem_traces_____2]
sem_test_convert_traces(testtracemachine(wfl_mainch,starttrace,2))=
{⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩⟩},
[sem_traces_____3]
sem_test_convert_traces(testtracemachine(wfl_mainch,starttrace,3))=
{⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩⟩},
[sem_traces_____4]
sem_test_convert_traces(testtracemachine(wfl_mainch,starttrace,4))=
{⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩⟩},
[sem_traces_____5]
sem_test_convert_traces(testtracemachine(wfl_mainch,starttrace,5))=
{⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨3,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨3,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨2,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨2,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩⟩},
[sem_traces_____6]
sem_test_convert_traces(testtracemachine(wfl_mainch,starttrace,6)) =
{⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨3,5,4,2⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨3,5,4,2⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨3,5,4,3⟩⟩,

⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨3,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨3,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨2,5,4,3⟩,⟨3,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨5,7,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨5,7,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨2,5,4,3⟩,⟨3,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨4,6,4,2⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨5,7,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨5,7,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨3,5,3,1⟩,⟨4,6,3,1⟩,⟨5,7,3,1⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨3,5,4,2⟩,⟨3,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨2,4,2,1⟩,⟨2,5,3,1⟩,⟨2,5,4,2⟩,⟨3,5,4,2⟩,⟨3,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨2,1,1,1⟩,⟨3,1,1,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨3,5,4,3⟩⟩,
⟨⟨1,1,1,1⟩,⟨1,1,2,1⟩,⟨2,4,2,1⟩,⟨3,4,2,1⟩,⟨3,5,3,1⟩,⟨3,5,4,2⟩,⟨3,5,4,3⟩⟩}

end - - class end

D.4.3 Test of collections

RSC_test2

scheme RSC_test3 =
extend RSC_test2 **with**
class

Test of a collection.

value
col_m1out : Event = mk_OutputEvent(11, "Msg1", mk_Address("B")),
col_m1in : Event = mk_InputEvent(11, mk_Address("A")),
col_m2out : Event = mk_OutputEvent(21, "Msg2", mk_Address("C")),
col_m2in : Event = mk_InputEvent(21, mk_Address("B")),

col_m22out : Event = mk_OutputEvent(21, "Msg2", mk_Address("B")),
col_m22in : Event = mk_InputEvent(21, mk_Address("C")),

col_m3out : Event = mk_OutputEvent(31, "Msg3", mk_Address("A")),
col_m3in : Event = mk_InputEvent(31, mk_Address("B")),
col_m4out : Event = mk_OutputEvent(41, "Msg4", mk_Address("D")),
col_m4in : Event = mk_InputEvent(41, mk_Address("C")),
col_m5out : Event = mk_OutputEvent(51, "Msg5", mk_Address("C")),
col_m5in : Event = mk_InputEvent(51, mk_Address("D")),

col_c1 : Event = mk_ConditionEvent("Cond1", 12, Hot, {"B","C"}),

col_mch1_a : Location* =
⟨mk_Location(Hot, col_m1out), mk_Location(Hot, col_m3in),mk_Location(Hot,StopEvent)⟩,
col_mch1_b : Location* =
⟨mk_Location(Hot, col_m1in),

```
mk_Location(Hot,col_c1),mk_Location(Hot,
col_m2out),mk_Location(Hot,col_m3out),mk_Location(Hot,StopEvent)⟩,
        col_mch1_c : Location* =
            ⟨mk_Location(Hot, col_c1), mk_Location(Hot, col_m2in),mk_Location(Hot,StopEvent)⟩,
        col_mch1 : Chart = [″A" +> col_mch1_a,"B" +> col_mch1_b, "C″ ↦
        col_mch1_c ],

        col_pch2_b : Location* =
            ⟨mk_Location(Hot, col_c1), mk_Location(Hot, col_m2out),mk_Location(Hot,StopEvent)⟩,
        col_pch2_c : Location* =
            ⟨mk_Location(Hot, col_c1), mk_Location(Hot,col_m2in),mk_Location(Hot,StopEvent)⟩,
        col_pch2 : Chart = [″B" +> col_pch2_b,"C″ ↦ col_pch2_c ],

        col_mch2_c : Location* =
            ⟨mk_Location(Hot, col_m4out), mk_Location(Hot, col_m5in),mk_Location(Hot,StopEvent)⟩,
        col_mch2_d : Location* =
            ⟨mk_Location(Hot, col_m4in),
mk_Location(Hot,col_m5out),mk_Location(Hot,StopEvent)⟩,
        col_mch2 : Chart = [″C" +> col_mch2_c,"D″ ↦ col_mch2_d ],

        col_lsc1 : RSC = mk_RSC′(″RSC1″, [ ], col_mch1, {}),
        col_lsc2 : RSC = mk_RSC′(″RSC2″, col_pch2, col_mch2, {}),

        col_collsc1 : ColRSC = mk_act(col_lsc1,initialize_chart(mainchart(col_lsc1))),
        col_collsc2 : ColRSC = mk_not_act(col_lsc2,{}),

        col_collection : Collection = [″RSC1" +> col_collsc1, "RSC2″ ↦ col_collsc2 ],


  print_states : Collection → (Inst_Name × Int*)-set print_states(col) is if col = [ ] then {} else let el = hd
col in {case col(el) of - - mk_act(_, state) -> (el, convert_state(state)), mk_not_act(lsc64, state2) → (el, ⟨⟩),
mk_done(_) → (el, ⟨⟩) end} union print_states(col \ {el}) end end,


        tm_collection : Collection × Int → Collection
        tm_collection(col, counter) ≡
            if counter = 0 then col else tm_collection(execute_collection(col), counter − 1) end ,

        test_func : Int → (Int × Int)
        test_func(i) ≡
        (i+1, i+2)

    test_case
        [col_wf_mch1_____________]
            wf_chart(col_mch1),
        [col_wf_pch2_____________]
            wf_chart(col_pch2),
        [col_wf_mch2_____________]
            wf_chart(col_mch2),
        [col_wf_lsc1_____________]
            wf_RSC(col_lsc1),
        [col_wf_lsc2_____________]
            wf_RSC(col_lsc2),
        [col_col_wf_1_____________]
             wf_col_active(col_collection),
        [col_col_wf_2_____________]
             wf_col_domain(col_collection),
        [col_col_wf_3_____________]
```

wf_col_consistent(col_collection)

[wf_lsc] wf_chart(tmc2), [wf_collection] wf_collection(testcol), [execute_collection28] print_states(tm_collection(testcol
25))

[events_lsc] events_lsc(testlsc), [lsc_ids] lsc_ids(testlsc)

end - -class end

D.5 CSP and LSC

D.5.1 Example 1

object types :
class

Initial LSC types.

type

LSC_Type == Existential | Universal,

Location :: temp : HotCold event : Event,
Event =
ActionEvent | MessageEvent | ConditionEvent | CoregionEvent |
Subchart | EndSubchart, - - | SendStartMessage |
- - receiveStartMessage, – |
- - TimerEvent,
ActionEvent,
MessageEvent ==
- - input message giving id of message to be received
mk_InputEvent(inchannelid : **Nat**) |
- - output message giving id of message to be sent and message
- - to be sent
mk_OutputEvent(outchannelid : **Nat**,
outpar : Message),

- - for each condition create a separate process which handles
- - condition evaluation and uses channel "cid"
ConditionEvent ::
- - name of condition
- - conname : Text
- - Condition ID
cid : **Nat**
- - current instance′s unique number (from 1 to total number of
- - instances sharing the condition)
no : **Nat**
- - temperature of condition
temp : HotCold,
- - set of instance names sharing the condition

```
--share : Text-set,

          CoregionEvent :: locl : Location*,
          Subchart ::
              -- subchart id
              scid : Nat
-- number of locations (excluding mk_subchart, including mk_endsubchart
length : Nat
-- list of instance IDs that indicate which instances are
-- participating in the subchart
share : Nat*,

          EndSubchart :: scid : Nat,
          -- scid not really necessary, but convenient when reading RSL specification

          -- temperature, general
          HotCold == Hot | Cold,

          -- abstract data sort
          Data,

          -- message consisting of name and data
          Message :: outpid: Text outpar : Data,

          State ::
          -- current position on instance
pos     : Int ↔ incr
          -- list of positions of subchart beginnings
subbeg  : Int*
          -- list of positions of subchart endings
subend  : Int*
          -- list of subchart ids
scid    : Int*
          -- list of subcharts sharings
sharel  : (Nat*)*
          -- indicator for showing whether instance is in cold
          -- (location-wise) region
opt     : Bool ↔ change

    end
```

A boolean channel.

```
scheme bool_chnl = class channel ch : Bool end
```

An int channel.

```
scheme int_chnl = class channel ch : Int end
```

types

An interrupt channel.

scheme irq_chnl = **class channel** ch : types.HotCold × **Nat*** × **Nat end**

A unit channel.

scheme unit_chnl = **class channel** ch : **Unit end**

types

Abstraction of underlying statechart.

scheme StateChart = **class channel** evt : types.Message **end**

types

Example instance A.

```
scheme S_A =
  class
    value
      name : Text = ″A″,
      id : Nat = 1,
      length : Nat = 4,
      tempData1 : types.Data,
      tempData2 : types.Data,

      Loc1 : types.Location = types.mk_Location(types.Hot,
    types.mk_OutputEvent(1,
  types.mk_Message(″Msg1″, tempData1))),

      Loc2 : types.Location = types.mk_Location(types.Hot,
    types.mk_Subchart(1,2,⟨1,2⟩)),
- - next to locations
      Loc3 : types.Location = types.mk_Location(types.Hot,
    types.mk_OutputEvent(
      3,
  types.mk_Message(″Msg3″,tempData2))),
      Loc4 : types.Location = types.mk_Location(types.Hot,
    types.mk_EndSubchart(1)),

      lsclocl : types.Location* = ⟨Loc1, Loc2, Loc3, Loc4⟩

  end
```

types

Example instance B.

```
scheme S_B =
    class
        value
            name : Text = "B",
            id : Nat = 2,
            length : Nat = 7,

            Loc1 : types.Location = types.mk_Location(types.Hot,
    types.mk_CoregionEvent(⟨Loc2, Loc3⟩)),

            Loc2 : types.Location = types.mk_Location(types.Hot,
    types.mk_InputEvent(1)),

            Loc3 : types.Location = types.mk_Location(types.Hot,
    types.mk_InputEvent(2)),

            Loc4 : types.Location = types.mk_Location(types.Hot,
    types.mk_ConditionEvent(1,1,types.Hot)),

            Loc5 : types.Location = types.mk_Location(types.Hot,
      types.mk_Subchart(1,2,⟨1,2⟩)),

            Loc6 : types.Location = types.mk_Location(types.Hot,
    types.mk_InputEvent(3)),

            Loc7 : types.Location = types.mk_Location(types.Hot,
    types.mk_EndSubchart(1)),

            lsclocl : types.Location* = ⟨Loc1, Loc2, Loc3, Loc4, Loc5,
            Loc6, Loc7⟩

    end


types
```

Example instance C.

```
scheme S_C =
    class
        value
            name : Text = "C",
            id : Nat = 3,
            length : Nat = 2,
            tempdata : types.Data,

            Loc1 : types.Location = types.mk_Location(types.Hot,
      types.mk_OutputEvent(2,
    types.mk_Message("Msg2", tempdata))),

            Loc2 : types.Location = types.mk_Location(types.Hot,
    types.mk_ConditionEvent(1,2,types.Hot)),
```

```
        lsclocl : types.Location* = ⟨Loc1, Loc2⟩

  end
```

A generic channel.

```
scheme chscheme(Data : class type Elem end) = class channel ch : Data.Elem end
```

bool_chnl, unit_chnl

A condition process scheme. Mst be instantiated for each condition.

```
scheme condition_p(

      -- receive channels
          sc_ch[ n:Nat, m:Nat ] : unit_chnl,

      -- result channels
          rc_ch[ n:Nat, m:Nat ] : bool_chnl,

          c_no : class
      value
   -- condition id (unique for each condition)
   condID : Nat,
   -- number of instances sharing the condition
   inst_no : Nat,
   -- predicate used to check if condition is met
   test_condition : Unit → Bool
     end
) =
          class
  value
    start : Unit → in {sc_ch[ n,m ].ch | n : Nat, m: Nat} out {rc_ch[ n,m ].ch | n : Nat, m: Nat} Unit
    start() ≡
      sync_cond(c_no.condID, 1);
send_result(c_no.condID,1, c_no.test_condition());
-- called recursively since condition may be inside a
-- iterating subchart
start(),


 -- synchronizes current condition in order to prepare evaluation
    sync_cond : Nat × Nat →
     in {sc_ch[ n,m ].ch | n : Nat, m: Nat}
     Unit
    sync_cond(condID, no) ≡
    if no>c_no.inst_no
    then
     skip
    else
     sc_ch[ condID, no ].ch?;
```

```
      sync_cond(condID, no+1)
    end,

  - - sends the result of the condition to the instances,
  - - thereby allowing them to proceed again
    send_result : Nat × Nat × Bool→
      out {rc_ch[ n,m ].ch | n : Nat, m: Nat}
      Unit
    send_result(condID, no, result) ≡
    if no>c_no.inst_no
    then
      skip
    else
      rc_ch[ condID, no ].ch!result;
      send_result(condID, no+1, result)
    end


            end

irq_chnl, bool_chnl, types
```

An interrupt process that should handle exits from charts.

```
scheme interrupt_p(
- - receive channels
      irqs : irq_chnl,
- - result channels
      irqr[ n:Nat ] : bool_chnl,
      irq_no : class
              value
              - - number of instances (each instance is numbered from 1..)
                   inst_no : Nat
              end
) =
class
     value
          - - waits for interrupts to arrive on the irq channel from instances
          start : Unit → in irqs.ch out {irqr[ n ].ch | n:Nat} Unit
          start() ≡
                  let
  (hot, share, notifier) = irqs.ch?
in
  if hot = types.Hot
  then - - hot condition evaluated to false, main chart
       - - abort, unsuccesful run
     interrupt_all(1, notifier, true)
  else - - cold condition
     if share = ⟨⟩ - - no subchart notify all instances
     then
       interrupt_all(1, notifier, false)
     else  - - a subchart, notify only relevant instances
interrupt_cold_sc(share, notifier)
     end
  end
end,
```

```
    - - interrupts all instances and stops,
        interrupt_all : Nat × Nat × Bool→ out {irqr[ n ].ch |n:Nat } Unit
        interrupt_all(no, notifier, hot) ≡
            if no > irq_no.inst_no
then - - all instances interrupted
  stop
else
  if no = notifier
  then - - instance where interrupt message originated must
- - not be notified
    interrupt_all(no+1, notifier, hot)
  else
    irqr[ no ].ch!hot;
    interrupt_all(no+1, notifier, hot)
  end
end,


    - - interrupts all the instances that are in the same subchart as the
    - - instance where the interrupt originated
        interrupt_cold_sc : Nat* × Nat → in irqs.ch out {irqr[ n ].ch |n:Nat} Unit
        interrupt_cold_sc(shre, notifier) ≡
            if shre = ⟨⟩ - - all stopped, start over
            then
                start()
            else
  if hd shre = notifier
  then
    interrupt_cold_sc(tl shre, notifier)
  else
    irqr[ hd shre ].ch!false;
    interrupt_cold_sc(tl shre, notifier)
  end
end

end - - class


unit_chnl, bool_chnl



Generic subchart process.


scheme subchart_p(

        - - receive channels
            ss_ch[ n:Nat, m:Nat ] : bool_chnl,

        - - feedback channels
            sr_ch[ n:Nat, m:Nat ] : bool_chnl,

            sc_no : class
      value
   - - subchart id (unique for each subchart)
   scID : Nat,
   - - number of instances sharing the subchart
```

```
    inst_no : Nat
variable
    - - number of times the subchart must be repeated
    iterations : Int
      end
) =
              class
  value
    start : Unit → write sc_no.iterations
    in {ss_ch[ n,m ].ch | n : Nat, m: Nat}
    out {sr_ch[ n,m ].ch | n : Nat, m: Nat}
    Unit
    start() ≡
      if sync_sc(sc_no.scID, 1, true)
then
  notify(sc_no.scID, 1, true);
  start()
else
  notify(sc_no.scID, 1, repeat());
- - called recursively
  start()
end,

  - - determines whether a subchart must be repeated (-1
  - - denotes asterisk)
    repeat : Unit → write sc_no.iterations Bool
    repeat() ≡
    if sc_no.iterations > 0 ∨ sc_no.iterations = −1
    then
      if sc_no.iterations = −1
then
  true
else
  sc_no.iterations := sc_no.iterations−1;
  true
end
    else
      false
    end,

  - - synchronizes current condition in order to prepare
  - - evaluation
  - - returns whether it is an entrance or not
    sync_sc : Nat × Nat × Bool →
      in {ss_ch[ n,m ].ch | n : Nat, m: Nat}
      Bool
    sync_sc(scID, no, entering) ≡
    if no>sc_no.inst_no
    then
      entering
    else
    - - synching next instance, ss_ch returns bool which
    - - denotes whether the instances are entering a subchart
    - - or leaving
      let
  entering = ss_ch[ scID, no ].ch?
in
        sync_sc(scID, no+1, entering)
```

```
    end
  end,

 -- sends the result of the condition to the instances,
 -- thereby allowing them to proceed
  notify : Nat × Nat × Bool →
   out {sr_ch[ n,m ].ch | n : Nat, m: Nat}
   Unit
  notify(scID, no, repeat) ≡
  if no>sc_no.inst_no
  then
   skip
  else
   sr_ch[ scID, no ].ch!repeat;
   notify(scID, no+1, repeat)
  end
          end
```

types, chscheme, StateChart, unit_chnl, bool_chnl, irq_chnl

A generic lsc specification.

```
scheme lsc(
    inst :
        class
            value
                lsclocl : types.Location*,
name : Text,
id : Nat,
length : Nat
            -- sc_spec : types.SC_Event-list

        end,
        channels[ id : Nat ] : chscheme(types{Message for Elem}),
        cs_ch[ n : Nat, m : Nat ] : unit_chnl,
        cr_ch[ n : Nat, m : Nat ] : bool_chnl,
        ss_ch[ n : Nat, m : Nat ] : bool_chnl,
        sr_ch[ n : Nat, m : Nat ] : bool_chnl,
        irqs_ch : irq_chnl,
        irqr_ch[ n: Nat ] : bool_chnl

) = with types in
    class

        value

            test : Unit → Unit,

            execute :
                State →
                    in sc.evt,
{channels[ id ].ch| id : Nat},
{cr_ch[ n,m ].ch | n : Nat, m: Nat},
{sr_ch[ n,m ].ch | n : Nat, m: Nat},
{irqr_ch[ n ].ch | n : Nat}
```

```
                    out sc.evt,
{channels[ id ].ch | id : Nat},
{cs_ch[ n,m ].ch | n : Nat, m: Nat},
{ss_ch[ n,m ].ch | n : Nat, m: Nat},
irqs_ch.ch
                    Unit
          execute(state) ≡
          if (pos(state) = inst.length)
          then
              - - after finishing behaviour of LSC all
- - behaviour may be exhibited
skip - - or chaos??
          else
          - - process may be interrupted or proceeds
        - - interrupted(state)
        - - |=|
          let
               interrupt = irqr_ch[ inst.id ].ch?,
               curevent = event(inst.lsclocl(pos(state))),
          - - determines temperature of location
curtemp = temp(inst.lsclocl(pos(state)))
          in
          if interrupt = false then
              if curtemp = Cold ∧ opt(state) = false
              then  - - instance has reached optional behaviour part, may
    - - stop or continue
                    process(change(opt(state) ∧ false, state), curevent ) □
  skip - - or chaos??
              else  - - process is in hot or cold location part and proceeds
 process(state, curevent)
end
          else
              - - process interrupted
if interrupt = true then
- - cold condition interrupt
     if subbeg(state) = ⟨⟩
     then - - there is no subchart when subbeg (and subend,
   - - scid) is empty, therefore exit main chart
   - - (succesful run, however)
 skip - - chaos
     else - - there is a subchart, exit current one
       execute(mk_State((hd subend(state))+1, tl subbeg(state), tl subend(state), tl
 scid(state), tl sharel(state), opt(state)))
     end
else
- - hot condition interrupt
 stop
end
          end
          end
          end,

          process :
                State × Event→
                in sc.evt,
{channels[ id ].ch| id : Nat},
{cr_ch[ n,m ].ch | n : Nat, m: Nat},
{sr_ch[ n,m ].ch | n : Nat, m: Nat},
```

```
{irqr_ch[ n ].ch | n : Nat}
                    out sc.evt,
{channels[ id ].ch | id : Nat},
{cs_ch[ n,m ].ch | n : Nat, m: Nat},
{ss_ch[ n,m ].ch | n : Nat, m: Nat},
irqs_ch.ch
                    Unit
                  process(state, curevent) ≡
                  case curevent of
 - - needs some relation to Statechart here,
 - - do some computation or whatever
 Event_to_ActionEvent(curevent) →


define


 compute();
 execute(incr(pos(state)+1, state)),

 - - sending of a message
                    mk_InputEvent(inmsgid) →
                        let v = channels[ inmsgid ].ch? in
                            sc.evt!v
                        end;
  execute(incr(pos(state)+1, state)),

 - - reception of a message
                    mk_OutputEvent(outmsgid, outpar) →
                    let v = sc.evt? in
                        channels[ outmsgid ].ch!v
                    end;
 execute(incr(pos(state)+1, state)),

 - - condition has several possible behaviours : hot/cold
 - - conditions, is or is not in subchart.
 - - hot condition not permitted, think about semantics in
 - - Come lets play (continuously evaluated) DEFINE
 mk_ConditionEvent(cid, no, temp) →
 - - sync with other instances and get result of condition
 if
   cs_ch[ cid, no ].ch!();
   cr_ch[ cid, no ].ch?
 then - - synchronized and evaluated to true
  execute(incr(pos(state)+1, state))
 else - - synchronized and evaluated to false
  if temp = Hot then - - stop all instances
     irqs_ch.ch!(Hot,⟨⟩, inst.id);    - - false hot condition, may not happen
     stop       - - chaos is not right, because LSC is not fulfilled
  else  - - false cold condition
    if subbeg(state) = ⟨⟩
    then - - there is no subchart when subbeg (and subend,
  - - scid) is empty, therefore exit main chart
irqs_ch.ch!(Cold, ⟨⟩, inst.id);
skip - -chaos
    else - - there is a subchart, interrupt all relevant
  - - instances
irqs_ch.ch!(Cold, hd sharel(state), inst.id);
       execute(mk_State((hd subend(state))+1, tl
```

```
subbeg(state), tl subend(state), tl scid(state),
tl sharel(state), opt(state)))
    end
  end
 end,

 mk_CoregionEvent(locl) →
  execute_coregion(locl, locl);
  execute(incr(pos(state)+(len locl)+1,state)),

 -- synchronizes the entry into subchart
 mk_Subchart(newscid, sublen, shr) →
  let
    index : Nat • shr(index) = inst.id
  in
  -- newsid: subchart id, given above
  -- index: share is list of instance id′s, the channel to
  -- be used matches the corresponding index
  -- true means this is an entry
    ss_ch[ newscid, index ].ch!true;
  -- receive sync signal back, dummy is irrelevant, since no
  -- iteration can take place upon entry of subchart
    let
      dummy  = sr_ch[ newscid, index ].ch?
    in
    -- continuing and storing information about entered subchart
      execute(mk_State(pos(state)+1, ⟨pos(state)⟩^subbeg(state),
⟨(pos(state)+sublen)⟩^subend(state), ⟨newscid⟩^scid(state) , ⟨shr⟩
^sharel(state), opt(state)))
    end
  end,

-- exit from subchart, must synchronize and check whether
-- subchart must be repeated
mk_EndSubchart(curscid) →
  let
    index : Nat • (hd sharel(state))(index) = inst.id
  in
  -- false means this is an exit, might iterate, depending
  -- on feedback
    ss_ch[ hd scid(state), index ].ch!false;
  -- received bool determines whether subchart is to be repeated
    if  sr_ch[ hd scid(state), index ].ch?
    then -- repeat subchart(does not synchronize again at
  -- start of subchart
      execute(mk_State(hd subbeg(state)+1, subbeg(state), subend(state), scid(state), sharel(state),
opt(state)))
    else -- go on and exit subchart
      execute(mk_State((pos(state))+1, tl subbeg(state), tl subend(state), tl scid(state), tl
sharel(state), opt(state)))
    end
  end

-- introduce some kind of TRSL for timers?
-- need to know how to define timers
```

mk_SetTimer(tname, tid, dur) → chaos, mk_StopTimer(stname, stid) → chaos, mk_TimeoutTimer(ttname, ttid) → chaos,

```
            - - _ -> skip

        end, - - case


        - - the message-events in the Location-list must be specified
        - - with external nondeterministic choice
        - - pointer may not be increased
        execute_coregion : types.Location* × types.Location* ~→
                in sc.evt, {channels[ id ].ch| id : Nat}
                out sc.evt, {channels[ id ].ch | id : Nat}
                Unit
        execute_coregion(ll1, ll2) ≡
            if len ll1 = 1
then
  execute_cor_event(hd ll1, ll2)
else
                    execute_cor_event(hd ll1, ll2) [] execute_coregion(tl ll1, ll2)
end
        pre ll1 ≠ ⟨⟩,

        - - work on this one, it is not correct, only one event is
        - - happening
        - - above still valid??
        execute_cor_event : Location × Location* →
                in sc.evt, {channels[ id ].ch| id : Nat}
                out sc.evt, {channels[ id ].ch | id : Nat}
                Unit
        execute_cor_event(loc, locl) ≡
            let
  curevent = event(loc)
in
  case curevent of
   - - sending of a message
                    mk_InputEvent(inmsgid) →
                        let v = channels[ inmsgid ].ch? in
                            sc.evt!v
                        end,

   - - reception of a message
                    mk_OutputEvent(outmsgid, outpar) →
                        let v = sc.evt? in
                            channels[ outmsgid ].ch!v
                        end
  end;
  let
    first : types.Location*, last : types.Location* • first⌢⟨loc⟩⌢last = locl
  in
    if first⌢last = ⟨⟩
    then
      skip
    else
      execute_coregion(first⌢last, first⌢last)
    end
```

```
  end
end,

        compute : Unit → Unit

    object sc : StateChart

  end


types, lsc, S_B, S_A, S_C, int_chnl, unit_chnl, bool_chnl, irq_chnl,
condition_p, subchart_p, interrupt_p
```

Complete example specification consisting of a LSC with the instances A, B and C.

```
scheme system =
  class

- ———— GENERAL PART ————————
    object
      - - channels for communicating between instances
      com_chnls[ id : Nat ] : chscheme(types{Message for Elem}),

      - - channel for synchronizaton of conditions
      cs_ch[ n : Nat, m : Nat ] : unit_chnl,

      - - channels for returning result from condition
      cr_ch[ n : Nat, m : Nat ] : bool_chnl,

      - - channels for synchronizaton of subcharts
      ss_ch[ n : Nat, m : Nat ] : bool_chnl,

      - - channels for synchronizaton of subcharts
      sr_ch[ n : Nat, m : Nat ] : bool_chnl,

      - - channel for sending interrupt
      irqs_ch : irq_chnl,

      - - channel for receving interrupts
      irqr_ch[ n: Nat ] : bool_chnl,

- ———— SPECIALIZED PART ————————-

      - - condition 1 values
      con_no1 : class
   value
   - - name of condition (only given for convenience)
     condname : Text = ″Condition1″,
- - unique id of condition
     condID : Nat = 1,
- - number of instances sharing the condition
inst_no : Nat = 2,
- - predicate for testing whether the condition is true or not
test_condition : Unit → Bool

  end,
```

```
            -- object for synchronizing and evaluating condition 1
            cp1 : condition_p(cs_ch, cr_ch, con_no1),

            -- subchart 1 values
            sc_no1 : class
      value
       -- unique id of subchart
       scID : Nat = 1,
-- number of instances in subchart
inst_no : Nat = 2
      variable
       -- used for counting how often the subchart must be
-- repeated, 0 denotes infinitely (termination can
-- only happen by condition exit)
       iterations : Int := 1
    end,

            -- object for synchronizing subchart 1 entry and exit
            scp1 : subchart_p(ss_ch, sr_ch, sc_no1),

            -- value with number of instances, used by interrupt process
            irq_no : class
                        value
      inst_no : Nat = 3
     end,

            -- object for handling interrupts
            irqp : interrupt_p(irqs_ch, irqr_ch, irq_no),

            -- instances are instantiated
            a : S_A,
            a_lsc : lsc(a, com_chnls, cs_ch, cr_ch, ss_ch, sr_ch, irqs_ch, irqr_ch),
            b : S_B,
            b_lsc : lsc(b, com_chnls, cs_ch, cr_ch, ss_ch, sr_ch, irqs_ch, irqr_ch),
            c : S_C,
            c_lsc : lsc(c, com_chnls, cs_ch, cr_ch, ss_ch, sr_ch, irqs_ch, irqr_ch)

        value
         -- all processes are started in parallel
            system : Unit → write any in any out any Unit
            system() ≡ a_lsc.execute(types.mk_State(0,⟨⟩,⟨⟩,⟨⟩,⟨⟩, false))
     || b_lsc.execute(types.mk_State(0,⟨⟩,⟨⟩,⟨⟩,⟨⟩, false))
     || c_lsc.execute(types.mk_State(0,⟨⟩,⟨⟩,⟨⟩,⟨⟩, false))
     || cp1.start()
     || scp1.start()
     || irqp.start()
    end
```

D.5.2 Example 2

```
scheme lsc =
    class
```

Example LSC where a LSC consists of the parallel composition on instances. Each instance is a series of functions that branch depending on the execution.

```
type
    Com = Msg_com | Sync_com,
    Msg_com = Msg3 | Msg4 | Msg5 | Msg6 | Msg7 | Msg8 | Msg9,
    Msg3,
    Msg4,
    Msg5,
    Msg6,
    Msg7,
    Msg8,
    Msg9,
    Sync_com = Bool

channel
    a_b : Com,
    b_a : Com,
    b_c : Com,
    c_b : Com,
    c_d : Com,
    d_c : Com,
    d_env : Com,
    a_cond2 : Bool,
    b_cond2 : Bool,
    b_cond1 : Bool,
    c_cond1 : Bool,
    a_sub1 : Bool,
    b_sub1 : Bool

value
    msg3 : Msg3,
    msg4 : Msg4,
    msg5 : Msg5,
    msg6 : Msg6,
    msg7 : Msg7,
    msg8 : Msg8,
    msg9 : Msg9,

    system : Unit → in any out any Unit
    system() ≡ lsc1(),

    lsc1 : Unit → in any out any Unit
    lsc1() ≡ inst_a1() ‖ inst_b() ‖ inst_c() ‖ inst_d(),


    inst_a1 : Unit → in any out any Unit
    inst_a1() ≡
        a_b!msg3 ;
        action1() ;
        a_sub1!true ;
        while let continue = a_sub1? in continue end do
            if a_cond2!true ; let continue = a_cond2? in continue end then a_b!msg6 ; a_sub1!true
            else inst_a2()
            end
        end ;
        inst_a2(),
```

```
inst_a2 : Unit → in any out any Unit
inst_a2() ≡ chaos,

inst_b : Unit → in any out any Unit
inst_b() ≡
    (let rmsg3 = a_b? in skip end ⌈⌉ let rmsg4 = c_b? in skip end) ;
    if b_cond1!true ; let continue = b_cond1? in continue end
    then
        b_sub1!true ;
        while let continue = b_cond1? in continue end do
            let rmsg6 = a_b? in skip end ; b_sub1!true
        end ;
        b_c!msg7
        ⌈⌉
        let rmsg9 = c_b? in skip end
    else stop
    end ;
    chaos,

inst_c : Unit → in any out any Unit
inst_c() ≡
    c_b!msg4 ;
    if c_cond1!true ; let continue = c_cond1? in continue end
    then let rmsg7 = b_c? in skip end ⌈⌉ c_b!msg9 ; let rmsg7 = b_c? in skip end ⌈⌉ c_b!msg9
    else stop
    end ;
    chaos,

inst_d : Unit → in any out any Unit
inst_d() ≡ let rmsg8 = c_d? in skip end ; d_env!msg5 ; stop,

cond1 : Unit → in b_cond1, c_cond1 out b_cond1, c_cond1 Unit
cond1() ≡
    let dummy = b_cond1? in skip end
    ⌈⌉
    let dummy = b_cond1? in skip end ; let dummy = b_cond1? in skip end
    ⌈⌉
    let dummy = b_cond1? in skip end ;
    let continue = condition1() in (b_cond1!continue ‖ c_cond1!continue) end,

sub1 : Unit → in a_sub1, b_sub1 out a_sub1, b_sub1 Unit
sub1() ≡
    while let repeat = repeat_sub1() in repeat end do
        let dummy = a_sub1? in skip end
        ⌈⌉
        let dummy = b_sub1? in skip end ; let dummy = a_sub1? in skip end
        ⌈⌉
        let dummy = b_sub1? in skip end ; (a_sub1!true ‖ b_sub1!true)
    end ;
    (a_sub1!false ‖ b_sub1!false),

- ———— functions ————————
```

condition 1 MUST be true

```
condition1 : Unit → Bool
```

```
        condition1() ≡ true,

        condition2 : Unit → Bool,
        action1 : Unit → Unit,
        repeat_sub1 : Unit → Bool

    end
```

D.6 Applicative RSC

D.6.1 Types

../RSC_semantics

```
scheme rsc_types =
    extend RSC_semantics with
    class
        type
            Collection = RSC-set
        type
            SysTrace = SysEvent^ω,
            SysEvent′ ==
                mk_SysAction(instance : Inst_Name, action : Text) |
                mk_SysMsg(src : Inst_Name-set, dst : Inst_Name-set, method : Text) |
                mk_SysCondition(shared_by : Inst_Name-set, cond : Text),
            SysEvent =
                {| se : SysEvent′ •
                    case se of
                        mk_SysMsg(src, dst, _) →
                            let n = card src + card dst in
                                n = 1 ∨ n = 2
                            end,
                        _ → true
                    end |}

        type
            Variables = VariableName →m Value,
            VariableName = Text,
            Value == Boolean(boolean: Bool) | Integer(integer: Int) | String(string: Text)

        type
            Condition = Inst_Name-set × Variables → Bool,
            Action = Inst_Name × Variables → Variables,
```

Src and dst included in signature. Src because a function that assignes a variable from src in dst needs this information. If the set is empty it denotes the environment.

```
            Message =
                Inst_Name-set × Inst_Name-set × Variables →
                    Variables
    end
```

D.6.2 Type object

rsc_types

An object for all the types needed.

object T : rsc_types

D.6.3 Semantics

T, specification

scheme semantics(SPEC : specification) =
  class
    value

The system trace that defines the events that happen in the system.

st : T.SysTrace,

The initial variables at the beginning of the system run.

v : T.Variables,

The list of variables in the system. The first element are the initial variables. The next element are the variables after performing the first system event of the system trace. And so on. The variables are the state of the system. This also implies that there is one more element in the variables list than in the system trace.

vl : T.Variables* = make_var_trace(st, v)

The following axioms say that the system trace and variables list must conform to the constraints imposed by all the \rsc′s that are specified. One is for \rsc′s without a prechart one for \rsc′s with a prechart.

axiom
  [valid_trace_no_prechart]
    ∀ rsc : T.RSC, vv : T.VariableName-**set** •
      rsc ∈ SPEC.rscs ∧ vv = SPEC.visible_variables(rsc) ∧ T.prechart(rsc) = [ ] ⇒
        valid_trace_nopre(rsc, st, vl),
  [valid_trace_with_prechart]
    ∀ rsc : T.RSC, vv : T.VariableName-**set** •
      rsc ∈ SPEC.rscs ∧ vv = SPEC.visible_variables(rsc) ∧T.prechart(rsc) ≠ [ ] ⇒
        valid_trace(rsc, st, vl, vv)

valid_trace checks if a system trace and list of variables is valid with a single RSC as constraint.

```
value
  valid_trace :
    T.RSC × T.SysTrace × T.Variables* × T.VariableName-set ~→ Bool
  valid_trace(rsc, t, vl, vv) ≡
    (∀ i,j, k : Int •
      {i, j, k} ⊆ inds t ∧ i ≤ j ∧ j < k ∧
      let
```

The prechart trace is not concerned with visible events, thus the lambda function. A prechart trace is simply a trace that conforms to the prechart specification.

```
      pre_trace =
        sub_trace(t, vl)(
            i, j − 1, λ e : T.SysEvent • true),
```

The mainchart trace is only concerned with constraining events that are visible in the mainchart, i.e. in our case events that occur somewhere in the mainchart. These events are filtered using the predicate visible_event.

```
      main_trace =
        sub_trace(t, vl)(
            j, k, visible_event(T.mainchart(rsc)))
    in
```

The essential part: if a trace that satisfies the prechart is observed, we MUST also observe a trace that satisfies the mainchart.

```
      valid_chart(rsc, T.prechart(rsc))(pre_trace) ⇒
        valid_chart(rsc, T.mainchart(rsc))(main_trace)
    end)
  pre len t + 1 = len vl ∧ T.prechart(rsc) ≠ [ ],
```

valid_trace for \rsc′s with no prechart. No prechart means that the ordering imposed by the mainchart must always be fulfilled.

```
  -- Er der et problem med visible events og index
  -- på et trace med
  -- visible events?
  valid_trace_nopre :
    T.RSC × T.SysTrace × T.Variables* ~→ Bool
  valid_trace_nopre(rsc, t, vl) ≡
    (∃ ilist : Nat* •
      len ilist ≥ 2 ∧
ilist(1) = 1 ∧
      ilist(len ilist) = len t ∧
(∀ i : Int • i ∈ inds ilist \ {len ilist} ⇒
```

If the systemtrace (t) is empty the following will not hold and false will be returned.

```
  ilist(i) ≤ ilist(i+1) ∧
```

It is checked that a list of indexes exist, where the mainchart is satisfied, thus implying that the mainchart always will hold due to the constraints on the indexes given above.

```
                valid_chart(rsc, T.mainchart(rsc))(
                    sub_trace(t,vl)(ilist(i), ilist(i + 1),
                        visible_event(T.mainchart(rsc))))))
        pre T.prechart(rsc) = [ ],
```

sub_trace finds a subtrace of a systrace and variables list in an interval using a predicate on each element.

```
        sub_trace :
            T.SysTrace × T.Variables^ω →
                Int × Int × (T.SysEvent → Bool) →
                    T.SysTrace × T.Variables^ω
        sub_trace(t, vl)(i, j, p) ≡
            let
```

Indices for events that are visible.

```
  idxl1 = ⟨n | n in ⟨i .. j⟩ • p(t(n))⟩,
```

Indices for variables. For each state the variables before and after are included.

```
  idxl2 = variable_indices(idxl1)
in
                (⟨t(n) | n in idxl1⟩, ⟨vl(n) | n in idxl2⟩)
            end,
```

variable_indices returns a list AB as AaBa where a = A+1, b = B+1 etc. Is used for determining indices of variables that are needed.

```
        variable_indices : Int* → Int*
        variable_indices(idxl) ≡
        if idxl = ⟨⟩ then ⟨⟩
        else
⟨hd idxl, hd idxl+1⟩^variable_indices(tl idxl)
        end,
```

valid_chart checks if a system trace and variables list is valid with regards to the constraints imposed by the given chart.

```
        valid_chart :
            T.RSC × T.Chart →
                T.SysTrace × T.Variables^ω ~→ Bool
        valid_chart(rsc, chart)(t, vl) ≡
```

```
        valid_chart(rsc, chart)(t, vl)(T.initialize_chart(chart))(hd vl),
```

Same as above, used for recursion.

```
    valid_chart :
        T.RSC × T.Chart →
            T.SysTrace × T.Variables^ω →
                T.State ~→
T.Variables → Bool
    valid_chart(rsc, chart)(t, vl)(state)(old_var) ≡
        if state = T.end_state(chart) then t = ⟨⟩
        else
            let
                (ok, state′) =
                    expected_event(chart)(state, hd t, hd vl)
            in
```

It is checked that the current system event (hd t) is actually an expected event in the chart with the given state. Repeated for the tail of the system trace t. If hd t at any point is not expected, the system does not conform to the constraints imposed by the chart.

```
                ok ∧ valid_chart(rsc, chart)(tl t, tl tl vl)(state′)(hd tl vl) ∧
```

Checking if the visible events before this event are the same as they where after the last visible event. This is to ensure that no invisible events have changed visible variables.

```
vis_var_ok(SPEC.visible_variables(rsc), old_var, hd vl)
            end
        end

    type
```

A set of EnabledEvents.

```
        EES = T.EnabledEvent-set,
```

A subtype of EES. Used for returning a result, where a maximum of 1 element is returned.

```
        EE = {| ees : EES • card ees ≤ 1 |}

    value
```

Checks if two sets contain the same variables with the same value for a given variable name set.

```
        vis_var_ok : Text-set × T.Variables × T.Variables → Bool
        vis_var_ok(vnames, old_var, cur_var) ≡
        (∀ n : Text • n ∈ vnames ⇒
n ∈ dom old_var ∧
```

```
  n ∈ dom cur_var ∧
  old_var(n) = cur_var(n)),
```

expected_event checks wether the given SysEvent is possible with the given chart, state and variable set. It also returns the new state in the RSC after performing the EnabledEvent that matches the SysEvent (or the current state if false).

```
expected_event :
    T.Chart →
        T.State × T.SysEvent × T.Variables →
            Bool × T.State
expected_event(chart)(state, event, var_before) ≡
    let
        state′ = perform_subchart_events(chart)(state),
        ees = T.get_enabled_events(chart, state′),
        exp_es =
            case event of
                T.mk_SysAction(_) →
                    expected_action(event, ees),
                T.mk_SysCondition(_) →
                    expected_condition(event, ees, var_before),
                T.mk_SysMsg(_, _, _) →
                    expected_message(event, ees)
            end
    in
```

No matching EnabledEvent found.

```
        if exp_es = {} then (false, state)
```

Matching EnabledEvent found. The charts state is advanced using the matching EnabledEvent.

```
        else (true, step_event(chart, state, hd exp_es))
        end

    end,
```

expected_action checks if an equivalent EnabledEvent of a action system event is present in the given EnabledEvent-set. If found, the EnabledEvent is returned.

```
expected_action : T.SysEvent × EES ~→ EE
expected_action(event, ees) ≡
    ee_subset(ees, action_match(event))
pre is_action(event),
```

action_match checks wether an EnabledEvent is an equivalent of the given system event.

```
action_match : T.SysEvent → T.EnabledEvent ~→ Bool
action_match(event)(ee) ≡
```

```
    (∃ id : T.ID •
        T.EnabledAction(
            T.instance(event), T.action(event), id) = ee)
pre is_action(event),
```

expected_condition checks if an equivalent EnabledEvent of a condition system event is present in the given EnabledEvent-set. If found, the EnabledEvent is returned.

```
expected_condition :
    T.SysEvent × EES × T.Variables ~→ EE
expected_condition(event, ees, vars) ≡
    ee_subset(ees, condition_match(event, vars))
pre is_condition(event),
```

condtion_match checks wether an EnabledEvent is an equivalent of the given system event. It must also consider the variables in order to check if the EnabledEvent matches with the result of the condition predicate.

```
condition_match :
    T.SysEvent × T.Variables →
        T.EnabledEvent ~→ Bool
condition_match(event, vars)(ee) ≡
```

The following is a match if the condition predicate returns true.

```
    (∃ id : T.ID •
        (T.EnabledCondition(
            T.shared_by(event), T.cond(event), id) = ee ∧
          SPEC.conditions(T.cond(event))(
            T.shared_by(event), vars))) ∨
```

If the condition predicate return false, there are two options wrt. EnabledEvents: an EnabledExitSubchart, which denotes a false cold condition within a subchart, and an EnabledExitMainchart which denotes a false cold condtion in a mainchart.

```
    (∃
```

action_match checks if a action SysEvent matches the given EnabledEvent.

```
        id : T.ID, instns : T.Inst_Name-set •
        (T.EnabledExitSubchart(
            instns, id, T.shared_by(event), T.cond(event)) =
          ee ∧
          ~ SPEC.conditions(T.cond(event))(
            T.shared_by(event), vars))) ∨
    (T.EnabledExitMainchart(
        T.shared_by(event), T.cond(event)) = ee ∧
     ~ SPEC.conditions(T.cond(event))(
        T.shared_by(event), vars))
pre is_condition(event),
```

expected_message checks if an equivalent EnabledEvent of a message system event is present in the given EnabledEvent-set. If found, the EnabledEvent is returned.

```
expected_message : T.SysEvent × EES ~→ EE
expected_message(event, ees) ≡
    ee_subset(ees, message_match(event))
pre is_message(event),
```

message_match checks wether an EnabledEvent is an equivalent of the given system event. There are two possibilites: an EnabledMessage which denotes a regular message and an EnabledCoregion which denotes an enabled message within a coregion.

```
message_match :
    T.SysEvent
    → T.EnabledEvent ~→ Bool
message_match(event)(ee) ≡
    (∃ id : T.ID •
        T.EnabledMessage(T.src(event), T.dst(event), id) =
            ee) ∨
    (∃ id : T.ID, i : Int •
        T.EnabledCoregion(
                T.src(event), T.dst(event), id, i) = ee)
pre is_message(event),
```

ee_subset returns a subset of the given EnabledEvent set with elements where the given predicate holds.

```
ee_subset : EES × (T.EnabledEvent → Bool) → EE
ee_subset(ees, p) ≡
    {ee | ee : T.EnabledEvent • ee ∈ ees ∧ p(ee)}

value
```

make_var_trace generates a list of variables given a system trace and the initial variables.

```
make_var_trace :
    T.SysTrace × T.Variables → T.Variables^ω
make_var_trace(t, v) ≡
    if t = ⟨⟩ then ⟨v⟩
    else
        let v′ = variable_step(hd t, v) in
            ⟨v⟩ ⌢ make_var_trace(tl t, v′)
        end
    end,
```

variable_step calculates the resulting variables given some initial variables and an system event that occurs. Only actions and messages modify the variables as conditions only read them.

```
variable_step :
    T.SysEvent × T.Variables → T.Variables
variable_step(e, v) ≡
```

```
        case e of
            T.mk_SysAction(iname, a) →
                SPEC.actions(
                    a)(iname, v),
            T.mk_SysMsg(src, dst, m) →
                SPEC.messages(m)(src, dst, v)
        end,
```

perform_subchart_events performs all the possible subchart events of a RSC in a given state.

```
    perform_subchart_events :
        T.Chart → T.State → T.State
    perform_subchart_events(chart)(state) ≡
        let
            ees = T.get_enabled_events(chart, state),
            subchart_ees = find_subchart_events(ees)
        in
            if subchart_ees = {} then state
            else
                let
                    newstate =
                        T.step_event(chart, hd subchart_ees, state)
                in
                    perform_subchart_events(chart)(newstate)
                end
            end
        end,
```

find_subchart_events finds the subchart events in the given EnabledEvent-set.

```
    find_subchart_events :
        T.EnabledEvent-set → T.EnabledEvent-set
    find_subchart_events(ees) ≡
        if ees = {} then {}
        else
            let ee = hd ees in
                case ee of
                    T.EnabledEnterSubchart(_, _, _) → {ee},
                    T.EnabledEndSubchart(_, _) → {ee},
                    _ → find_subchart_events(ees \ {ee})
                end
            end
        end

value
```

is_action checks if a system event is an action.

```
    is_action : T.SysEvent → Bool
    is_action(event) ≡
        (∃ param : T.Inst_Name × Text •
            event = T.mk_SysAction(param)),
```

is_condition checks if a system event is a condtion.

```
is_condition : T.SysEvent → Bool
is_condition(event) ≡
    (∃ param : T.Inst_Name-set × Text •
        event = T.mk_SysCondition(param)),
```

is_message checks if a system event is a message.

```
is_message : T.SysEvent → Bool
is_message(event) ≡
    (∃
        param :
            T.Inst_Name-set × T.Inst_Name-set × Text
     •
        event = T.mk_SysMsg(param)),
```

this is from RSC_semantics, must be removed later.

```
step_event :
    T.Chart × T.State × T.EnabledEvent → T.State,
```

visible_event checks wether a given system event is a visible event in the given chart. If forbidden events where to be taken into account, whis function had to be changed.

```
visible_event :
    T.Chart →
        T.
        SysEvent → Bool
visible_event(chart)(event) ≡
    case event of
```

Checking if a action event corresponding to the system action event is present on the chart.

```
        T.mk_SysAction(iname, action) →
            (∃ e_id : T.ID •
                T.mk_ActionEvent(action, e_id) ∈
                    chart(iname)),
```

Checking if a message event corresponding to the system message event is present on the chart.

```
        T.mk_SysMsg(src, dst, method) →
```

The Environment is the source.

```
    if src = {}
    then
        (∃ e_id : T.ID •
            T.mk_InputEvent(e_id, T.Environment) ∈
                chart(hd src))
    else
```

The Environment is the destination.

```
        if dst = {}
        then
            (∃
                e_id : T.ID, e_t : T.HotCold
            •
                T.mk_OutputEvent(
                        e_id, method,
                        T.Environment) ∈ chart(hd src))
        else
```

The source and destination are instances on the chart.

```
            (∃ e_id : T.ID •
                T.mk_InputEvent(
                        e_id, T.mk_Address(hd src)) ∈
                    chart(hd src)) ∧
            (∃
                e_id : T.ID, e_t : T.HotCold
            •
                T.mk_OutputEvent(
                        e_id, method,
                        T.mk_Address(hd dst)) ∈
                    chart(hd src))
        end
    end,
```

Checking if a condition event corresponding to the system condtion event is present on the chart.

```
T.mk_SysCondition(shared_by, cond) →
    (∃
        iname : T.Inst_Name, e_id : T.ID,
        e_t : T.HotCold
    •
        T.mk_ConditionEvent(
                cond, e_id, e_t, shared_by) ∈
            chart(iname))
end,
```

overload of ∈: it checks if an Event one of the events specified on a location in the given Location*.

```
∈ : T.Event × T.Location* → Bool
```

```
        e ∈ ll ≡ e ∈ ⟨T.event(l) | l in ll⟩
    end
```

D.6.4 Semantics formal parameter

T

The formal parameter for a system

```
scheme specification =
    class
        value
            rscs : T.Collection,
            actions : Text →m T.Action,
            conditions : Text →m T.Condition,
            messages : Text →m T.Message,
            visible_variables : T.RSC →m T.VariableName-set
    end
```

D.6.5 Account example

The following example is a RSL model using two \rsc s to constrain the behaviour. The various parts are explained in detail in the following.

T

```
scheme account =
class
value
```

The two \rsc s are specified according to figure 6.19.

```
pre1_m1out : T.Event = T.mk_OutputEvent(01, ″InsertCard", T.mk_Address("ATM″)),
pre1_m1in : T.Event = T.mk_InputEvent(01, T.mk_Address(″Customer″)),
pre1_m2out : T.Event = T.mk_OutputEvent(02, ″SelectWithdrawal", T.mk_Address("ATM″)),
pre1_m2in : T.Event = T.mk_InputEvent(02, T.mk_Address(″Customer″)),
main1_m1out : T.Event = T.mk_OutputEvent(1, ″EnterAmount", T.mk_Address("ATM″)),
main1_m1in : T.Event = T.mk_InputEvent(1, T.mk_Address(″Customer″)),
main1_cond : T.Event = T.mk_ConditionEvent(″BalanceOK", 11, T.Cold, {"ATM","Customer″}),
main1_m2out : T.Event = T.mk_OutputEvent(2, ″Dispense", T.mk_Address("Customer″)),
main1_m2in : T.Event = T.mk_InputEvent(2, T.mk_Address(″ATM″)),
main_actA : T.Event = T.mk_ActionEvent(″NewBalance″, 21),
main1_m3out : T.Event = T.mk_OutputEvent(3, ″EjectCard", T.mk_Address("Customer″)),
main1_m3in : T.Event = T.mk_InputEvent(3, T.mk_Address(″ATM″)),
```

```
pre2_m1out : T.Event = T.mk_OutputEvent(01, "EnterAmount", T.mk_Address("ATM")),
pre2_m1in : T.Event = T.mk_InputEvent(01, T.mk_Address("Customer")),
pre2_cond : T.Event = T.mk_ConditionEvent("NotBalanceOK", 11, T.Hot, {"ATM","Customer"}),
main2_m1out : T.Event = T.mk_OutputEvent(1, "EjectCard", T.mk_Address("Customer")),
main2_m1in : T.Event = T.mk_InputEvent(1, T.mk_Address("ATM")),

pre1_insta : T.Location* =
    ⟨T.mk_Location(T.Hot, pre1_m1out),
      T.mk_Location(T.Hot, pre1_m2out),
      T.mk_Location(T.Hot, T.StopEvent)⟩,
pre1_instb : T.Location* =
    ⟨T.mk_Location(T.Hot, pre1_m1in),
      T.mk_Location(T.Hot, pre1_m2in),
      T.mk_Location(T.Hot, T.StopEvent)⟩,
main1_insta : T.Location* =
    ⟨T.mk_Location(T.Hot, main1_m1out),
      T.mk_Location(T.Hot, main1_cond),
      T.mk_Location(T.Hot, main1_m2in),
      T.mk_Location(T.Hot, main1_m3in),
      T.mk_Location(T.Hot, T.StopEvent)⟩,
main1_instb : T.Location* =
    ⟨T.mk_Location(T.Hot, main1_m1in),
      T.mk_Location(T.Hot, main1_cond),
      T.mk_Location(T.Hot, main1_m2out),
      T.mk_Location(T.Hot, main1_m3out),
      T.mk_Location(T.Hot, T.StopEvent)⟩,

mainch1 : T.Chart = [ "Customer" ↦ main1_insta,
                      "ATM" ↦ main1_instb ],
prech1  : T.Chart = [ "Customer" ↦ pre1_insta,
                      "ATM" ↦ pre1_instb ],
```

The first RSC which describes succesful withdrawal.

```
rsc1 : T.RSC = T.mk_RSC'("withdrawal", prech1, mainch1, {}) ,

pre2_insta : T.Location* =
    ⟨T.mk_Location(T.Hot, pre2_m1out),
      T.mk_Location(T.Hot, pre2_cond),
      T.mk_Location(T.Hot, T.StopEvent)⟩,
pre2_instb : T.Location* =
    ⟨T.mk_Location(T.Hot, pre2_m1in),
      T.mk_Location(T.Hot, pre2_cond),
      T.mk_Location(T.Hot, T.StopEvent)⟩,
main2_insta : T.Location* =
    ⟨T.mk_Location(T.Hot, main2_m1in),
      T.mk_Location(T.Hot, T.StopEvent)⟩,
main2_instb : T.Location* =
    ⟨T.mk_Location(T.Hot, main2_m1out),
      T.mk_Location(T.Hot, T.StopEvent)⟩,
mainch2 : T.Chart = [ "Customer" ↦ main2_insta,
                      "ATM" ↦ main2_instb ],
prech2  : T.Chart = [ "Customer" ↦ pre2_insta,
                      "ATM" ↦ pre2_instb ],
```

The second RSC which describes unsuccesful withdrawal.

rsc2 : T.RSC = T.mk_RSC′(″unsuccesful″, prech2, mainch2, {})

value

The constraints are given by the above two RSC's.

rscs : T.Collection = {rsc1, rsc2},

The actions, conditions and messages that are present on the RSC's are specified.

actions : **Text** $\overrightarrow{m}$ T.Action =
  [″NewBalance″ ↦ new_balance],
conditions : **Text** $\overrightarrow{m}$ T.Condition =
  [″BalanceOK″ ↦ balance_ok,
    ″NotBalanceOK″ ↦ not_balance_ok],
messages : **Text** $\overrightarrow{m}$ T.Message =
  [″InsertCard″ ↦ insert_card,
    ″SelectWithdrawal″ ↦ select_withdrawal,
    ″EnterAmount″ ↦ enter_amount,
    ″Dispense″ ↦ dispense,
    ″EjectCard″ ↦ eject_card],

The following variables have been specified to be used in the system. We only have three accounts in this local ATM.

variables : T.Variables =
  [″ATM.amount″ ↦ T.Integer(0),
    ″ATM.balance″↦ T.Integer(0),
    ″ATM.cashsupply″ ↦ T.Integer(10000),
    ″ATM.account1″ ↦ T.Integer(100),
    ″ATM.account2″ ↦ T.Integer(64),
    ″ATM.account3″ ↦ T.Integer(500)],

The variables that are visible for each chart are specified. Visible variables for a RSC may not be altered by other RSC's as long as its mainchart is active.

visible_variables : T.RSC $\overrightarrow{m}$ **Text-set** =
  [rsc1 ↦ {″ATM.amount", "ATM.balance", "ATM.cashsupply″,
  ″ATM.account1", "ATM.account2", "ATM.account3″},
    rsc2 ↦ {″ATM.amount", "ATM.balance″}]

In the following the signatures of the functions include the instance−names as specified in the types. They might be used in larger examples where the same message may happen on several instances. In this example this is not necessary and the given parameters are not always used.

balance_ok checks if the current balance stored in the ATM is higher than the requested withdrawal amount.

```
value
balance_ok :
    T.Inst_Name-set × T.Variables → Bool
balance_ok(ins, v) ≡
    if "ATM" isin ins /\ "Customer" ∈ ins
    then
        T.integer(v("ATM.balance")) >= T.integer(v("ATM.amount"))
    else
        false
    end,
```

The negation of balance_ok.

```
not_balance_ok :
    T.Inst_Name-set × T.Variables → Bool
not_balance_ok(ins, v) ≡ ∼balance_ok(ins,v),
```

If the cash is dispensed the variables are updated using the action new_balance.

```
new_balance :
    T.Inst_Name × T.Variables → T.Variables
new_balance(iname, v) ≡
  let
   newbalance = T.integer(v("ATM.balance")) - T.integer(v("ATM.amount")),
   newcash = T.integer(v("ATM.cashsupply")) - T.integer(v("ATM.amount"))
  in
   v † ["ATM.balance" +> T.Integer(newbalance), "ATM.cashsupply" ↦ T.Integer(newcash)]
  end ,
```

insert_card retrieves the balance of the account bound to the card that is inserted.

```
insert_card :
    T.Inst_Name-set × T.Inst_Name-set × T.Variables → T.Variables
insert_card(iname1, iname2, v) ≡
   v † ["ATM.balance" ↦ v(read_account_number())],
```

Underspecified function of the hardware reading the account number. Note that this function is not part of the RSC's.

```
read_account_number : Unit → Text,
```

Underspecified function for the ATM hardware that a withdrawal has been requested.

select_withdrawal :
T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables → T.Variables
select_withdrawal(iname1,iname2, v) ≡ v,

Function for abstracting the entering of the requested amount via the ATM′s keypad.

enter_amount :
T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables → T.Variables
enter_amount(iname1, iname2, v) ≡
v † [″`ATM.amount`″ ↦ T.Integer(read_key_pad())],

Underspecfied function for retriving the entered amount on the keypad. Again, this is not part of the RSC's.

read_key_pad : **Unit** → **Int**,

Dispense should tell the ATM hardware to dispense the requested amount. It should read the ″`ATM.amount`″ variable and let the hardware dispense the cash.

dispense :
T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables → T.Variables
dispense(iname1, iname2, v) ≡ v,

Underspecified function for abstracting the ATM hardware that ejects the card.

eject_card :
T.Inst_Name-**set** × T.Inst_Name-**set** × T.Variables → T.Variables
eject_card(iname1,iname2, v) ≡ v

end

The complete system of the account example is created using the semantics.

semantics, account

scheme system =
class
object ACC : account, system : semantics(ACC)
end

Appendix E

Contents of companion CD

The CD that accompanies this thesis has the following contents with the names of the directories on the CD emphasised:

root md5sums of the complete CD. Script starteclipse that starts Eclipse with all necessary dependencies.

eclipse The Eclipse Tool Platform 3.0.1 which includes GEF and Draw2D runtime 3.0.1 and ESDE. Ready to run.

esdeexamples Example .esde files that can be opened in ESDE. They were used as test cases.

esdejavadoc JavaDoc documentation for ESDE.

esdeplugin Our ESDE plugin ready for deployment in existing Eclipse installations. A version with and without source code is available.

esdeproject The source code of ESDE as a Eclipse project. It can be copied to an Eclipse workspace in order to browse, run and debug the source code using Eclipse.

libsd The compiled C++ library of the Scheme Diagram RSL specifications. Is used in ESDE for creating, checking and printing a Scheme Diagram in RSL. This version is compiled for Linux.

literature Most of the articles and reports we have used and referenced. Some were not available electronically.

java142 Java Runtime Environment version 1.4.2_07 which is necessary for running Eclipse.

rsl All the RSL specifications that are presented in the thesis.

rsl2latex Rsl2latex tool: used for converting *.rsl* files to *.tex* files for inclusion in LaTeX document.

rslbin RSL binaries of the specifications that are translatable. These include the test binaries.

rsltc The RSL type checker (version 2.5) used in ESDE for typechecking .rsl files.

thesis The thesis in dvi, ps and pdf. Included are the figures from the thesis.

thesistex The Latex files and figures the thesis is based on.

Appendix F

Use of ESDE CASE Tool

Contents

F.1 Installation 365
F.1.1 Live-CD 365
F.1.2 Installation of plugin 365
F.2 User manual 366
F.2.1 Overview 366
F.2.2 Description of elements 366
F.2.3 Palette 367
F.2.4 Action buttons 368
F.2.5 Canvas 368
F.2.6 Labels 368

F.1 Installation

F.1.1 Live-CD

Eclipse with GEF/Draw2D and the ESDE plugin can be run directly from the companion CD on a Linux-based PC with GTK windowing system and a *BASH* shell. Execute the script *starteclipse*. It must be started in the root directory of the CD. It starts Eclipse with all the necessary dependencies from the CD.

When Eclipse opens, choose where to place the workspace Eclipse uses. Thereafter choose workbench and Eclipse is ready.

F.1.2 Installation of plugin

The ESDE plugin is supplied ready for distribution to already existing installations. ESDE has been developed and testet with the mentioned versions. Later versions probably work as well. Requirements:

- Linux-based PC (testet on Debian with kernel 2.6.10)
- Eclipse 3.0.1 with GEF/Draw2D 3.0.1 plugins installed.
- Java 1.4.2 SDK.
- rsltc tool version 2.5 available via system path
- the libsd.so library

Copy the directory "CDROOT"/esdeplugin/source-build/rsl.esde.schemeeditor_0.6.4 or "CDROOT"/esdeplugin/binary-build/rsl.esde.schemeeditor_0.6.4, where CDROOT is the CD mount point, to the plugins folder in the Eclipse installation directory.

Eclipse must be started with the following command:

eclipse -vmargs -Djava.library.path="path/libsd/"

where path/libsd is the directory where the libsd.so file is stored.

F.2 User manual

For general Eclipse help see the built in help in Eclipse [8] or the online help [46].

Open the Eclipe workbench. Create a new project if you do not have one: *File - New - Project - Simple - Project - Next - Enter Name - Finish.*

Right click on the project and choose *New - Other*. Choose *Scheme Diagram - Next* and give the file a name with the extension *.esde*. Eclipse should automatically open the editor with the empty canvas.

Examples of *.esde* diagrams are included on the companion CD. They can be imported for viewing: *File - Import - File System* Choose the directory "CDROOT"/esdeexamples, select the wanted files and click *Finish*.

F.2.1 Overview

See figure F.1 for a screenshot of ESDE. The following ESDE features are present:

1. The *Actionbar* with buttons.
2. The *Canvas* for drawing the Scheme Diagram.
3. The *Outline* view that shows a thumbnail of the complete canvas.
4. The *Palette* with Select and Marquee tool, and selections for adding elements.

F.2.2 Description of elements

The following describes the elements and how to use them.

Schemes Schemes consist of a name and 5 compartments, one for each type of RSL declaration: Types, Values, Variables, Channels and Axioms.

Objects Objects consist of a name and the name of the scheme it is an instance of, separated by a ":". It has one compartment for parameter information. For each parameter in the scheme the object is an instance of, parameter information must be supplied.

Extend An extend relation must be drawn from scheme A to scheme B if scheme A extends scheme B.

Implement An implement relation can be drawn from scheme A to scheme B if scheme A statically implements scheme B.

Global A global association can be drawn from scheme/object A to object B if scheme/object A uses the global object B.

Nested A nested association must be drawn from scheme A to scheme B if scheme A has a nested object which is an instance of scheme B. The association has two labels. Near scheme B a label which gives the rolename of the object in scheme A. On the middle of the relation parameter information can be given if scheme B has formal parameters.

Parameter A parameter association must be drawn from scheme A to scheme B if scheme A has scheme B as a formal parameter. The association has two labels. Near scheme B a label which gives the rolename of the formal parameter in scheme A. On the middle of the relation parameter information can be given if scheme B has formal parameters.

Entries Entries can be added to the 5 RSL declaration compartments according to the syntax given in F.2.6.

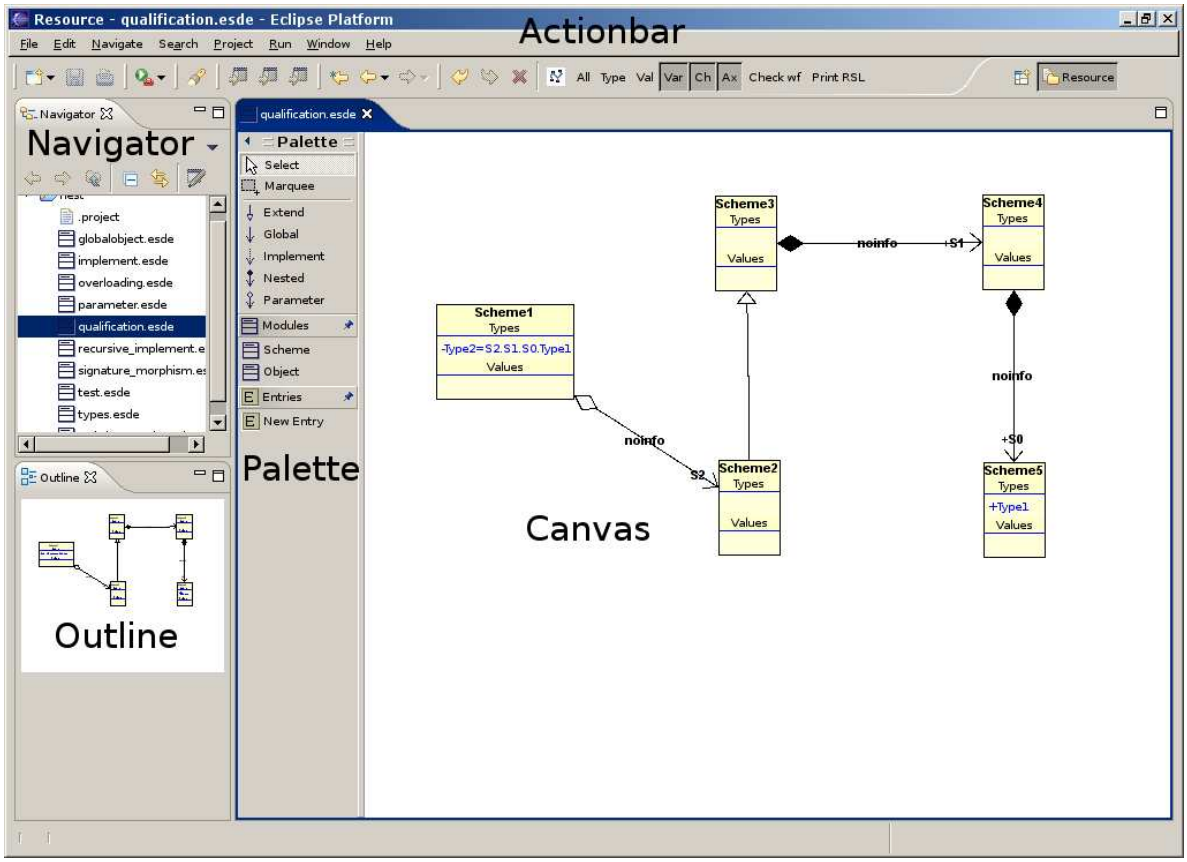

Figure F.1: A screenshot of ESDE with a Scheme Diagram.

F.2.3 Palette

The palette is used for manipulating the Scheme Diagram on the canvas.

Select Used for selecting elements that are to be deleted, renamed or moved.

Marquee Used for selecting several elements at once for deleting or moving.

Extend Used for adding an extend relation between two modules.

Global Used for adding an extend association between two modules.

Implement Used for adding an implement relation between two modules.

Nested Used for adding a nested association between two modules.

Parameter Used for adding a parameter association between two modules.

Scheme Used for adding a scheme to the canvas.

Object Used for adding an object to the canvas.

Entries Used for adding entries in the various compartments, e.g. Types.

All elements are given autogenerated names in order to show the syntax of the various names.

The editor does not enforce well formedness conditions. E.g. it allows to draw circular relations between modules. However there is a restriction: it is not allowed to draw a relation from A to B and from B to A at the same time, as this makes no sense.

F.2.4 Action buttons

The following ESDE specific action buttons are available:

Redo/undo . For redo/undo of commands. Are only available if a command has been undone/done.

Delete For deleting modules, relations and entries. Only available if an element is selected.

"Layout" For toggling automatic layout on/off. When on, all the elements are automatically moved in an ordered fashion. Does usually not create the most optimal layout but creates a basis for manual adjustments.

All For hiding all compartments in the schemes in order to get a easy to inspect diagram.

Type, Val, Var, Ch, Ax For hiding a specific compartment in schems: Type, Value, Variable, Channel and Axiom Compartments.

Check Saves the model as a RSL model and checks if it is well formed.

Print The same as above. If the model is well formed it is possible to select a directory. In the directory the text equivalent of the RSL specification is saved. Each module is saved in a file with the modules name and a *.rsl* extension. Thereafter the files are syntax checked using the RSLTC tool. If (not expected) syntax errors occur, they are displayed.

F.2.5 Canvas

It is possible to right click on the canvas and select the undo/redo/delete actions. If a scheme is selected another option is available: *Create Object*. It creates a new object with the selected scheme name automatically set as the scheme the object is an instance of.

F.2.6 Labels

In the following the syntax for user editable fields are given. The syntax is given using regular expressions.

General formats for inputs in labels

Name : **[a-zA-Z][\w]***
Must start with letter. Rest must be alphanummeric literals or underscore.

Simple Type expression (STE) : (Unit | Bool | Int | Nat | Real | Text | Char | [[Name.]*[Name]])[(-set | -infset | -list | -inflist)]?
Consists of a native type literal or a type possibly with a qualification. May be a set or list.

Type Expression (TE) : (STE [>< STE]* | STE (-> STE)+)
Is either a simple type expression, a product type expression or a function type expression.

Simple Map (SM) :(\[\]| \[Name +> Name[,Name +>Name]?\])
A map from names to names.

Actual Parameter Information (API) : (noinfo| Name +>\(Name,SM\)[,Name +>\(Name,SM\)]?
An actual parameter. noinfo denotes no actual parameter information. Otherwise it is a map from the formal parameter name to the actual parameter including a map for fitting information.

The following labels can be edited with input in the given syntax:

Scheme name : **Name**
The name of the scheme.

Object name : **Name:Name**
Must be a name followed by semicolon followed by name. The first name denotes the object name, the last denoted which scheme the object is an instance of.

Object parameter information : **Name +> Name, SM** Each parameter information gives the formal parameter (first Name) and the actual parameter (second Name) with possible fitting information.

Nested rolename : **[+-]Name**
Must start with a + or -. + Denotes visibility public, - denotes private. Must be follwed by a name that denotes the rolename of the supplier in the client.

Nested fitting : **API**
Must supplu actual parameter information.

Parameter rolename : **Name**
Name denotes the rolename of the supplier in the client.

Parameter fitting :**API**
Must supplu actual parameter information.

Types : **[+-]Name[=TE]?**
Must be a name preceeded by + or - for visibility as above. Possibly followed by an equation sign and a type expression.

Values, Variables, Channels : **[+-]Name[:TE]?** Same as above but with ":" instead of "=".

Axioms : **Name**
Must be a name that denotes the axiom.

Appendix G

Conferences

Contents

G.1 ICTAC 2004, Guiyang 371
G.1.1 Tutorials 371
G.1.2 Key note speakers 372
G.1.3 Talks 372
G.2 SEFM 2004, Beijing 373
G.2.1 Tutorials 373
G.2.2 Key note talks 373
G.2.3 Talks 373

During our stay at UNU-IIST we were invited to two conferences. Both conferences we attended where co-sponsored by UNU-IIST. For each conference the attended tutorials and talks will be presented.

G.1 ICTAC 2004, Guiyang

International Colloquium on Theoretical Aspects of Computing (ICTAC) 2004, Guiyang.
Duration: 20-24. September 2004
Homepage: http://www.iist.unu.edu/ICTAC2004/index.html
Organized by: Guizhou Academy of Sciences and UNU-IIST
The aim of the colloquium is to bring together practitioners and researchers from academia, industry and government to present research results, and exchange experience, ideas, and solutions for their problems in theoretical aspects of computing. It is planned to publish the proceedings in the Springer Lecture Notes Series.

G.1.1 Tutorials

- Formal Theories of Software Testing
  Hong Zhu, Oxford Brookes University, UK
- Formal Aspects of Software Architecture
  J L Fiadeiro, University of Leicester
- Formal Engineering Methods for Industrial Software Development - An Introduction to the SOFL Specification Language and Method
  Shaoying Liu
- Functional Predicate Calculus and Generic Functionals in Software Engineering
  Raymond Boute, University of Ghent, Belgium

G.1.2 Key note speakers

- Jifeng He, UNU-IIST, Macao
- Jose Luiz Fiadeiro, University of Leicester, UK
- Huimin Lin, Institute of Software, Chinese Academy of Sciences, China
- K. Rustan M. Leino, Microsoft Research, USA

G.1.3 Talks

- Reasoning about co-Büchi Tree Automata
  Salvatore La Torre and Aniello Murano (Università degli Studi di Salerno, Italy)
- Switched Probabilistic I/O Automata
  Ling Cheung, Frits Vaandrager (University of Nijmegen, The Netherlands), Nancy Lynch (MIT Computer Science and Artifical Intelligence Laboratory, USA) and Roberto Segala (Università di Verona, Italy)
- Foundations for the Run-time Monitoring of Reactive Systems - Fundamentals of the MaC language
  Mahesh Viswanathan (University of Illinois at Urbana Champaign, USA) and Moonzoo Kim (Pohang University of Science and Technology, Korea)
- Duration Calculus: A Real-time Semantic for B
  Samuel Colin, Georges Mariano (INRETS, France) and Vincent Poirriez (LAMIH, France)
- Atomic Components
  Steve Reeves and David Streader (University of Waikato, New Zealand)
- A Proof of Weak Termination Providing the Right Way to Terminate
  Olivier Fissore, Isabelle Gnaedig and Hélène Kirchner (LORIA-INRIA & LORIA-CNRS, France)
- A Logical Characterization of Efficiency Preorders
  Neelesh Korade (Persistent Systems Private Limited, India) and S. Arun-Kumar (Indian Institute of Technology, India)
- Specifying Software Connectors
  Marco Antonio Barbosa and Luís Soares Barbosa (Universidade do Minho, Portugal)
- A Formal Framework for Ontology Integration Based on a Default Extension to DDL
  Yinglong Ma, Jun Wei and Shaohua Liu (Institute of Software, Chinese Academy of Sciences, China)
- A Predicative Semantic Model for Integrating UML Models
  Jing Yang, Quan Long (UNU-IIST, Macao) and Xiaoshan Li (University of Macau, Macao)
- Automatic Mapping from Statecharts to Verilog
  Viet-Anh Vu Tran (Vietsoftware Company, Vietnam), Shengchao Qin and Wei Ngan Chin (National University of Singapore, Singapore)
- Reverse Observation Equivalence Between Labelled State Transition Systems
  Yanjun Wen, Ji Wang and Zhichang Qi (National Laboratory for Parallel and Distributed Processing, China)
- An Approach to Integration Testing Based on Data Flow Specifications
  Yuting Chen, Shaoying Liu, and Fumiko Nagoya (Hosei University, Japan)
- Combining Algebraic and Model-based Test Case Generation
  Li Dan and Bernhard K. Aichernig (UNU-IIST, Macao)
- Minimal Spanning Set for Coverage Testing of Interactive Systems
  Fevzi Belli and Christof J. Budnik (University of Paderborn, Germany)
- Reasoning about OWL & ORL Ontologies in PVS
  Jin Song Dong, Yu Zhang Feng and Yuan Fang Li (National University of Singapore, Singapore)

G.2 SEFM 2004, Beijing

Software Engineering and Formal Methods (SEFM) 2004, Beijing.
Duration: 26-30. September 2004
Homepage: http://www.iist.unu.edu/SEFM2004/
Organized by: Peking University and UNU-IIST
The objective of the conference is to bring together practitioners and researchers from academia, industry and government to exchange views on the theoretical foundation of formal methods, their application to software engineering and the socio-economic impact of their use. The proceedings have been published by the IEEE Computer Society Press.

G.2.1 Tutorials

- Software Architectures: Evolution and Mobility
  J. L. Fiadeiro, University of Leicester
- Model-Based Development: Mastering the Complexity of Reactive Systems
  Bernhard Schaetz, Fakultaet fuer Informatik, TU Muenchen

G.2.2 Key note talks

- Computation Orchestration: A Basis for Wide-Area Computing
  Jayadev Misra, University of Texas at Austin, USA
- Property-Driven Development
  Martin Wirsing, Ludwig Maximilian University, Munich, Germany
- Care, Feeding and Growth of Software Systems
  Mathai Joseph, Tata Research Development and Design Centre, India
- Random Testing in Isabelle/HOL
  Tobias Nipkow, Technische Universität München, Germany
- An Introduction to ABC Approach
  Hong Mei, Peking University, Beijing, China

G.2.3 Talks

- Symbolic Verification of Infinite Systems using a Finite Union of DFA's
  Suman Roy (Honeywell Technology Solutions Lab., India)
- Global vs. Local Model Checking: A Comparison of Verification Techniques for Infinite State Systems
  Tobias Schuele and Klaus Schneider (University of Kaiserslautern, Germany)
- Proof Reuse for Deductive Program Verification
  Bernhard Beckert and Vladimir Klebanov (University of Koblenz-Landau, Germany)
- Checking Extended CTL Properties using Guarded Quotient Structures
  A. Prasad Sistla, Xiaodong Wang and Min Zhou (University of Illinois at Chicago, USA)
- Modeling Peer-to-Peer Service Goals in UML
  Richard Torbjørn Sanders (SINTEF ICT/NTNU, Norway) and Rolv Bræk (NTNU, Norway)
- Past- and Future-Oriented Time-Bounded Temporal Properties with OCL
  Stephan Flake (Orga Systems GmbH) and Wolfgang Mueller(Paderborn University/C-LAB)
- On semantics and Refinement of UML Statecharts: A Coalgebraic View
  Sun Meng, Zhang Naixiao (LMAM, Peking Universit, China) and Luis S. Barbosa (Minho University, Portugal)

- The Rhapsody UML Verification Environment

  Ingo Schinz, Christian Mrugalla (OFFIS, Germany), Tobe Toben and Bernd Westphal (Carl von Ossietzky Universität, Germany)

- An Asynchronous Communication Model for Distributed Concurrent Objects

  Einar Broch Johnsen and Olaf Owe (University of Oslo, Norway)

- Modeling and Temporal Logics for Timed Component Connectors

  Farhad Arbab, Frank de Boer, Jan Rutten (CWI, The Netherlands) and Christel Baier (Universität Bonn, Germany)

- Glass-Box and Black-Box Views on Object-Oriented Specifications

  Michel Bidoit (CNRS & ENS de Cachan, France), Rolf Hennicker, Alexander Knapp and Hubert Baumeister (Ludwig-Maximilians-Universität, München, Germany)

- Exception Safety for C#

  K. Rustan M. Leino and Wolfram Schulte (Microsoft Research, WA, USA)

- Heuristics for Refinement Relations

  Florian Kammueller (Technische Universität Berlin, Germany) and J.W.Sanders (Programming Research Group, OUCL, United Kingdom)

- Towards Action Refinement for Concurrent Systems with Causal Ambiguity

  Jonzhao Wu (Universität Mannheim, Germany) and Hougang Yue (Chinese Academy of Sciences, China)

- Refine and Gabriel: Support for Refinement and Tactics

  Marcel Oliviera (University of Kent, England), Manuela Xavier (Universidade Federal de Pernambuco, Brazil) and Ana Cavalcanti (University of Kent, England)

- Automated Element-Wise Resoning with Sets

  Georg Struth (Universität Augsburg, Germany)

- A Formalism for Conformance Analysis and Its Applications

  Tien N. Nguyen and Ethan V. Munson (University of Wisconsin-Milwaukee, USA)

- The Formal, Tool Supported Development of Real Time Systems

  Dr. Richard O. Sinnott (University of Glasgow, Scotland)

- Using relation algebra for the analysis of Petri nets in a CASE tool based approach

  Alexander Fronk (University of Dortmund, Germany)

- Formal Verification of Requirements using SPIN: A Case Study on Web Services

  Raman Kahamiakin, Marco Pistore (University of Trento, Italy) and Marco Roveri (ITC-irst, Italy)

- How to Verify Dynamic Properties of Information Systems

  Neil Evans, Helen Treharne (Royal Holloway, England), Regine Laleau (IUT de Fontainebleau, France) and Marc Frappier (Universite de Sherbrooke, Canada)

Appendix H

Seminars

Contents

H.1 Seminars at UNU-IIST 375
H.1.1 "In-house talk" 375
H.1.2 Formal software specification and development using RAISE 375
H.1.3 Travelling Processes 376
H.1.4 Test Purpose Generation by Specification Mutation in Distributed Systems 376
H.1.5 A formal model for JavaBeans 376
H.1.6 Testing and Diagnosis of Software Design Specifications 377
H.1.7 UML: Promises, Problems and Solutions 377

H.1 Seminars at UNU-IIST

This section will list chronologically the seminars which we have attended during our stay at UNU-IIST. For each seminar the speaker, abstract, relevance to our project, date, and duration will be given.

H.1.1 "In-house talk"

He Jifeng, UNU-IIST research fellow

Friday, 6. August, 2004, Duration: 2 hour
Abstract:
The talk is about how to integrate various tools based on different semantic frameworks. Specifically the integration with bisimulation was discussed.

Relevance to our project:
How to integrate tools in another way than in this thesis.

H.1.2 Formal software specification and development using RAISE

Chris George, UNU-IIST Director a.i.

Everyday during 6-10 September 2004., Duration: 3 hours each day
Abstract:
RAISE - Rigorous Approach to Industrial Software Engineering - was first developed in European collaborative

projects during 1985-94. The RAISE Specification Language (RSL) is wide spectrum, supporting descriptions ranging in style between abstract and concrete, applicative and imperative, sequential and concurrent. RSL is also modular, supporting the writing of large descriptions.
RAISE includes a method for software development, and also a set of free, open-source, portable tools. Tool supported activities include generation of specifications from UML class diagrams, validation and verification of specifications, refinement, prototyping, execution of test cases, mutation testing, test coverage analysis, generation of documents, and generation of program code in C++ by translation.
The course will introduce the language and method, and also include practical work with the tools.

Relevance to our project:
Modules where covered which was very relevant to our specification of the Scheme Diagram. Furthermore channels and concurrency in RSL was covered which was relevant with regards to our attempt to use the CSP part of RSL for modelling LSC's

H.1.3 Travelling Processes

Xinbei Tang, University of Kent, UK

Monday, 13. September, 2004, Duration: 2 hours
Abstract:
This talk describes a refinement-based development method for mobile processes. Process mobility is interpreted as the assignment or communication of higher-order variables, whose values are process constants or parameterised processes, in which target variables update their values and source variables lose their values. The mathematical basis for the work is Hoare and He's Unifying Theories of Programming (UTP). In this talk, we present a set of algebraic laws to be used for the development of mobile systems. The correctness of these laws is ensured by the UTP semantics of mobile processes. We illustrate our theory through a simple example that can be implemented in both a centralised and a distributed way, and we show how the centralised system may be step-wisely developed into the distributed one using our proposed laws.

Relevance to our project:
N/A

H.1.4 Test Purpose Generation by Specification Mutation in Distributed Systems

Carlo Corrales Delgado, UNU-IIST fellow

Friday, 17. September, 2004, Duration: 2 hour
Abstract:
Mutation Testing is a fault-injection technique which can assess how thoroughly a set of test cases exercises an implementation. We modify the approach in two ways: First, it is applied on the specification level, second, test cases are generated rather than assessed. More precisely, we generate a test purpose which is an abstract description of a test goal. Then, using the TGV test case generator, the actual test cases can be generated from a test purpose. In this talk, we describe our fault-based testing approach as well as the use of the CADP tools to automate it. In addition, we briefly present a short case study applying the method to test a HTTP web server.

Relevance to our project:
N/A

H.1.5 A formal model for JavaBeans

Bhim P. Upadhyaya, UNU-IIST fellow

Friday, 17. September 2004, Duration: 2 hour
Abstract:
Component based software development focuses on building software systems by assembling existing software components. This makes the systems more maintainable, reduces devlopment time and minimizes development as well as maintenance costs. The Java programming language supports component based software development through JavaBeans. Specifying JavaBeans in a natural language is ambigious to the software systems developers. The use of a formal technique helps to express JavaBeans and consequently JavaBeans-based software systems precisely. In this seminar, we present a formal model of JavaBeans, whereby a system can be divided into a number of interconnected JavaBeans. We adopt the notion of refinement to formalize the replaceability of JavaBeans.

Relevance to our project:
Use of formal techniques, especially formalising existing informal techniques.

H.1.6 Testing and Diagnosis of Software Design Specifications

W. Eric Wong
Department of Computer Science
University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/ẽwong

Monday, 4. October, 2004, Duration: 2 hour
Abstract:
Statistical data show that early fault detection in the software development process can cut cost significantly. Additionally, with improved technology for automatic code generation from architectural design specifications, it becomes even more important to have a highly reliable system design from the beginning. To ensure this, the quality of the system must be predicted and improved as early as possible. This talk will focus on how to extend source code-based techniques to the software design specification level to allow more efficient specification validation and maintenance. Also discussed will be a novel approach for constructing a small set of effective test sequences from a design model to satisfy certain structural coverage criteria. These tests can be used not only for conformance testing with a focus on the consistency between the specification and the implementation but also for the validation of the design specifications themselves.

Relevance to our project:
A few elements of the presented work are related to ours. They have introduced a diagram for SDL with equivalent semantics as the textual representation. Another interesting relation is the use of MSC in their case tool. It is however only used to display the progress of a test run for a user.

H.1.7 UML: Promises, Problems and Solutions

Zhiming Liu, UNU-IIST research fellow

Friday, 8. October, 2004, Duration: 2 hours
Abstract:
This talk discusses the promises and the problems in the use of UML in software development. We will also look at the research directions in formal support of UML. We will then discuss the the idea of using a formal component-based and object-oriented specification language (CBOOL) to formalize and combine UML models. With the formalization, we develop a set of refinement laws of UML models to capture the essential nature, principles and patterns of object-oriented design. With the the incremental and iterative features of object-orientation and the Rational Unified Process (RUP), our method supports precise and consistent UML model transformation.

Relevance to our project:
The talk was somwehat related to our work of formailising LSC's.

Appendix I

A tale of two cities

Contents

I.1 Macau 379
I.2 Singapore 380

As most of our thesis was carried out in Macau and Singapore we include a short commentary about the two very different cities. "A tale of two cities" is a book by Charles Dickens. It is situated in 18th century in the years up to the french revolution in 1789. It has descriptions of London and Paris, hence the name, that fits well for this section.

I.1 Macau

Macau is located only 50 km south west of Hong Kong. Almost everybody has heard of Hong Kong, not so with Macau. Macau is a former Portuguese colony, which initially was a trading post. It was transferred back to China in 1999, two years after Hong Kong. Still today there are many remnants of the Portuguese colonists. There are picturesque old colonial style houses which have been beautifully restored. Today, much of the economy is based on gambling. It is a Las Vegas in China, with of course many Chinese tourists. They visit the large casinos, like the internationally known *Sands*.

Macau consists of the Macau peninsula (see figure I.2) and two islands, Taipa and Coloane. The peninsula is one of the most densely populated cities in the world, with about 500.000 inhabitants living on a few square kilometres. Nonetheless there are plenty of parks and green areas that invite for a leisurely stroll in the warm climate. The southern part also has a modern district with distinctly modern high risers. To the north it is more compact housing, displaying the hustle and bustle of the mainly Chinese population.

Taipa is becoming increasingly built up, with many new housing complexes which are rather uninteresting. Coloane is surprisingly untouched and gives the opportunity for walks and beach days.

Macau has also a yearly Grand Prix, with the Formula 3 race as the highlight. Public roads are used as track, just as in Macau. An exiting event where we had front row seats, as the track was just in front of the UNU-IIST property. What an excitement!

All in all Macau was a great experience and very different from the more high-paced culture of downtown Hong Kong. If you are in Hong Kong and have a day to spare, stop by and enjoy some of the offerings of Macau.

Figure I.1: Map of the Macau peninsula. We studied at UNU-IIST, which is located just south of the location marked with 3, which is the old Guia light tower. It provides a scenic view over most of Macau.

I.2 Singapore

Singapore is a former British colony and became self-governing in 1959. Singapore was a member in the Malayan Union (since 1963 Malaysia) until it was kicked out and it became the Republic of Singapore in 1965. Since, its economic growth has made it a financial and commercial hub in Asia, well aided by its big oil refining industry.

About 70% of the population are of Chinese heritage, and various Chinese dialects are widely spoken. English is however the official language and is spoken by most. Quite a contrast to Macau, where very few speak English. The rest of the population is mainly Malayan and Indian. Up till today there are still tensions between Malaysia and Singapore, despite the fact that their bonds are tight. Malaysia is a heavy supplier of labour to the economic growing Singapore.

An interesting fact for us "democratic" westerners is, that Singapore is in effect a one-party state, governed by the Peoples Action Partly (PAP) since its independence. It is lead by the "grand old man", Lee Kuan Yew as Prime Minister since the independence of Singapore 1959. But it definitely seems like he does his work well.

Singapore is surprisingly western in its appearance, especially in the old Colonial and Central Business District. But Singapore offers a vast array of multi ethnic and cultural neighbourhoods. Especially in Little India it was fun to stroll around and have a look at all the shops. In general Singapore is a shoppers paradise. Spearheaded by the famous Orchard Road, where you will find an abundance of department stores, malls and shops.

Everywhere the city is surprisingly clean and well kept. This is probably helped by the harsh punishments for people who litter or otherwise behave "badly". Singapore feels very safe and is a wonderful city. It is definitely worth a visit.

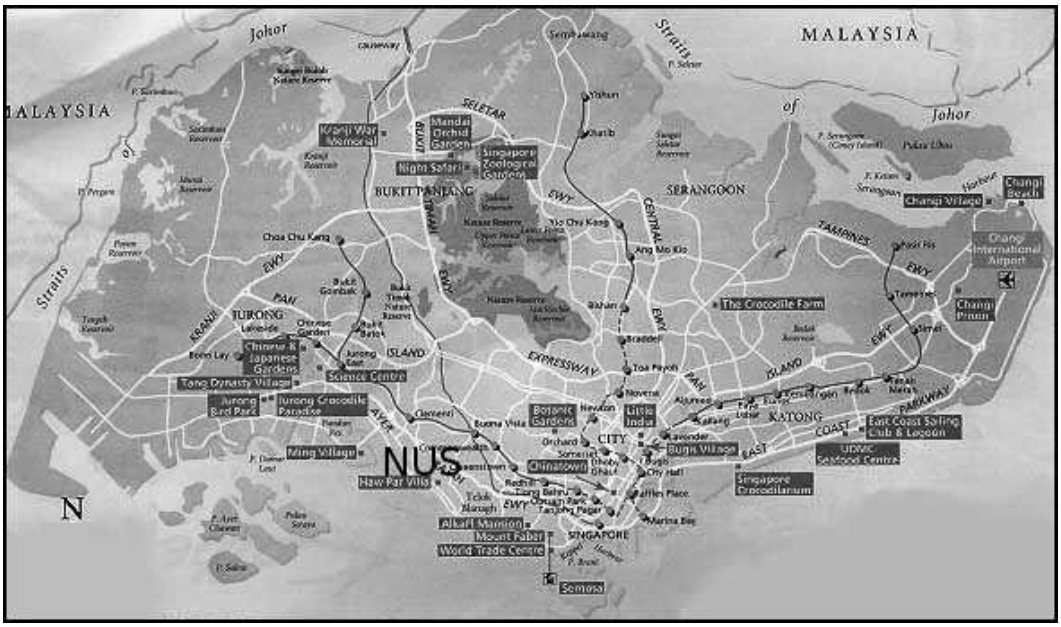

Figure I.2: Map of Singapore. We studied at National University of Singapore. It is located at the "U" in NUS. Central Singapore is to the east of it.