

An Eclipse based Development Environment for RAISE

Marieta Vasilica Fasie

DTU



Kongens Lyngby 2013
IMM-M.Sc.-2013-16

Technical University of Denmark

DTU COMPUTE

Department of Applied Mathematics and Computer Science

Matematiktorvet, DK-2800 Kongens Lyngby, Denmark

Phone +45 45253351, Fax +45 45882673

reception@imm.dtu.dk

www.imm.dtu.dk IMM-M.Sc.-2013-16

Summary (English)

In order to support the RAISE (Rigorous Approach To Industrial Software Engineering) formal method and the RAISE specification language, a series of tools have been developed in the past. Although the tools have been successfully used both in industrial and academic environments, they lack many of the features a development environment offers nowadays.

The goal of this master thesis is twofold. Firstly, to create an integrated development environment for RAISE that is easy to extend and integrate with new tools, and secondly, to test EBON's (Extended Business Object Notation) applicability on plug-ins development, using the tool under development as a case study.

The thesis describes a methodology for designing and analyzing plug-ins, that is based on BON methodology and that is used to analyze and design eRAISE. Then eRAISE is implemented based on the Eclipse framework providing an integrated development environment for RAISE's tool suite. By using the plug-in mechanism available in Eclipse, eRAISE can easily be extended with new features and tools. Furthermore, the report proposes some ideas on how can the new plug-in be extended and improved.

Summary (Danish)

For at understøtte RAISE (Rigorous Approach To Industrial Software Engineering) formal method og RAISE specifikationsproget, er en række værktøjer blevet udviklet. Selvom værktøjerne er blevet succesfuldt anvendt både i industrielle og akademiske miljøer, mangler mange af de funktioner et udviklingsmiljø tilbyder i dag.

Der er to mål med denne kandidatafhandling. For det første at skabe et integreret udviklingsmiljø for RAISE, der er let at udvide og integrere med nye værktøjer. For det andet er at teste EBONs (Extended Business Object Notation) anvendelighed på udviklingen af plug-ins, ved at bruge værktøjet under udvikling som et case study.

Afhandlingen beskriver en metode til at designe og analysere plug-ins som er baseret på BON metoden, og som bruges til at analysere og designe eRAISE. Baseret på Eclipse framework giver eRAISE et integreret udviklingsmiljø til RAISE's tool suite. Ved hjælp af plug-in mekanismen i Eclipse, kan eRAISE let udvides med nye funktioner og værktøjer. Derudover foreslås der ideer i rapporten om hvordan den nye plug-in udvides og forbedres.

Preface

This thesis was prepared at DTU Compute at the Technical University of Denmark in fulfillment of the requirements for acquiring a M.Sc. degree in Computer Science and Engineering.

The project has been supervised by Associate Professor Anne Elisabeth Haxthausen and Professor Joseph Kiniry.

Lyngby, 15-May-2013

A handwritten signature in black ink, appearing to be 'Marieta Vasilica Fasie', written in a cursive style.

Marieta Vasilica Fasie

Acknowledgments

Special thanks goes to both my supervisors for their great support and constructive feedback throughout the entire thesis.

I would like to thank Associate Professor Anne Elisabeth Haxthausen, for her constant feedback, for paying special attention to details, for always inspecting the quality of my work and for always making time for me.

I would like to thank Professor Joseph Kiniry for always bringing great and new ideas, for seizing opportunities, for making things fun, interesting and exciting and for having great answers to all my questions.

Contents

Summary (English)	i
Summary (Danish)	iii
Preface	v
Acknowledgments	vii
1 Introduction	1
1.1 Motivation	1
1.2 Goals	3
1.3 Paper structure	4
1.4 How to read the paper	4
2 Background	7
2.1 RAISE	7
2.1.1 RAISE concepts	8
2.1.2 Eden tool set	8
2.1.3 Rsltc tool set	9
2.2 Eclipse	10
2.2.1 Eclipse concepts	11
3 Development tools	13
3.1 BON	13
3.1.1 Static models	14
3.1.2 Dynamic models	15
3.2 BONc	15
3.3 Beetlz	16
3.4 Eclipse PDE	17

3.4.1	New Plug-in Project wizard	18
3.4.2	Plug-in Manifest Editor	18
3.4.3	Plug-in debug	19
3.4.4	Plug-in tests	19
4	Analysis and design	21
4.1	Analysis and design method	21
4.2	Domain modeling	23
4.3	User Interface	26
4.4	Events	29
4.5	Components	32
4.6	Components Communication	32
4.7	Code Generation	34
5	Implementation	37
5.1	Plan	37
5.2	rsl.core plug-in	39
5.2.1	Type check	39
5.2.2	SML translate	41
5.2.3	Test cases execution	42
5.2.4	L ^A T _E X generation	43
5.3	rsl.editor plug-in	44
5.3.1	Editor	44
5.3.2	Markers	49
5.4	rsl.testcases plug-in	51
5.5	rsl.wizard plug-in	56
5.5.1	RSL perspective	56
5.5.2	New RSL Project wizard	57
5.6	Extension points	60
6	Testing	63
6.1	Manual testing	63
6.1.1	Input validation	64
6.2	Automated testing	66
6.2.1	JUnit testing	66
7	User guide	69
7.1	Writing RSL specification	69
7.1.1	Create a new RSL project	70
7.1.2	Create a new RSL file	71
7.1.3	Edit the RSL file	71
7.2	Type check a RSL specification	72
7.3	Translate RSL specification to SML	74
7.4	Run test cases	74

7.5	Generate Latex document	76
7.6	Actions on more than one file	76
7.6.1	RSL menu	76
7.6.2	Context menus on multiple files	77
8	Future work	79
8.1	Future work	79
8.1.1	Example	80
9	Conclusions	85
9.1	What was achieved	85
9.2	Conclusions	86
A	Article	87
B	eRAISE domain model	91
C	UI mock-ups	101
D	Scenarios	113
E	Events	117
F	Static diagram	119
G	Components' interfaces	121
H	Generated Java code	125
I	Prioritized scenarios	129
J	RSL key words	133
K	Colours used inside RSL editor	135
L	SML run-time errors	137
	Bibliography	139

Introduction

This chapter is meant to introduce the reader in the project topic. First, a discussion is made on the reasons that are behind the current project. Secondly, the project goals are stated and explained. The last two sections explain the reader how the paper is structured and offers guidelines for reading it.

1.1 Motivation

RAISE consists of a formal method [Gro95], a specification language [The92] and a set of tools built around the language. So far, there have been two generations of tools. The first generation of tools provided all the support required for applying RAISE to industrial software development, but the tools could only be used on SUN workstations. The second generation of tools, *rsltc* [rsl08], solved the portability problem and brought a big number of supporting tools. However, this new set lacks the editing support part. Any editor can be used for writing RAISE specifications, but only *emacs* provides some editing support and only in the form of syntax highlighting.

Furthermore, no work has been done in the last years for improving the set of *rsltc* tools and therefore this set is missing the features that many development environments have. Very little of the help support that we use nowadays when

developing Java or C projects is provided within `rsltc`. For example, there is no support for working with big projects, no automatic editing support, no team support and so on.

Tools' extensibility is another important aspect that must be considered when talking about the tools lack of development. `rsltc` is written in C using the GENTLE Compiler Construction System [Sch05] and the emacs interface is written in Lisp. Although these technologies are widely used, and the `rsltc` source code is easy to get, a method that does not involve modifying and building the source code, would probably attract more contributors.

A very used way for extending a software capabilities is through the plug-in mechanism. Plug-ins are very convenient to use since they provide an easy way to add features, they hide the implementation details from the extensions, they allow parallel development of features and much more. Thus plug-ins encourage contributions without having to worry about the contributor level of expertise. Of course plug-ins have also many disadvantages, since they increase rapidly the complexity of a product and since testing them automatically is quite a challenge.

A good example of plug-ins integration and strong plug-in architecture is Eclipse. In Eclipse everything is a plug-in build on top of a plug-in framework. But besides this, Eclipse is also an excellent integrated development environment (IDE) offering developers great features and support for developing their software products. The fact that Eclipse is free (released under the terms of Eclipse Public License), provides all the features an IDE should have and is easily extendable, makes it popular inside academic environments. Many academics choose to package their tools inside Eclipse in order to make it more user friendly and to gain many of the features provided by Eclipse IDE.

While finding plug-in examples and guidelines on how to implement and use their interfaces is quite easy, not the same thing can be said about finding quality plug-ins. There are many bad designed plug-ins with poor documentation, that maybe were not even passed through a design phase, but directly implemented in order to get fast results and functionality. This situation is mostly attributed to the lack of methodologies in what concerns the plug-in development. There is not much published work in this area and the ones that are, do not provide a methodology e.g., [NLS⁺12] provides some principles, but not a method.

There are many question raised in the previous paragraphs, questions like:

- What is the future of the RAISE tools?
- How should the RAISE tool set be extended?

- Is the Eclipse wrapping a solution?
- If yes, then how should the plug-in be designed so that it will be easily extendable?

The current project is trying to answer these questions by creating an Eclipse based development environment for RAISE named *eRAISE*. Furthermore, the current paper is intended to pay special attention to eRAISE's analysis and design phase by using a methodology based on BON [WN94]. This methodology has made the subject of an article [FK13] published by the author of this theses and her supervisors.

1.2 Goals

As it was mentioned in the previous section, there are two main goals that this project intends to reach:

1. To create an Eclipse based development environment for RAISE that can be easily extended with new features and tools.
2. To test EBON's applicability on plug-ins development, using eRAISE as a case study.

The first goal is quite wide and ambiguous and therefore it requires more explanation. The current project is not intended to integrate all existing RAISE tools inside Eclipse and to recreate the exact state of the tools inside Eclipse, but to create a core, a starting point from where different projects can start and bring further contributions. No exact limit has been prior established for how many RAISE tools are to be integrated and/or created, since the focus is on providing a quality, well designed system. The set of requirements for eRAISE are identified in chapter 4 and can be found prioritized in Appendix I.

The importance for creating a reliable plug-in with good documentation brings the focus on the second goal. eRAISE's analysis and design phase will be based on a methodology grounded in BON.

1.3 Paper structure

Chapter 2 provides the background of RAISE and Eclipse in a dedicated section for each. A bit of history, purpose and generalities are presented in order to introduce the concepts. Chapter 3 describes the tools that have been used throughout the entire product development, from analysis and design to implementation and testing. In chapter 4 the analysis and design phase of eRAISE is described in a detailed manner. Chapter 5 presents the implementation details and the decisions that have been made when implementing eRAISE. The testing phase of the product is captured in chapter 6, where the testing methods, the applied tests and the results are shown. Chapter 7 presents a short user guide meant to help the user familiarize with eRAISE. Chapter 8 provides ideas and examples on how eRAISE can be extended. The last chapter, chapter 9, draws the conclusions of this paper.

Additional information, meant to support decisions, to offer examples or show work fragments is presented in appendixes. Appendix A contains the article published by the author of this thesis and her supervisors. Appendix B presents the system domain model expressed in EBON notation. Appendix C contains the UI sketches for the system features. The complete list of EBON scenarios and events are captured in appendix D and appendix E. The system components and the interfaces are captured in appendix F and appendix G. The Java skeleton code generated by Beetlz can be found in appendix H. Appendix I contains the set of scenarios in the order they were used in the implementation phase. Appendix J presets the keywords existing in the RAISE language, while appendix K presents the colours that have been used for syntax highlighting inside the editor. Appendix L captures the set of possible run-time errors generated by smlnj when executing RSL test cases.

1.4 How to read the paper

The current paper can be read by any person that has some background in programming. But different readers may be interested in different parts of the paper and this subsection tries to identify the groups of readers and the chapters interesting for them.

For the reader who has read *A Rigorous Methodology for Analyzing and Designing Plug-Ins* [FK13] article and is interested in finding out more about the described methodology and its applicability, chapter 4 covers this part. However it may be interesting to read also chapter 2 and chapter 3 in order to get

familiarized with the method and tools that support the methodology.

The eRAISE user can find in chapter 7 an user guide for eRAISE functionality.

For the reader who wants to extend eRAISE with new functionality, reading chapter 7 and chapter 8 is probably enough to get an overview of how eRAISE is working and what interfaces it exposes.

The reader who is interested in updating different parts of eRAISE, must read chapter 4, chapter 5 and chapter 6, since they contain the details and decisions behind analysis and design, implementation and testing.

For the rest of the readers it is recommended to go throughout the entire thesis. In the beginning of each chapter, the reader can find a summary describing the chapter content that can be used to decide whether the current chapter is bringing new information for the reader.

Background

This chapter presents generalities about RAISE and Eclipse. It is intended to be read by people who know few or nothing about these subjects. The chapter has two sections, one for RAISE and one for Eclipse. The RAISE section presents its history, concepts and its tool set. The Eclipse section presents generalities and the main concepts used when working and using this framework. If the reader is comfortable with EBON and Eclipse, she can skip this chapter completely.

2.1 RAISE

RAISE stands for Rigorous Approach to Industrial Software Engineering and consists of a formal method, a specification language (RSL) and a set of tools built around the language.

RAISE began to take shape during the RAISE ESPRIT program in 1985-1990. The aim of this project, whose starting point was VDM, was to bring an improvement over other formal methods like Z, ML, CSP, CCS, Clear, Larch and OBJ [for97], [Hax99]. The outcome of this program was the RAISE formal method, the RSL language and some supporting tools [Geo03].

In the next years, 1990-1995, during the LaCoS project, RAISE has been successfully applied on a series of industrial projects leading to the development of new tools. This first generation of tools was called *eden* and developed by CRI A/S using the Cornell synthesizer generator [Rep]. The list of tools comprised by the first generation is presented in subsection 2.1.2. From that point, RAISE has become the main formal method used in UNU/IIST [UNU]. However, these tools had the disadvantage that they were available only on SUN workstations. The UNU/IIST mission was to help developing countries and since few institutions in these countries were using SUN workstations [Geo03], a new generation of RAISE tools was created.

The second generation of RAISE tools, called *rsrtc*, was developed starting with 1998 at UNU/IIST. *rsrtc* was developed using the GENTLE Compiler Construction System [Sch05], a C based compiler, while its emacs interface is written in Lisp. The set of tools comprises the elements presented in subsection 2.1.3.

2.1.1 RAISE concepts

An RSL specification consists of a module definition. A module can contain declarations of types, values, variables, channels, modules and axioms [Gro95]. There are two kinds of modules: objects and schemes.

2.1.2 Eden tool set

The first generation of RAISE tools comprised the following elements:

1. Syntax directed editors
For RSL modules, theories, development relations and justifications.
2. Repository with version management
3. Pretty printer
4. Ada translator
5. C++ translator

2.1.3 Rsltc tool set

The second generation of RAISE tools can be used on any platform that supports C. It provides a command line interface, but also offers the possibility to be used from inside emacs. The input of the rsltc tools is an rsl text file, which can be modified using any editor. However emacs is recommended since it provides syntax highlighting and a menu to access the various tools. Based on [rsl08] and [Geo03], the rsltc tool set comprises the following tool components:

1. Syntax and type checking
Verifies the syntax and the types in the rsl file given as input. The output of the tool is the module name along with the list of errors if there are any. Every error message is preceded by the line and column number of the code that generated the error.
2. Module dependencies tree
Shows the module dependencies in a tree format using ASCII. A module's dependencies are presented on its right side on the next lines. The direct dependencies have one more indentation than the number of indentations the referring module has. Therefore the left most module is the one with the most dependencies.
3. Module dependencies graph using VCG
Outputs a module dependency graph in a .vcg file using the Visualization of Computer Graphs tool. The schemes are represented as red rectangles, objects as blue ones, theories as yellow diamonds and development relations as cyan triangles.
4. Pretty printing
Pretty prints the current module into the standard output.
5. Confidence condition generation
Used to check the consistency of the RSL source. The output of this is a list of confidence conditions for the specification in the rsl file given as input.
6. C++ translator
Translates the RSL specification into C++ executable code. Only a subset of RSL can be translated into C++ and the results of this translation are an .h file and an .cpp file
7. SML translator
Translates the RSL specification into executable SML code. For a module named *test*, the tool creates two new files named *test.sml* and *test_.sml*.

8. PVS translator
Translates the RSL specification into PVS. The output of the translation is a .pvs file. The PVS translator is used for theorem proving.
9. SAL translator
Translates RSL specification into SAL specification. The output of this translation contains multiple .sal files. Three of them correspond to the three versions used for translating a file, while the others can contain the types. Used for model checking the RSL-SAL specifications.
10. Support for \LaTeX documents
Generates a .tex file that can be integrated in a \LaTeX document.
11. UML to RSL translator
The purpose of this tool is to formalize the UML class diagrams using RSL. It works by translating an xml file, which represents a UML class diagram, into RSL files. The translated xml file must be the output of the modeling tool in which the class diagram was created.
12. Test case execution
It is based on the RSL test case feature. The tool uses the SML translator and SML run time system to execute the tests.
13. Mutation testing
Offers test coverage support.

2.2 Eclipse

Eclipse is a software development environment whose purpose is to provide a "universal toolset for development" [Ecl13]. It started as a project in late 90's at IBM and it was build around the Java technologies [Cer05].

Eclipse provides development support for many different languages making it a great tool for those who code in more than one language. However, Eclipse is mostly known for its Java development tools (JDT).

There are many reasons for Eclipse popularity. One is the fact that it provides all the features that eases and help the work of software developers. Among this, the most important features are:

- Code writing support
For example the Java editor provides syntax highlighting, type aware completion that helps programmers with suggestions while typing in the text

editor, folding which offers the possibility of hiding and showing fragments of the currently displayed text, automatic generation of getters and setters for different fields and many others.

- **Code debugging support**
Allows a developer to follow the code execution while using facilities like setting breakpoint, stepping through the code, suspending threads and analyzing variables at run time.
- **Team support**
Eclipse simplifies the work with source code repositories helping teams synchronize their work e.g., Subversive [Sub] plug-in which provides support for working with SVN repositories, EGit plug-in [EGt] for the GIT version control system, Mylyn [Myl] which facilitates working with very large projects by applying a task-focused technique.
- **Documentation support**
For example Eclipse provides automatic generation of Javadoc comment templates inside the Java code. From these comments, Java documentation is generated in an HTML format.

Another important aspect that makes Eclipse popular is its extensibility. The Eclipse architecture and its plug-in development mechanism that allows third parties to contribute to Eclipse are presented in detail in section 3.4.

2.2.1 Eclipse concepts

When using Eclipse it is important to understand the concepts that underpin the framework. The most important concepts are each described in a separate paragraph in the following. For a better understanding of the notions, Figure 2.1 presents the Eclipse Workbench along with its constituent components.

Workbench is the environment Eclipse provides in order to develop a product. The workbench window can contain one or more perspectives. Figure 2.1 illustrates an Eclipse workbench window.

Workspace is used to describe the directory on the hard drive that stores the projects.

The **Perspective** specifies the initial layout of the workbench window components. It contains a number of windows, menus and toolbar items that are useful

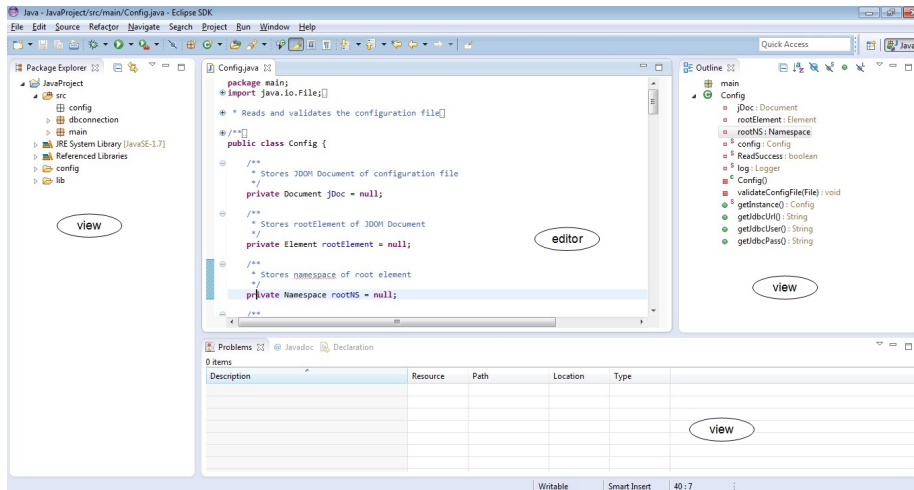


Figure 2.1: Eclipse Workbench

when accomplishing a certain work. In Figure 2.1, the active perspective is Java perspective.

Resources is the name used to refer to files, folders and projects inside the Workbench. Files and folders correspond to the files and folders existing in the file system. A project maps to a folder in the file system, but it contains beside files and folders other elements used to build, share, version control etc. the project.

Since Eclipse is an IDE, the **Editor** is in most cases the main component of a perspective. Eclipse provides editors for different types of files, and they are called *embedded editors*. If there is no editor defined for a specific type of file, Eclipse opens it using an *external editor* outside of workbench.

Actions represent the commands and user actions available in the workbench. There are three types of actions: **top level menus**, **context menus** and **toolbars**. The workbench in Figure 2.1 has 10 top level menus: File, Edit, Source and so on, and a toolbar associated to the Java perspective. A context menu is the set of actions that are allowed when rightclicking in a certain context. When a user rightclicks on a resource, inside a window or inside the editor, a popup menu appears displaying all the possible actions that can be performed for the selected entity.

Development tools

This chapter introduces the technologies and tools used throughout the entire development of eRAISE. There are four sections: BON, BONc, Beetlz and Eclipse PDE, meant to be read by people that hear for the first time about them or that have few knowledge on the subject. This chapter describes generalities and concepts, while the next chapter, chapter 4 describes where in the development process these technologies are used and how.

3.1 BON

Business Object Notation (BON) is a method used for describing and analyzing object-oriented software systems. It was promoted by Walden and Nerson in the mid-90s within the Eiffel community [WN95]. Ostroff, Paige, and Kiniry formalized parts of the BON language and reasoned about BON specifications [LOP02, Kin01, PKOL02, PO01]. Fairmichael, Kiniry, and Darulova developed the BONc and Beetlz tools for reasoning about BON specifications and their refinement to JML-annotated Java (See <http://tinyurl.com/brgcrzc> for more information). Finally, Kiniry and Fairmichael have extended BON in a variety of ways to produce Extended BON (EBON), which permits one to add new domain-specific syntax and semantics to the core BON language [Kin02].

There are three main principles that underlay the BON method [Wal]:

- seamlessness
BON provides support for the entire product life cycle by easing the transition between analysis, design and implementation. This way the collaboration between technical and non-technical people involved in different phases of the system development process, is improved.
- reversability
BON provides a smooth transition not only from analysis and design to implementation, but also backwards from implementation to design. Therefore the model of the system and the documentation are always up to date with its implementation.
- software contracts
The class specifications are expressed using assertions in terms of preconditions, postconditions and invariants, thus facilitating the exposure of contracts between classes.

BON also provides a textual and graphical notation to support the method. Therefore the notation is also build on the three principles mentioned before plus it is general, it supports scalability by providing means of grouping multiple units into higher level units and it supports typed interfaces.

Since BON is independent of any programming language, it relies only on object-oriented concepts to describe a system [WN94]. Therefore system units are **classes** that can be logically grouped into **clusters**. The existing relations between classes are described in terms of **inheritance** and **client** relations. Starting from these notions, which are further elaborated in subsection 3.1.1 and subsection 3.1.2, an object-oriented software system can be described statically and dynamically using BON **static** and **dynamic models**, respectively. The static models capture the system structure, its components and the relations existing between them at a specific moment in time. The dynamic models describe the system behavior over time, looking at how objects interact, which operations are called and what messages are being passed.

3.1.1 Static models

The BON approach uses the static models to capture the classes of a system, their interfaces, how they relate to each other, and how they are grouped in higher level units named clusters.

The static model can be described both informally so that non technical people can understand the system and formally in a more detailed manner used by designers, programmers and so on.

The informal description is captured using three modeling charts: **system chart**, **cluster chart** and **class chart** which are created using natural language. The system chart, which is exactly one per system, contains a description of the classes and clusters composing the system. The cluster charts comprises the description of the classes and other clusters that compose it. The class chart, besides the class description, contains also the information that other classes may ask from the represented class, the services that the class can provide and the rules the class and its clients must obey.

The formal description of the system is made inside **static diagrams**. They present the classes' typed interfaces, the software contracts, but also the static relations between different classes and clusters. The static relations are **inheritance** and **client, supplier relations**. The inheritance is the same concept as used in object-oriented method, while the **supplier** is the component providing an interface and all components using it are **clients**.

3.1.2 Dynamic models

The dynamic models are used to present the system behaviour and are described using three types of charts: **event chart**, **scenario chart** and **object creation chart**.

The event charts present the external stimuli that make the system react and the system responses to these stimuli; in BON terminology called **internal events** and **external events**. The scenario charts present a partial system execution as a series of events, usually starting with an internal or external event. Object creation charts present the classes that create instances of other classes and the classes that are being instantiated. And for all these, BON provides dynamic diagrams comprising the scenarios with their sequence of events and the objects.

3.2 BONc

BONc, the BON compiler, is a typechecker and a parser for BON. It takes as input one or more *.bon* files and then parses and typechecks them. The output of the tool is the list of found errors, if there are any.

Besides typechecking and parsing, BONc provides many other options. A complete list can be found at [Kin] with the most important being:

- Pretty print.

Pretty prints the input files to the standard output or to the file given as argument.

- Generate documentation.

Produces documentation from the input files either as *html* pages or as plain text files.

- Generate class dictionary.

Generates the class dictionary for the input files either as *html* pages or as plain text files.

- Generate relational graph.

Constructs a graph representing the clustering and inheritance relations existing in the input files.

BONc can be used both from command line and integrated in Eclipse. The installation details are well documented on the BONc home page <http://www.kindsoftware.com/products/opensource/BONc>.

3.3 Beetlz

Beetlz is a tool that automatically generates JML-annotated Java code from EBON specification [Dar09]. Besides code generation, Beetlz also performs consistency checking between the system modeled in BON and the system implemented in JML-annotated Java. This way any change made in the architecture, can be easily identified both in the model and in the implementation.

Beetlz can be used both from command line and integrated with Eclipse. Installation details can be found on the Beetlz home page <http://kindsoftware.com/products/opensource/Beetlz>.

3.4 Eclipse PDE

When Eclipse was created, it was intended to be easily extended with contributions from third parties. Therefore Eclipse was build around the plug-in concept, that allows new plug-ins to be build on top of existing plug-ins in order to enhance their functionality. Thus Eclipse is just a set of plug-ins build on top of a run-time engine.

In order for a plug-in to be extended it needs to define **extension points**. These are contracts that specify how to add functionality. When a new plug-in extends an existing one, it must provide an **extension** (contribution) that conforms with the contract specified by the existing plug-in's extension point.

Multiple plug-ins that can be combined to perform a certain task, can be grouped together in a **feature**.

Although Eclipse is build of many plug-ins and it needs to load them all up during start-up, Eclipse manages to start quite fast (a couple of seconds). This is possible since plug-ins have a **declaration** part and an **implementation** part, which are separated [Gam04]. Therefore, when Eclipse starts up it only uses the declaration parts of the plug-ins in order to show the user what are its capabilities. And only when a user decides to use a certain plug-in e.g., by clicking on a button, the Eclipse loads the plug-in implementation.

To be more specific, the declaration of a plug-in is realized through an **manifest file**. This file contains the declaration and description of the plug-in services and dependencies e.g., if the plug-in creates a button, the manifest file stores the name of the button, the icon that visually represents the action, the button type e.g., push, radio, pull-down. The entire functionality of the contribution is made in Java, and the manifest file only stores the link to the Java part that implements the associated functionality.

Eclipse Plug-In Development Environment (PDE) provides support for creating Eclipse plug-ins, taking into account all the aspects that were previously mentioned in this subsection. Eclipse PDE is build on top of Java Development Tools, providing all the JDT support for writing the plug-in functionality in Java.

Eclipse PDE provides tool support to create, develop, debug, test and build an Eclipse based product.

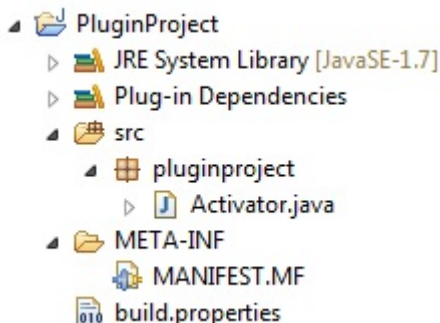


Figure 3.1: A plug-in structure example

3.4.1 New Plug-in Project wizard

Eclipse **New Plug-in Project wizard** provides help for creating the plug-in structure and initial information. The initial information contains aspects like the plug-in name, its version, identifier, which version of Eclipse is the plug-in targeted to work with and so on. The New Plug-in Project wizard also offers the possibility to start the project from a template, in which case the wizard automatically generates the pieces required to provide the service.

After completing the information required by the wizard, the new plug-in project is created having the structure presented in Figure 3.1.

The manifest file is called **MANIFEST.MF** and is stored in an **META-INF** directory. The manifest file stores the project information and the dependencies of the project.

The wizard can also generate a manifest class, called **Activator.java**, located in Figure 3.1 under the **src** folder. This class is the first class instantiated and it is used by Eclipse throughout the entire plugin life. This class also provides methods for accessing the workbench resources and setting plug-in preferences.

3.4.2 Plug-in Manifest Editor

Eclipse PDE offers a complex editor for the plug-in manifest file, but not only. It is called **Plug-in Manifest Editor** and contains 9 pages: Overview, Dependencies, Runtime, Extension, Extension Points, Build, Manifest.MF, build.properties and plugin.xml. When a new extension, or extension point is added, PDE auto-

matically creates a new file called **plugin.xml**. This stores information about the provided extensions or defined extension points and its source can be viewed in the plugin.xml page of the manifest editor.

3.4.3 Plug-in debug

Since PDE is build on top of JDT, the plug-in can be debugged using the **JDT Debug component**. Therefore the plug-in debug process is quite similar to a Java project debug, using setting breakpoint, stepping through the code, suspending threads and analyzing variables at run time facilities. The difference is that the plug-in is launched in the **Runtime Workbench** environment.

3.4.4 Plug-in tests

The easiest way to test a plug-in is by doing manual testing. However, using the **JUnit framework** a more systematic testing can be realized. Thus, a new plug-in must be created that has two dependencies: the tested plug-in and the **org.junit4**. Then test fixture are created to test different aspects of the plug-in and run in the JUnit window.

Analysis and design

This chapter describes the analysis and design phase of the eRAISE system development. The method that has been used is based on EBON and has made the subject of a paper [FK13], written by the author of this thesis and her supervisors, that can be found in Appendix A and that will be presented during International Conference in Software Engineering Workshop 2013 [top].

4.1 Analysis and design method

For the analysis and design part of the eRAISE system it was decided to go for a new approach based on EBON method. The reason behind this decision is that currently there are no published methodologies for plug-in development, although some work was done in this direction e.g., Lamprecht et al. discusses some principles in plug-in development [NLS⁺12]. The explanation for the lack of methodologies in this area could be that plug-in development has only recently become the focus of scientists.

There are many reasons why EBON was preferred over the more famous UML, as a modeling language. One is that EBON semantics is more clear and unambiguous than UML. Furthermore, EBON is easier to use and to learn and

it is more concise. But more important, EBON was designed to support seamlessness, reversibility and design by contract. Seamlessness and reversibility are the basis for a maintainable product, while the design by contract technique assures its reliability. All these statements are sustained in [Pai99] where the two methods are discussed and compared. This technical paper brings many more other arguments in favour of BON and emphasizes UML deficiencies e.g., EBON has only one classifier, the class, compared with UML that has eight: class, datatype, use case, interface, component, node, signal and subsystem. The big number of classifiers in UML leads to redundancy and ambiguity and compromises the seamlessness principle e.g., the *datatype* and *interface* classifiers can both be covered by the *class* construct.

The methodology used to analyze and design eRAISE is based upon BON methodology. The methodology, as applied to eRAISE development, has six stages:

- domain modeling
- user interface
- events
- components
- components communication
- code generation

Each of these steps is further described in a separate section and presented in the order they are applied. The sections describe the guidelines for completing the steps, the idea behind them and the output of each phase. Besides describing the methodology, the following sections present also the eRAISE system analysis and design, making eRAISE act as a case study for the methodology. The EBON syntax is also described as the examples are being shown.

In the eRAISE case, the six steps are applied sequentially, one after the other, where the deliverables of previous steps are used as starting points for next steps. But this does not mean that we can not return to previous steps and make refinements when necessary. Once the six steps are completed they can be retaken in a new iteration bringing a new series of refinements. Thus the process that we have used is both incremental and iterative. However, the method can be used in many ways e.g., sequence using the Waterfall model or in an iterative manner e.g., using Spiral model and it is up to the user of the method to decide what works well for the project under development and for the team developing it.

4.2 Domain modeling

The first step when analyzing and designing a system is to establish its domain model. This is done in order to create a common vocabulary between those involved in the project and to identify the concepts used in the product development process. This means that the most important entities and high level classifiers related to the system domain must be identified, explained and documented from the very beginning, so they can be unanimously understood and used throughout the entire product life cycle.

Starting from the project name and description, the domain model is constructed by analyzing areas like Eclipse, RAISE and graphical user interface. All notions and terms introduced in the chapter 2 are used as the starting point of this phase. The result is a list of terms along with their explanation, essentially describing entities and elements from a high level point of view. Some examples from the list are notions like *RSL perspective*, *editor*, *typechecker*, *console*, *SML translator*, *tests runner*, *L^AT_EX generator*, *GUI handlers* and so on.

When performing domain analysis we try to identify concepts that are redundant, which concepts relate to others, etc.. Some of these items can be grouped in a more general notion, while others are big enough to cover multiple notions. For example, *RSL Perspective* can be seen as a notion that comprises all other items since inside Eclipse all RAISE elements can be grouped under a single perspective. Likewise, *core* can be a notion that comprises components like typechecker, SML translator, tests runner and L^AT_EX generator.

Such notions are captured using the EBON *system_chart*, *cluster_chart* and *class_chart* elements. These charts describe the system informally, using natural language and therefore they are perfect for this phase of analysis and design, where a common vocabulary needs to be established, understood and documented. Inside the three charts, the notions that have been identified are documented as *classes*, which can be grouped under *clusters* and all these make up an unique *system*.

The eRAISE domain model is captured using EBON textual notation and presented in Appendix B. Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4, Figure 4.5 and Figure 4.6 present also the eRAISE domain model, but in the form of *html charts*. The html charts are html documentation generated by applying BONc on the EBON textual representation of the domain model. The reason for using the documentation charts here instead of the textual notation is because they are friendlier to the nontechnical people that may be involved in this early phase of the analysis and design.

SYSTEM		eRAISESystem	Part: 1/1
PURPOSE		INDEXING author : "Marieta Vasilica Fasie";	
Cluster	Description		
RSLPerspective	"The Eclipse RAISE perspective. It contains all components and functionality relevant for a RAISE project"		

Figure 4.1: eRAISE system chart

CLUSTER		RSLPerspective	Part: 1/1
PURPOSE		INDEXING	
Cluster	Description		
core	"Comprises functionality for typechecking an RSL specification, translating it to SML, executing the test case and for generating the Latex documentation"		
editor	"Groups all editor specific functionality"		
testcases	"Contains the elements displaying the RSL test cases"		
wizard	"Contains all components that are dealing with the creation of a new RSL project"		

Figure 4.2: RSLPerspective cluster chart

Inside the domain model, there is only one, unique system, named *eRAISESystem* captured in a `system_chart` and presented in Figure 4.1. The *eRAISESystem* contains only one cluster that groups all the other concepts. *RSLPerspective*, presented in Figure 4.2, covers four clusters named *core*, *editor*, *testcases* and *wizard* whose descriptions are captured in the `system_chart`, using natural language.

The four clusters are further detailed in a `cluster_chart` each. The *core* cluster, presented in Figure 4.3, comprises all concepts referring to the core functionality: `TypeChecker`, `SMLTranslator`, `LatexGenerator`, `TestRunner`. We also decided to add here a `Console` and a `ResourceHandler` concept and the concepts representing GUI elements. The latter are grouped under one cluster named *guihandler* as it is captured in Figure 4.3.

The *editor* cluster presented in Figure 4.4, groups all concepts related to the editing of an RSL specification. Inside the *editor* cluster, the *config* cluster groups all the elements used for configuring the RSL editor.

Figure 4.5 presents the *testcases* cluster `cluster_chart` comprising two other clusters: *model* and *ui*. The *wizard* cluster captured in Figure 4.6 groups the concepts that compose a wizard: `NewRSLProjectWizard`, `RSLProjectPage`.

CLUSTER		core	Part: 1/1
PURPOSE		INDEXING	
Class	Description		
Console	"Displays the output of different components e.g. TypeChecker, SMLTranslator"		
ResourceHandler	"Handles the actions done on workspace resources"		
SMLTranslator	"Translates RSL specifications to SML code"		
LatexGenerator	"Integrates RSL specification in Latex"		
TestRunner	"Executes the SML files"		
TypeChecker	"The RSL syntax and type checker"		
Cluster	Description		
guihandlers	"Comprises all UI handlers"		

Figure 4.3: core cluster chart

CLUSTER		editor	Part: 1/1
PURPOSE		INDEXING	
Class	Description		
ConsoleToProblems	"Connects the Console and the Problems View"		
Problem	"Represents an RSL type check error"		
ProblemsView	"Displays any problem existing in the current workspace"		
RSLEditor	"The RSL text editor"		
Cluster	Description		
config	"Contains components that configure the editor for the RSL specification"		

Figure 4.4: editor cluster chart

Besides introducing notions and describing them, the domain model also describes how concepts behave and how their behavior is constrained. In EBON, behavior is specified by using two concepts called *queries* and *commands*, collectively known as *features*. Behavioral constraints are specified using an EBON concept called *constraints*. A *command* is a service that a *class* provides, that changes the state of the object that implements the class, while a stateless *query* is a request for information from a specific object.

Therefore, for each concept previously identified, one must think of its behavior and the constraints that surround it. This additional information also describes the system informally, from a high level and its role is to help with the later design decisions. e.g., in the eRAISE domain model, there is the *Console* concept that represents a UI element displaying the output to the user. Thus, it

CLUSTER		testcases	Part: 1 / 1
PURPOSE		INDEXING	
Cluster	Description		
model	"Comprises the classes creating the test cases model that is displayed inside RTestView"		
ui	"Contains the classes displaying the test cases results"		

Figure 4.5: testcases cluster chart

CLUSTER		wizard	Part: 1 / 1
PURPOSE		INDEXING	
Class	Description		
NewRSLProjectWizard	"RSL new project wizard"		
RSLPerspective	"Groups the RSL associated views and actions"		
RSLProjectPage	"Page collecting the user input"		

Figure 4.6: wizard cluster chart

offers the service of displaying informative and error messages and the possibility to clear the output. These two services are captured inside the `Console` class `_chart`, using the EBON commands, and presented in Figure 4.7. Within the `_chart` there is also a *constraint* stating that the console output must be cleared before displaying a new message.

The complete domain model concepts, their behaviour and constraints can be found in Appendix B in EBON textual notation.

4.3 User Interface

The purpose of this step is to determine the plug-in functionality from the user's point of view. This means identifying all the things a user can do from the plug-in's UI. The UI feature set consequently derives the requirements of the plug-in and designs the UI in the same time. Therefore, for each user action that is relevant and important for the plug-in, a mock-up user interface is created. If many user actions are similar, they can be grouped under a single user interface.

The mock-up user interface can be a vague handmade sketch or a precise drawing

CLASS	Console	Part: 1/1
TYPE OF OBJECT "Displays the output of different components e.g. TypeChecker, SMLTranslator"	INDEXING in_cluster : "RSLPerspective";	
Queries		
Commands	"Display informative or error messages", "Clear console content"	
Constraints	"Delete content before displaying a new message"	

Figure 4.7: Console class chart

made with an advanced graphical editing program. The intention here is not presentation and precision, but instead feature completeness and UI consistency. In this paper the mock-ups were created by taking a screenshot of Eclipse and then hand-editing the resulting image.

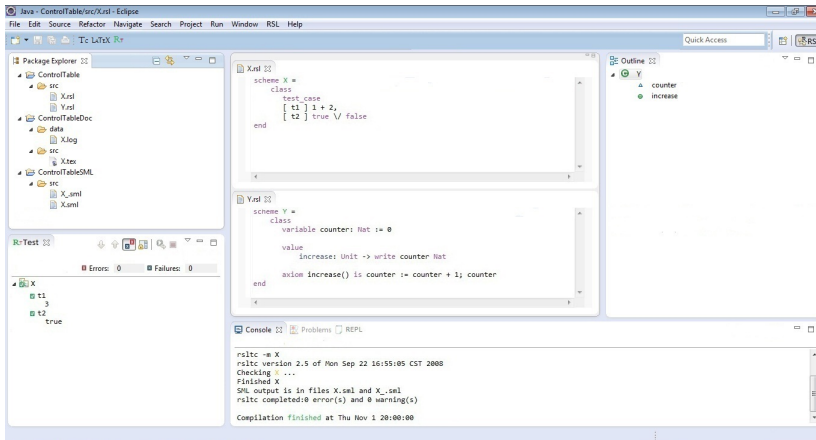


Figure 4.8: The RSL Eclipse perspective

For the eRAISE plug-in, it was decided that all UI elements will be displayed under the same Eclipse perspective, named RSL perspective, in order to group the information and favour the usability. The RSL perspective is presented in Figure 4.8 and contains 6 views: PackageExplorer, RTest, Console, Problems, REPL and Outline grouped around the RSL editor, an RSL menu and RSL toolbar items.

Inside the RSL perspective, the user should have the possibility to typecheck all RSL files existing in the workbench. Also, the user should have the possibility

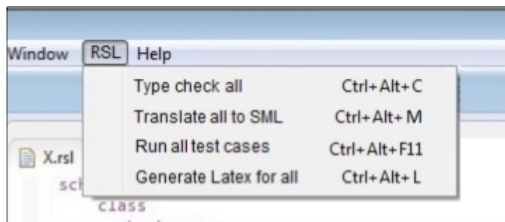


Figure 4.9: The RSL menu item

to translate all RSL files in workbench to SML, to run all test cases existing in all files, and to generate \LaTeX documents for them. Therefore, it was decided that these four actions should be grouped under the RSL menu item and presented in the same UI, for consistency and simplicity. Figure 4.9 illustrates the RSL menu item as part of the Eclipse IDE menubar. Each submenu has an associated keyboard shortcut e.g., the *Type check all* submenu has associated the $\text{CTRL}+\text{Alt}+\text{C}$ keyboard shortcut.

SCENARIOS	RSLMENU	Part: 1/1
COMMENT	INDEXING	
<p>"TCAIIMenu" "The user can type check all RSL files in the workspace. Success or failure messages will be displayed along with the list of errors in case of a failure"</p>		
<p>"SMLAIIMenu" "The user can translate to SML all RSL files in the workspace. Success or failure messages will be displayed along with the list of errors in case of a failure"</p>		
<p>"RunAllIMenu" "The user can run all test cases in the workspace. Success or failure messages will be displayed along with the list of errors in case of a failure"</p>		
<p>"LatexAllIMenu" "The user can generate Latex files for all files in the workspace. Success or failure messages will be displayed along with the list of errors in case of a failure"</p>		

Figure 4.10: Scenario chart for the RSL Menu

While the user interface is being drawn, product requirements are documented using EBON *scenario_chart* elements. The beautiful part about using EBON is that it allows the requirements specification to be captured using natural language. Therefore no intermediate step is required between identifying the requirements and documenting them. The requirements associated with the RSL menu in Figure 4.9 are captured in the html chart in Figure 4.10. For each submenu item there is a *scenario* element defined by a name and a description. The four scenarios are grouped under a *scenario_chart* associated to the Eclipse RSL menu. For the *scenario_chart* expressed in EBON textual notation please

refer to Appendix D.

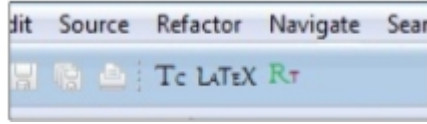


Figure 4.11: The RSL buttons

SCENARIOS	BUTTONS	Part: 1/1
COMMENT	INDEXING	
<p>"TCButton" "The user clicks on the typecheck button to typecheck the currently active file."</p>		
<p>"RunButton" "The user clicks on the run test cases button to run the testes in the currently active file."</p>		
<p>"LatexButton" "The user clicks the Latex button to generate latex for the currently active file"</p>		

Figure 4.12: The RSL buttons scenario chart

The eRAISE GUI provides also buttons for taking actions on the active RSL file that is opened inside the editor. Thus the user can typecheck, run the tests and generate \LaTeX for the file opened in the editor. Figure 4.11 illustrates the three buttons as part of the Eclipse GUI toolbar. The requirements associated with these three toolbar items are presented in the scenario_chart in Figure 4.12.

And the same technique is applied for all the other GUI elements. First the GUI interface is created and then the user requirements identified and documented using EBON. The complete list of UI mock-ups is presented in Appendix C , while their associated requirements are captured using EBON textual notation in Appendix D.

4.4 Events

In this stage of the method the entire system is seen as a black box. The focus is on the external actions that make the system react and on the system's outgoing responses. More formally, within EBON, scenarios are composed of events, thus there is a refinement relationship between scenarios and events.

An *incoming external event* is any action that determines the system to change its state. For example it can be a user clicking a button or another system

```

event_chart UserActions
incoming
explanation "External events triggering representative system behaviour"
event "TYPECHECKALL: User clicks RSL menu and then clicks on Type Check
      all option or presses Ctrl+Alt+C" involves ResourceHandler,
      TypeChecker, Console, ConsoleToProblems, ProblemsView

```

Listing 4.1: Incoming event chart for typechecking features.

```

event_chart UserMessages
outgoing
explanation "Internal events triggering responses meant to inform the user."
event "CONSOLEUPDATE: Success or failure messages displayed in console"
involves Console, TypeChecker
event "PROBLEMSUPDATE: Problems view update"
involves TypeChecker, Console, ConsoleToProblems, ProblemsView

```

Listing 4.2: Outgoing event chart for typechecking features.

sending a request. An *outgoing internal event* is the response the system sends to an incoming external event. The system outgoing event for the action of pressing the button could, e.g., be the display of a new window or writing a message to the standard output.

Looking back at the RSL menu scenario presented in Figure 4.10, the user has the possibility to type check all RSL files. This is illustrated in Figure 4.9 by the presence of a submenu item named *Type check all*. Therefore, the *incoming external action* in this case is: *the user selects the Type check all submenu item*. And this external event has been determined just by looking at the scenarios previously identified. However, there is another user event that triggers the same system reaction and that is using the keyboard shortcut: *The user presses Ctrl+Alt+C*.

Once established, the user actions are captured in EBON using *event_chart* elements. The *event_chart* is either *ingoing* or *outgoing* depending on the type of the events they capture. Since the two user incoming events that have just been identified aim for the same functionality, they are considered the same event (TYPECHECKALL) and captured in Listing 4.1. The *involves* part in Listing 4.1 is explained later, in section 4.6, in detail, as it denotes component communication patterns.

If an incoming action triggers changes in the system state, the next task is to decide how the system should respond to the action, and what are the changes that have taken place. For the eRAISE case study, it was decided that, after the user selects the *Type check all* submenu item, all RSL files in the workbench are typechecked and a message for each typechecked file should be displayed in

```

scenario_chart RSLMENU
scenario "TCallMenu"
description "The user can TYPECHECKALL RSL files in the workspace. This implies
    PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "SMLaLLMenu"
description "The user can SMLTRANSLATEALL RSL files in the workspace. This implies
    PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "RunAllMenu"
description "The user can RUNALLTESTS cases in the workspace. This implies
    RTESTUPDATE, PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "LatexAllMenu"
description "The user can GENERATELATEXALL for all files in the workspace. This
    implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"

```

Listing 4.3: Scenario chart comprising the events for the typechecking feature.

the standard output. The messages inform the user about how the typechecking evolved and since the case study GUI is Eclipse based, the standard output is considered by default the Eclipse *Console view*. However, in some cases, the typechecking may not be successful due to errors in the input files. In this case it would be nice to know what caused the problem and where it can be found. Thus, the system should provide the necessary information in the Eclipse *Problem view*. To sum up, after the user selects the *Type check all* submenu item, the system updates the *Console* and *Problem* views with appropriate information. Listing 4.2 presents the two events captured in an outgoing *event_chart* under the names of *CONSOLEUPDATE* and *PROBLEMSUPDATE* respectively.

Once the events have been identified and given a proper name, they can be used to rewrite the scenarios identified in section 4.3. The reason for doing this is to emphasize the actions a user takes during a scenario and the responses the system must provide. Only the event name is used inside the scenario description, making it shorter, less open to interpretation and conciser. Listing 4.3 illustrates how the *MENU* scenario description presented in Figure 4.10 changes once the events names are being added e.g., "user can type check all" description was replaced with "user can TYPECHECKALL", where Listing 4.1 describes exactly what the TYPECHECKALL event implies. The "Success or failure messages will be displayed along with the list of errors in case of a failure" has been replaced by "This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE", where PROBLEMSUPDATE and CONSOLEUPDATE events are presented in Listing 4.2. EDITORUPDATE action is an outgoing event that displays a marker in the editor at the line where an error has been discovered.

The rest of the events are identified in the same way, by starting from the user interfaces and scenarios determined in section 4.3. The events are given names, or are grouped under the same name if multiple events trigger the same system reaction, and captured inside event charts. Then the events' names are used to

replace the event description inside the scenario charts. The complete list of the internal and external events is presented in Appendix D using EBON textual notation.

4.5 Components

At this stage in the methodology, in contrast with the previous one, one looks *inside* the system at the components that constitute its architecture. Components refer either to concrete elements, e.g., a *Zoom in button*, or abstract concepts, like *User authentication* which covers everything in the system responsible for authenticating a user.

The place to start identifying the system architecture components is the domain model presented in section 4.2. The high level classifiers captured there must be transformed into concrete data types in order to bring the system development closer to the implementation phase. The advantage of using EBON is that it simplifies the transit between the domain model and architecture and manages to capture the concrete data types in a language independent fashion. This is done by taking the entities captured in the *system chart*, *cluster charts* and *class charts* and transfer them into *static diagrams*. An EBON *static diagram* contains multiple *components* which can be *clusters* or *classes* and which have the same meaning as the ones composing the *system chart* in section 4.2.

Applying this step on eRAISE, the system chart presented in Figure 4.1, in section 4.2 becomes the *static diagram SystemArchitecture* from Listing 4.4. Listing 4.4 presents only a caption and not the entire components which can be found in Appendix F.

4.6 Components Communication

This section's concern is how components interact with each other and what interfaces they present to the other components that want to communicate with them. The starting point is the list of incoming and their corresponding outgoing actions identified in section 4.4. This helps identifying the components that react to an external stimuli and the components responsible for the outgoing actions. Once the starting and ending point of the data flow is established, the other interacting components are determined by evaluating the scenarios in section 4.3.

```

static_diagram SystemArchitecture
--Shows the architecture of the eRAISE plugin system
component
  cluster core
  component
    class Console
    class ResourceHandler
    class SMLTranslator
    class LatexGenerator
    class TestRunner
    class TypeChecker

    cluster guihandlers
    component
      class TCHandler
      class SMLHandler
      class RTHandler
    end
  end

  cluster editor
  component
    class ConsoleToProblems
    class Problems
    class ProblemsView
    class RSLEditor
  ...

```

Listing 4.4: Static diagram comprising the eRAISE components

In EBON notation, the components communication is seen in terms of *client*, *supplier* relationship. The component providing the interface is a *supplier* and all components using it are *clients*.

In eRAISE, one incoming event is *TYPECHECKALL* presented in Listing 4.1, in section 4.4 and its corresponding outgoing events are *CONSOLEUPDATE* and *PROBLEMSUPDATE* presented in Listing 4.2, in section 4.4. The *TYPECHECKALL* action must trigger the *TypeChecker* in order to typecheck all RSL files in the workspace. The component responsible for the *CONSOLEUPDATE* is the *Console* and the one for *PROBLEMSUPDATE* is *Problems*. The *TypeChecker* component typechecks the input and directly informs the *Console* about the status. Therefore, the *TypeChecker* is a client of *Console* and this is expressed in EBON as: *TypeChecker client Console*. The client relations must be added in the *static_diagram* after the components declaration.

Since *TypeChecker* is stimulated by the *TYPECHECKALL* event and it is a client of *Console* which generates the outgoing response, it can be said that *TYPECHECKALL* event involves the *TypeChecker* and the *Console* components. And this is how the *involves* part in Listing 4.1, section 4.4 is constructed. The same method is applied to *PROBLEMSUPDATE* event. This event is triggered if there are errors displayed in the console. Therefore *TypeChecker* sends

```
class ProblemsView
  feature
    update
      -> problems: SET[PROBLEM]
  end
...

```

Listing 4.5: ProblemsView component interface

a message to *Console* and if the message is an error, a third component called *ConsoleToProblems*, that monitors the *Console*, notifies the *Problems* component. Therefore, the *PROBLEMSUPDATE* event involves the *TypeChecker*, *Console*, *ConsoleToProblems* and *Problems* components. This is captured in Listing 4.2, section 4.4.

Once it was decided what components are interacting, it must be established how to do so. This means establishing the contracts between components by identifying the information a client needs and the messages it sends to its supplier. EBON supports the formal specification of typed interfaces in a programming language-independent fashion. Classes are parameterized and contain formally specified *features*. Each classifier in the domain model maps to exactly one class within the formal model, and each feature of each class within the domain model maps to one formally specified feature in that class' interface.

For example, the *ProblemsView* component has a feature called *update* which allows the client components to send the problems that will further be displayed to the user in the ProblemsView. The situation is captured in Listing 4.5, where a caption of the components typed interfaces is presented. EBON also supports preconditions and postconditions specified using the *require* and *ensure* keywords. The complete list of components and their typed interfaces can be seen in Appendix G.

Once the components and their interaction is established the system architecture is complete. Figure 4.13 presents eRAISE system architecture using EBON graphical notation.

4.7 Code Generation

Once the analysis and design parts are finished, the next step is to generate the formally specified code skeleton. This step is accomplished using the Beetlz tool, which was described in chapter 3 and which automatically generates JML-annotated Java code from an EBON specification. The input of this tool is the

```
public /*@ nullable_by_default @*/ class ProblemsView {  
    public void update(Set<Problem> problems){}  
}
```

Listing 4.6: Java ProblemsView class generated by Beetlz

EBON *system_chart* and *static_diagram* that were obtained throughout the entire analysis and design. With one click, Beetlz converts all EBON specifications into JML-annotated, Javadoc documented Java code. Beetlz also performs refinement analysis so that architecture drift is automatically identified as the system evolves, either at the model-level in EBON, or within the implementation in Java.

For example, Listing 4.6 shows the *ProblemsView* class generated by Beetlz for the component with the same name described in the *static_diagram* in Listing 4.5. The entire skeleton code generated by Beetlz can be seen in Appendix H.

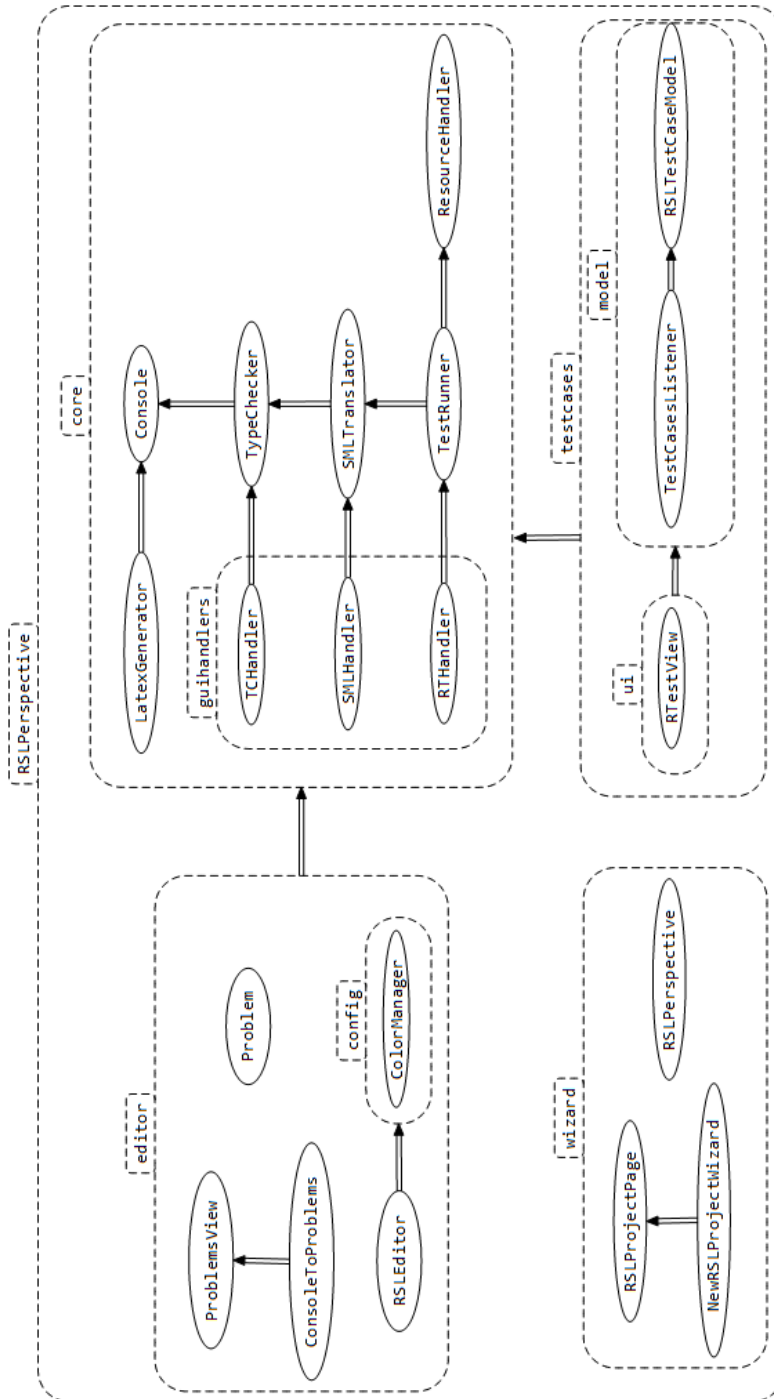


Figure 4.13: Static diagram

Implementation

This chapter covers the project implementation phase. The first subsection describes the plan after which the implementation was conducted starting from the artifacts obtained in the analysis and design. The rest of the subsections describe a plug-in each, offering code examples and reasons behind implementation decisions.

5.1 Plan

The starting point of the implementation is the skeleton code generated in the end of the analysis and design phase that can be found in Appendix H. Starting from this code and the scenarios identified in the same section, the plan of the implementation is established as further described.

All scenarios captured in Appendix D are added in the Mylyn task list as shown in Figure 5.1. The tasks are grouped based on the functionality they refer to, under different categories e.g., all tasks that refer to type checking are grouped under the *Type check category*, all task that involve an RSL translation to SML are grouped under the *SML translate category* and so on. Then each task is taken individually and the code that solves it is written. When a task is completed

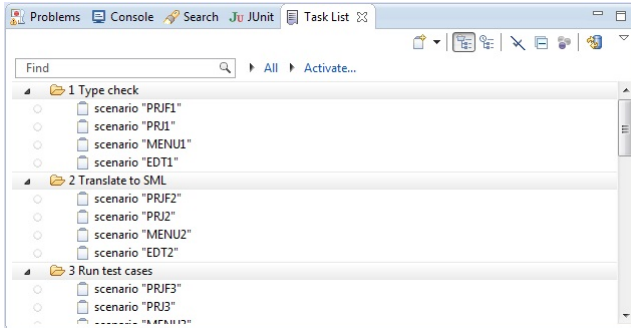


Figure 5.1: Mylyn task list comprising the EBON scenarios

the next one is started and so on, until all tasks are implemented.

The analysis and design phase has already defined the plug-ins that compose eRAISE in the form of the four clusters defined in the eRAISE system_chart Appendix B. Thus the four implemented plug-ins are:

1. `rsl.typechecker` plug-in
The core of the eRAISE system. Encapsulates the functionality for type-checking a RSL specification, for translating it to SML, for executing the test case and for generating the \LaTeX documentation.
2. `rsl.editor` plug-in
Implements the RSL editor and all editor specific functionality e.g., keywords highlighting, displaying the typecheck errors' locations inside the RSL specification, file saving actions and so on.
3. `rsl.testcases` plug-in
Interpreting the output of the SML execution of the RSL test cases and display the results.
4. `rsl.wizard` plug-in
This plug-in contains the implementation of the wizard that creates a new RSL project, plus groups all the RSL views and actions.

Each plug-in implementation is further described in a separate subsection along with the scenarios that it covers.

The implementation description is supported with small examples from the Java code. The examples can be an entire method or just a few lines of code meant to exemplify how the technical solution was implemented. Also from the code

examples, the *try/catch* blocks have been removed as any of the *log* information, in order to make the examples easier to read. The entire source code behind eRAISE can be found online on GitHub under <https://github.com/kiniry/eRAISE>.

5.2 rsl.core plug-in

5.2.1 Type check

The implementation starts by solving the scenario presented in Listing 5.1.

```
scenario "PRJF1"
description "The user can TYPECHECK one RSL file. This implies CONSOLEUPDATE, PROBLEMSUPDATE
and EDITORUPDATE"
```

Listing 5.1: Caption from PROJECT_EXPLORE_FILE scenario

Since the existing *rsltc* tools set already offers the functionality of typechecking a RSL specification, the *rsltc* tool is integrated inside eRAISE and executed every time a file needs to be typechecked. In order to execute *rsltc* from inside the plug-in, it must be run as a Java external application. This is done by creating a new *ProcessBuilder* having as parameters the location of the *rsltc* tool and the location of the RSL specification that needs to be typechecked. But in order to execute a DOS command from a Java program, it must be wrapped in the **cmd** as it can be seen in Listing 5.2. The `/c` switch closes the command shell after the command has finished executing. After the process has been started in a new thread, its output is collected line by line. This output is the output of the *rsltc* typechecker and informs about how the typechecking of the RSL specification went. This output is what is displayed in the Console view.

```
String commands[] = {"cmd", "/C", programPath, filePath};
ProcessBuilder builder = new ProcessBuilder(commands);
//correlate the error messages with the output messages
builder.redirectErrorStream(true);
//start the program
process = builder.start();
//get the input stream
InputStream is = process.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
//read the program output line by line
String line = br.readLine();
while (line != null && ! line.trim().equals("--EOF--")) {
    infomessage += line+"\n";
    line = br.readLine();
}
}
```

Listing 5.2: Executing *rsltc* as a separate process

Since many other tools may depend on the output of the typecheck e.g., if an RSL specification is not correct, then it can not be translated to other languages, it was decided to create an extension point through which other plug-ins can listen to the output of the typecheck. More details about this is presented in section 5.6.

Once the typecheck functionality is implemented, the next step is to create the RSL file context menu inside the *Package Explorer* view, so that the user can select it and trigger the typechecking. The context menu item is created as an additional element to the popup menu in the PackageExplorer view as illustrated in Listing 5.3. The *visibleWhen* assures that the context menu item is being displayed only on the RSL files. The fact that no restriction on the number of selected files exists, means that the user can select multiple RSL files and still see and select the *Type check* context menu item.

```

<menuContribution allPopups="false"
  locationURI="popup:org.eclipse.jdt.ui.PackageExplorer?after=additions">
  <command
    commandId="core.command.typecheckelement"
    label="Type check"
    style="push">
    <visibleWhen checkEnabled="false">
      <iterate>
        <adapt type="org.eclipse.core.resources.IResource">
          <test
            property="org.eclipse.core.resources.extension"
            value="rsl">
          </test>
        </adapt>
      </iterate>
    </visibleWhen>
  </command>
..

```

Listing 5.3: Creating a RSL file context menu inside the Package Explorer

Once the command was triggered the class implementing the handler of the command, in this case the *TypeCheckHandler* class, is called. It reads the selection, as presented in Listing 5.4 and calls the *typecheck* method with the selected file as a parameter.

```

//get the selections in the Project explorer
IStructuredSelection selection = (IStructuredSelection) window.getSelectionService().
getSelection("org.eclipse.jdt.ui.PackageExplorer");

```

Listing 5.4: Identifying the selected elements inside the Project Explorer view

Once the type check of one file is implemented, the additions of the scenarios presented in Listing 5.5 becomes quite straight forward. The idea is to create the GUI elements inside the *plugin.xml* file and to limit their visibility to the

situations where they can be applied. Once the actions are triggered, the type of the element that was selected for typecheck, must be analyzed. If a project was selected for typecheck, then all RSL file inside the project must be type checked. If the RSL menu from the menu bar was clicked then all RSL files from the workspace must be identified and called typecheck for each of them. If the user has selected the *Type check* option inside the RSL editor, then the typecheck will be called for the currently opened file inside the editor.

```

scenario "PRJ1"
description "The user can TYPECHECKPRJ RSL files in a project. This implies
PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "MENU1"
description "The user can TYPECHECKALL RSL files in the workspace. This implies
PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "EDT1"
description "The user can TYPECHECK the active file. This implies
PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"

```

Listing 5.5: Other scenarios concerning the type check functionality

5.2.2 SML translate

The implementation of the SML translation is similar to that of the typechecker. We are starting by implementing the SML translation of a file captured in Listing 5.6 and then generalize it for multiple RSL modules.

```

scenario "PRJF2"
description "The user can translate to SML one RSL file. This implies CONSOLEUPDATE,
PROJECTEXPLORERUPDATE, PROBLEMSUPDATE and EDITORUPDATE"

```

Listing 5.6: Caption from PROJECT_EXPLORE_FILE scenario

rsrtc tools set already offers the functionality of translating to SML a RSL specification, thus we are executing it from inside eRAISE every time RSL modules need to be translated to SML. As mentioned in the typechecker case, *rsrtc* must be run as a Java external application and listened for the SML translation output. The output is displayed in the Console view.

After the SML translator has been run, two new SML files are being created in the same directory as the RSL specification that has been translated. The challenge here is that *rsrtc* does not offer the possibility to choose the location of the SML files. They are by default created in the same directory as the RSL specification and this aspect can only be changed by modifying the *rsrtc* source code. In our scenarios, the SML files must be stored in a new SML project associated with the RSL project containing the RSL specification. Thus, there are two possibilities for solving the SML files location problems: either we change

the source code of `rsrtc` or move the files after their creation in the appropriate project. We decided to go with the second approach since moving resources in Eclipse is fast enough to not be visible to the user, plus it will not interfere with the user activity. Furthermore, modifying `rsrtc` source code is not our intention since applying a small patch or quick fix could lead to other problems.

Once the SML translator functionality has been implemented, the *SML Translate* user actions must be made visible in the appropriate menus and windows, just like in the type checker situation presented in Listing 5.3.

```
scenario "PRJ2"
description "The user can SMLTRANSLATEPRJ RSL files in a project. This implies
PROBLEMSUPDATE, CONSOLEUPDATE, PROJECTEXPLORERUPDATE and EDITORUPDATE"
scenario "MENU2"
description "The user can SMLTRANSLATEALL RSL files in the workspace. This implies
PROBLEMSUPDATE, CONSOLEUPDATE, PROJECTEXPLORERUPDATE and EDITORUPDATE"
scenario "EDT2"
description "The user can SMLTRANSLATE the active file. This implies
PROBLEMSUPDATE, CONSOLEUPDATE, PROJECTEXPLORERUPDATE and EDITORUPDATE"
```

Listing 5.7: Other scenarios concerning the SML translate functionality

In the end, SML translation functionality is generalized for multiple RSL files to cover the scenarios presented in Listing 5.7

5.2.3 Test cases execution

The test cases execution implementation starts with the scenario presented in Listing 5.8.

```
scenario "PRJF3"
description "The user can RUNTESTS in one RSL file. This implies RTESTUPDATE,
PROBLEMSUPDATE, PROJECTEXPLORERUPDATE and CONSOLEUPDATE"
```

Listing 5.8: Scenarios concerning the test cases execution functionality

In order for the RSL test cases to be interpreted, the RSL specification needs to first be translated to SML and then the SML file must be executed with *smlnj* [SML]. In order to execute *smlnj* from inside the plug-in, it must be run as a Java external application. This is done by creating a new *ProcessBuilder* having as parameters the location of the *smlnj* (which is the *resources* folder inside the plug-in) and the location of the SML specification that needs to be executed. Also the process environment variables must be set as in Listing 5.9 in order for the JVM to find *smlnj* binaries. The *smlpath* variable contains the location where *smlnj* is stored inside the plug-in.

```
String commands[] = {"cmd", "/C", "sml", smlfilePath};
ProcessBuilder builder = new ProcessBuilder(commands);
Map<String, String> env = builder.environment();
env.put("Path", smlpath);
env.put("SMLNJ_HOME", pathVal);
env.put("RSLML_HOME", rslPathVal);
```

Listing 5.9: Setting process environments fro running smlnj

Executing now a SML file that was obtained from an RSL specification will not work. And the reason for this is that when `rsltc` translates a RSL specification in SML it creates two files e.g., `test.sml` and `test_.sml`. The first one contains the files and libraries that need to be loaded and then calls the execution of the second file that contains the RSL specification translated in SML. Inside this first SML file, `rsltc` tool hardcodes the location of the `rslml.cm` file to `/usr/share/rsltc/sml/rslml.cm`. `rslml.cm` file contains the group of files that need to be used in the execution process and the runtime can not find it. Thus we need to edit the SML file with its correct location during runtime, which is `system_path/resources/raise/sml/rslml.cm`.

The output of the `smlnj` contains a lot of information and among it the test cases results as presented and described in Listing 5.28 in section 5.4. The test cases results are not used inside this plug-in, but inside the `rsl.testcases` plug-in where they are displayed inside the *RTest* View. Therefore we create an extension point that allows for a listener to register to it and listen for the output of the test cases execution. This way if any change may affect `rsl.core` plug-in in the future, it will not affect `rsl.testcases` plug-in. More details about how the `rsl.core` exposes the test results to its listeners is presented in section 5.6.

After the test cases execution is implemented, the user actions that trigger this functionality must be made visible and the situation generalized for the cases when the user wants to run the test cases from multiple files.

5.2.4 L^AT_EX generation

The scenario presented in Listing 5.10 states that by clicking the Latex button in the toolbar, or by selecting the Generate Latex context menu or by using Alt+L shortcuts, L^AT_EX documentation is generated comprising the RSL file.

```
scenario "PRJF4"
description "The user can GENERATELATEX for one RSL file. This implies
CONSOLEUPDATEenand PROJECTEXPLORERUPDATE"
```

Listing 5.10: Scenario concerning the latex generation functionality

Thus a new documentation project having the name of the RSL project plus the suffix *Doc* (as mentioned in Listing 5.11) is created. Inside the project a *src* folder is created containing a *main.tex* file as presented in Listing 5.12.

```
scenario "R9"
description "A RAISE project can have associated one documentation project
that has the same name plus the suffix 'DOC'. This project is created when
the first documentation file is created"
```

Listing 5.11: Scenario

```
//add doc project
ProjectExplorer.addProject(latexProject);
//create main.tex file
ProjectExplorer.addFile("main.tex", new Path(latexProjectName+"/src"));
```

Listing 5.12: The creation of the documentation project and \LaTeX main file

When *main.tex* file is created the following \LaTeX lines are being added:

```
\documentclass{article}
\usepackage{listings}
\begin{document}
\lstset{language=rsl}
\lstinputlisting{file_location/rsl_file_name}
\end{document}
```

This string is externalized and added in a file inside *resources* folder with the name of *latex_template.txt*. The newly obtained *main.tex* file can be directly compiled with \LaTeX .

5.3 rsl.editor plug-in

5.3.1 Editor

The RSL editor implementation is started with scenario presented in Listing 5.13.

```
scenario "ED9"
description "RSL keywords and mathematical words are written with a different
colour in the editor"
```

Listing 5.13: Caption from editor scenario

Since the RSL editor is a source code editor, it is developed using the Eclipse JFace Text framework. This framework is used for creating and manipulating text documents [hel13] and is very powerful, flexible and very complex in the same time. JFace Text framework sees the editor as composed of two parts: one that holds the document content, called *document*, and one that takes care of how the content is displayed inside the editor, named *viewer*.

Concretely, in order to create an Eclipse editor, one must use the *org.eclipse.ui.editors* extension point. The configuration for the RSL editor specifies its name, its unique identifier, the files extensions that are understood by the editor and the class that implements the editor functionality. The part of the manifest file that contains this information is captured in Listing 5.14.

```
<extension point="org.eclipse.ui.editors">
  <editor
    name="RSL Editor"
    id="rsleditor.editors.RSLEditor"
    extensions="rsl"
    class="rsleditor.editors.RSLEditor">
  </editor>
</extension>
```

Listing 5.14: Caption from the rsl editor manifest file

The *RSLEditor* class extends the *TextEditor* class from *org.eclipse.ui.editors.text* package. This way the RSL editor gains the basic functionality of the standard text editor for file resources [hel13] and provides the possibility of adding many other features on top. The basic functionality inherited by the RSL editor are:

- Text display
- User possibility to modify text
- Standard text editing operations: cut, copy, paste
- Find and replace operations
- Undo and redo typing operation

This *RSLEditor* class is in charge of creating and instantiating the model and view part of the editor. This is done by the code presented in Listing 5.15.

```
SourceViewerConfiguration viewerCongif = new RSLConfiguration(colorManager);
setSourceViewerConfiguration(viewerCongif);

FileDocumentProvider documentProvider = new RSLDocumentProvider();
setDocumentProvider(documentProvider);
```

Listing 5.15: Class RSLEditor caption

The *RSLDocumentProvider* class's role is to create the memory representation of the file that is stored on the disk. This is done by extending the *FileDocumentProvider* class which implements the *IDocumentProvider* interface and which provides functionality for loading files from disks and keep track of their changes. This is done in the *createDocument* method where a document partitioner is created and attached to the document. This method is presented in Listing 5.16.

```
protected IDocument createDocument(Object element) throws CoreException {

    //create the document
    IDocument document = super.createDocument(element);

    //create a new document partitioner and attach it to the document
    if (document != null) {
        IDocumentPartitioner partitioner =
            new FastPartitioner(
                new RSLPartitionScanner(),
                new String[] {
                    RSLPartitionScanner.RSL_COMMENT
                });
        partitioner.connect(document);
        document.setDocumentPartitioner(partitioner);
    }
}
```

Listing 5.16: createDocument method in class RSLDocumentProvider

The way the document partitioner works is by using a scanner to scan through the document content and return partitions of different content types. Partitions are small parts of the text document having associated a content type, a length and an offset [hel13]. Partitions are disjoint and their reunion form the document content. The idea behind partitions is that the editor must treat different parts of the text differently: e.g., a line of comment is coloured differently than a line of code.

The way the document content is split in content type partitions is based on rules. Eclipse provides some basic rules for creating the partitions through the *IPredicateRule* interface and its standard implementation *PatternRule* class. This mechanism is capable of detecting parts of text that start and end with a pattern. Starting from this rules, there are two kind of content types partitions determined for the RSL code:

- RSL comment
- RSL code

Since Eclipse provides a default partition type, *IDocument.DEFAULT_CONTENT_TYPE*, it is enough to establish the rules for the RSL comment and the

rest of the text (code) is automatically considered a different type, the default type. The partitions types and the rules for detecting the partitions are defined in *RSLPartitionsScanner* class and presented in Listing 5.17. There are two rules defined for identifying a RSL comment partition: one states that every sequence that starts with a `/*` and ends with a `*/` and that can lie on multiple lines is a comment and another one that states that every line starting with the `-` sequence is also a comment.

```
public final static String RSL_COMMENT = "_rsl_comment";

public RSLPartitionScanner(){

    IToken rslComment = new Token(RSL_COMMENT);
    IPredicateRule[] rules = new IPredicateRule[2];

    rules[0] = new MultiLineRule("/*", "*/", rslComment);
    rules[1] = new SingleLineRule("-", "", rslComment);

    setPredicateRules(rules);
}
```

Listing 5.17: Caption from the RSLPartitionScanner class

Once the model part is implemented and configured, the next step is to take care of the UI part of the editor. Eclipse provides a class that handles displaying source code inside an editor. This class is called *SourceViewer* and is a specialization of the *TextViewer* class inside *org.eclipse.jface.text* package. The *RSLConfiguration* extends this class taking care of the syntax highlighting. Therefore it creates an *RSLCodeScanner* that scans through the RSL code searching for the sequences that need to be coloured. The aspects that are coloured in the RSL code are the RSL keywords, the strings and the characters. The complete list of the RSL keywords is presented in Appendix J. Listing 5.18 illustrates how every keyword, string and character are considered tokens that have display information associated.

```
IToken keyword = new Token( new TextAttribute(
    colorManager.getColor(IRSLSyntaxColors.PINK), null, SWT.BOLD));

IToken string = new Token(new TextAttribute(
    colorManager.getColor(IRSLSyntaxColors.GREEN)));

WordRule wordRule= new WordRule(new RSLWordDetector(),other);
for (int i= 0; i < keyWords.length; i++)
    wordRule.addWord(keyWords[i], keyword);

rules.add(wordRule);

//rule for detecting Texts
rules.add(new MultiLineRule("\\"", "\"", string));

//rule for detecting Characters
rules.add(new SingleLineRule("\\'", "'", string));
```

Listing 5.18: Caption from the RSLCodeScanner class

A keyword is coloured in pink and is bold, while a string and a character is green. Please refer to K for a complete list of the colours used for displaying the RSL content, and their RGB value. Then for every keyword in the *keyWord* set variable, a new word rule is created, associating each word to the *keyword* token. The Texts and Characters are both associated to the *string* token, since both are being displayed in the same way. Please notice in Listing 5.18, that the *WordRule* needs a *RSLWordDetector* to determine if a character can be part of the word in the current context.

Another important aspect of an editor is to maintain all the features described so far, while the user is making changes to the document. This part is done using the *IPresentationReconciler* interface which keeps tracks of the document changes by attaching it an instance of the *IPresentationDamager* and *IPresentationRepairer*.

The next scenario that must be implemented is presented in Listing 5.19 and stated that when a user saves a file (either by pressing the Ctrl+S keyboard shortcut, or by selecting File → Save, or File → SaveAs) it is automatically type checked.

```
scenario "EDT5"
description "The user SAVE the current RSL specification and the TYPECHECK is
automatically run."
```

Listing 5.19: Caption from EDITOR scenario

By extending the *TextEditor*, the *RSLEditor* can override the *doSave* and *doSaveAs* methods, that are triggered when a file is saved. But the question here is how to call the typecheck functionality, that is not implemented inside this plug-in, but in *rsl.core*. One solution would be to run it as a separate process and use sockets in order to listen for the output. Another solution would be to create a temporary file where the *doSave* and *doSaveAS* methods can write when they are called, and which the *rsl.core* monitors. However, we consider these two solutions unnecessarily complicated since the two plug-ins *rsl.core* and *rsl.editor* are delivered together. Therefore, we are choosing a simpler approach where we make the typecheck functionality visible inside the *rsl.editor* plug-in, so it can be called from the appropriate methods. We make this decision knowing that *rsl.editor* is not going to be delivered without the *rsl.core* plug-in.

By overriding the *doSave* and *doSaveAs* methods, we need to make sure that the content of the file is first saved and only then it can be typechecked. The situation is presented in Listing 5.20.

```
@Override
public void doSave(IProgressMonitor monitor){
    super.doSave(monitor);
```

```
//get active file in editor ifile
...
TypeChecker tc = new TypeChecker();
tc.typeCheck(ifile);
```

Listing 5.20: Caption from the RSLCodeScanner class

5.3.2 Markers

The next implemented aspect is dealing with the type check errors. The scenario that captures this aspect is the PROBLEMS scenario_chart:

```
scenario "PRB1"
  description "The user can see the problems existing in the workspace with PROBLEMSUPDATE
and EDITORUPDATE."
scenario "PRB2"
  description "For each problem the description, resource, path, location
and type are specified."
```

PROBLEMSUPDATE event states that the Eclipse Problem view must be updated and the EDITORUPDATE states that when an type check error occurs it must also be signaled in the editor.

The connection between the output of the RSL typechecker displayed in the Console view and the Problems view is the *ConsoleToProblems* component. This component must be continuously listening to what is being displayed in the Console and if an error is displayed, the Problems view must be updated.

After some research, it was found that Eclipse provides an extension point that looks for regular expressions in the text console, named *org.eclipse.ui.console.consolepatternmatchlisteners* [hel13]. According to [rsl08], the error output of the RSL typechecker is regular having the form of:

```
file.rsl:line:column: error message
```

Thus for every error discovered in a RSL specification, the name of the file is specified, followed by :, followed by the number of the line in the RSL specification where the error was encountered, followed by :, the column number where the error starts, then again :, a space and the error message. Having regular expressions in the console, a consolePatternMatchListeners extension point is added to the *rsl.editor* plug-in as presented in Listing 5.21.

```
<extension point="org.eclipse.ui.console.consolePatternMatchListeners">
  <consolePatternMatchListener
    class="rsl.editor.problems.ConsoleToProblems"
    id="rsl.editor.consolePatternMatchListener"
    regex="(.*):(\d+):(\d+):(\s*)(.*)">
  </consolePatternMatchListener>
</extension>
```

Listing 5.21: The `consolePatternMatchListener` extension point

The class implementing the *IPatternMatchListenerDelegate* interface and thus, the class notified when a regular expression was found is the *ConsoleToProblems*. The regular expression is composed of five groups:

1. the first group of `(.*)`
Matches any number of characters representing the name of the file
2. the first group of `(\d+)`
Matches one or more digits representing the line number
3. the second group of `(\d+)`
Matches one or more digits representing the column number
4. `(\s*)`
States that any number of space characters can be found in this position
5. the second group of `(.*)` Matches any sequence of characters representing the error message.

When the match is found in the Console, an *PatternMatchEvent* is triggered and the *matchFound* method in the *ConsoleToProblems* class called. Based on the information gathered from the Console output, problems are created as showed in Listing 5.22.

```
//extract error messages from Console
Matcher matcher = ERROR_PATTERN.matcher(lineStr);

if (matcher.matches()) {
    fileName = matcher.group(1);
    line = Integer.parseInt(matcher.group(2));
    column = Integer.parseInt(matcher.group(3));
    err_message = matcher.group(5);
}
else {
    return;}

//create a Problem
Problem prb = new Problem(2, err_message, line, column);
```

Listing 5.22: Caption from *matchFound* method in class *ConsoleToProblems*

Problems must be displayed in the Problems view, in the editor and in the Project Explorer view. To make sure that all three views are consistent and updated at the same time, the concept of **marker** is used. Markers are pieces of information tagged to a resource in order to assure GUI consistency. There are three different types of markers: tasks, bookmarks and problems, and the ones used to represent errors in a file are the **problems** markers. Thus for every error discovered by the typechecker in an RSL specification, a problem marker is added to that specific RSL file. The problem markers are created inside the *ProblemsView* class, by extracting the information from the problems created in Listing 5.22.

```
int lineNumber = problem.getLineNumber();
String message = problem.getMessage();
//map that stores all marker properties
HashMap<String, Integer> map = new HashMap<String, Integer>();

MarkerUtilities.setLineNumber(map, lineNumber);
MarkerUtilities.setMessage(map, message);

map.put(IMarker.SEVERITY, new Integer(IMarker.SEVERITY_ERROR));

//first set values and then create marker
MarkerUtilities.createMarker(file, map, IMarker.PROBLEM);
```

Listing 5.23: Creating the problem markers inside the printProblems method

When creating markers, first their values are being set: line number, message and severity as it is illustrated in Listing 5.23 and only after that the marker is created and attached to a file. We learned from our own experience that the order is very important, since if a marker is first created and then its values are set, the editor does not display the marker at the right location. More about this problem can be read at [Ecl]. After a marker is created and attached to a file, the Problems View, the Editor and the Package explorer will automatically be updated.

5.4 rsl.testcases plug-in

The next task, captured in Listing 5.24, is to display the results of the test cases evaluation in the *RTest* view.

```
scenario "TEST2"
description "Tests results are shown in a separate view involves RTESTUPDATE"
```

Listing 5.24: Caption from TEST scenario

Thus a new *RTest* view is created as a contribution to the *org.eclipse.ui.views* extension point, as presented in Listing 5.25.

```
<extension point="org.eclipse.ui.views">
  <view
    class="rsl.testcases.ui.TestCaseViewPart"
    icon="icons/13RT.png"
    id="rsl.testcases.testview"
    name="RTest"
    restorable="true">
  </view>
</extension>
```

Listing 5.25: RTest view declaration

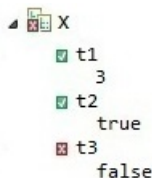


Figure 5.2: Test cases results displayed in the RTest view

The RTest view must display not only the test results, but also the name of the test, if any, and the name of the RSL specification containing the test cases. This information must be displayed in a tree format, where the first level is represented by the RSL files names, the second level comprises the test cases names and the last level is the associated tests results as shown in Figure 5.2. But while working on this part, we realized that it is not mandatory to give names to tests inside the RSL specification, and therefore we decided to discard the third level and show the test results on the second level after their names. If a test does not have a name then only its execution result will be displayed.

To display the test cases in a tree format, an *org.eclipse.jface.viewers.TreeViewer* is created inside the RTest view, as presented in Listing 5.26.

```
viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
TestCaseContentProvider contentProv = new TestCaseContentProvider();
viewer.setContentProvider(contentProv);
contentProv.setViewer(viewer);
viewer.setLabelProvider(new TestCaseLabelProvider());
// Expand level
viewer.setAutoExpandLevel(2);
// Provide the input to the viewer
viewer.setInput(new RSLTestCaseModel());
```

Listing 5.26: Creating a Tree Viewer inside TestCaseViewPart class

The *TreeViewer* needs a content provider to provide the information to be displayed on every level and a label provider to provide the small icons that will be

displayed next to each level. Looking at Listing 5.26, the content provider is the *TestCaseContentProvider* class that implements the *ITreeContentProvider* interface and that provides methods for accessing the elements on the first level of the tree (the files' names) and their children (the test cases' names and results).

The *TestCaseLabelProvider* offers methods for accessing the appropriate icons to display next to the test names depending on the test result.

In the design phase it was considered that a test can be either a success (if the test result is true or any other value that is not false) or false (if the test result value is false). But while implementing the execution of the test cases in section 5.2, we discovered that the tests can also generate run time exceptions when executed with smlnj. For example run time exceptions are generated when zero is raised to a non-positive power, or when a division by zero is encountered. The SML runtime system catches the runtime errors within each test case, and thus they do not disturb the execution of the next test cases. The test result of a test that generated errors, is the associated error message. The complete list of all runtime error message that can be generated by the SML runtime system is presented in ???. Based on these facts, it was decided that a test result value can be in one of the three states:

- success
The test value is true or any other value that is not false.
- false
The test value is false.
- error
The test has generated a runtime error.

If a test is a success then a green icon is displayed next to it, if it is false then a blue icon is displayed and in an error case, a red icon will mark the failure. An icon is also displayed next to the RSL file name, based on the results of the test cases defined inside the RSL file. If one or more tests generated a runtime error, then a red icon is displayed next to the file name. If there are no error tests, but there is at least one false test, then the icon next to the file name is blue. If there are no error or false tests then all tests have a green icon next to them and so is the file name.

The last line in Listing 5.26 sets the input of the tree viewer as a new *RSLTestCaseModel*. The *RSLTestCaseModel* class represents the tree structure on two levels as a list of lists. The first list contains the name of the RSL files and their associated lists of test cases and test results.

The next aspect is to populate the model and implicitly the view with the tests results. And this must be done every time the user chooses to run the test cases. It can be that the user chooses to run the tests cases of the currently opened file in the editor, or to run all tests cases in the workspace.

In order to listen for the test cases results, we just need to add an extension to the *testcaseslisteners* extension point defined in the *rsl.typechecker* plug-in in section 5.2. The new extension declaration is presented in Listing 5.44.

```
<extension point="rsl.typechecker.testcaseslisteners">
  <listener
    class="rsl.testcases.model.TestCasesListener">
  </listener>
</extension>
```

Listing 5.27: testcases extension to rsl.typechecker.testcaseslisteners extension point

The listener class is *TestCasesListener*, which must implement the *IRSLTestCasesListener* interface as specified in the *rsl.typechecker.testcaseslisteners* extension point contract. Thus after each execution of a RSL specification, the *finish* method inside the *TestCasesListener* is called with the output of the SML run. Thus not only the tests results are sent to the listener, but the entire SML run output. An example of an SML run output is further presented in Listing 5.28

```
1 Standard ML of New Jersey v110.59 [built: Mon Jun 05 13:26:49 2006]
2 [opening D:/RAISE/src/TestSML/X.sml]
3 [autoloading]
4 [library $smlnj/cm/cm.cm is stable]
5 [library $smlnj/internal/cm-sig-lib.cm is stable]
6 [library $/pgraph.cm is stable]
7 [library $smlnj/internal/srcpath-lib.cm is stable]
8 [library $SMLNJ-BASIS/basis.cm is stable]
9 [autoloading done]
10 val it = () : unit
11 val it = true : bool
12 [autoloading]
13 [autoloading done]
14 val it = () : unit
15 val it = () : unit
16 [autoloading]
17 [autoloading done]
18 val it = () : unit
19 val it = true : bool
20 [autoloading]
21 [autoloading done]
22 [opening D:/RAISE/src/TestSML/X_.sml]
23 structure RT_Int : <sig>
24 structure RT_Bool : <sig>
25 structure X : <sig>
26 open X
27 val it = () : unit
28 val it = () : unit
29 val it = () : unit
```

```

30 val it = () : unit
31 [t1] 3
32 val it = () : unit
33 [t2] true
34 val it = () : unit
35 val it = () : unit
36 val it = () : unit
37 val it = () : unit
38 -

```

Listing 5.28: An example of SML run output

The output message is filled with additional information that is not of interest if we only want to display the test cases and their results. This is why we need to filter this message and search for the lines that present the test names and their execution result. The first part of the output message (in the previous example from line 1 to 21) contains messages about libraries being loaded. Then the SML file is being opened in line 22 [**opening D:/RAISE/src/TestSML/X_ .sml**] and the structures loaded (lines 23, 24 and 25). The part that interests us is after the **open X** line (line 26), since **X** is the RSL specification containing the test cases. Based on the documentation at [rsl08] and on the rsltc tools source code, the tests results are the lines framed by two **val it = () : unit** lines and that do not contain any of the following strings:

- Unexecuted expressions in
- Complete expression coverage of
- error(s)

Based on the rules described so far, the lines 31 and 33 in Listing 5.28 contain test cases results. If the test case has a name then it must be between the straight brackets [and]. What comes after the brackets until the end of the line is the test case result.

As mentioned before, the entire output of the SML run is sent to the *testsFinished* in the *TestCasesListener* class. Therefore, this method is the one filtering the received message based on the rules previously described, as captured in Listing 5.29.

```

line1 = r.readLine(); //read the line with "use X"
line2 = r.readLine();
while ( line1 != null && line2 != null && (line3=r.readLine()) != null ) {
    //rules defined based on rsltc.el source code and the documentation at
    //ftp://ftp.iist.unu.edu/pub/RAISE/dist/user_guide/ug.pdf
    if( line1.contains("val it = () : unit") && !line2.contains("val it = () : unit")
        && !line2.contains("Unexecuted expressions in")
        && !line2.contains("Complete expression coverage of")
        && !line2.contains("error(s)")

```

```

        && line3.contains("val it = () : unit") )
    {
        ...
    }
    line1 = line2;
    line2 = line3;
}

```

Listing 5.29: Java code filtering the SML output

Once a test name and result are identified a new test case is created and added in the test cases list of the RSL file, in order to be displayed in the RTest view. Listing 5.30 presents captures the Java code that accomplishes this.

```

TestCase tc = new TestCase(testName, value);
testfile.getTestCases().add(tc);
...
content.refreshView();

```

Listing 5.30: Java code filtering the SML output

5.5 rsl.wizard plug-in

Since a plug-in storing only the perspective is a very small plug-in, we decided to add here also the implementation of the New RSL Project Wizard.

5.5.1 RSL perspective

The requirement presented in Listing 5.31, states that all RSL related actions and views must be grouped under a single perspective named RSL.

```

scenario "R6"
  description "All RSL actions and views are stores inside the RSL perspective"

```

Listing 5.31: Perspective requirement

Thus, we use the *org.eclipse.ui.perspectives* extension point to declare the RSL perspective as presented in Listing 5.32.

```

<extension point="org.eclipse.ui.perspectives">
  <perspective
    class="rsl.perspective.RSLPerspectiveFactory"
    id="rsl.perspective.rslperspective"
    name="RSL">
  </perspective>
</extension>

```

Listing 5.32: RSL perspective extension

The name of the perspective is set to *RSL* and the class describing the initial layout is the *RSLPerspectiveFactory* class. Inside the *createInitialLayout* method, the views positions and dimensions are established. Inside an Eclipse workbench page, the central component is considered the editor and all the other views are arranged around it.

```
public void createInitialLayout(IPageLayout layout) {
    //get editor area
    String editorArea = layout.getEditorArea();
    //add the Explorer view on the left
    layout.addView(IPageLayout.ID_PROJECT_EXPLORER, IPageLayout.LEFT, 0.20f,
        editorArea);
    //add Consoles and Problems views under the editor
    IFolderLayout bottom = layout.createFolder("bottom", IPageLayout.BOTTOM, 0.66f,
        editorArea);
    bottom.addView(IConsoleConstants.ID_CONSOLE_VIEW);
    bottom.addView(IPageLayout.ID_PROBLEM_VIEW);
    //add RTest view under the Project Explorer
    layout.addView(ID_RTEST_VIEW, IPageLayout.BOTTOM, 0.5f,
        IPageLayout.ID_PROJECT_EXPLORER);
}
```

Listing 5.33: RSL perspective layout initialization

Listing 5.33 describes the arrangement of the views around the RSL editor and their initial sizes. The user can modify their arrangement and sizes, but every time the user chooses to restore the RSL perspective the *createInitialLayout* method is called and the elements' positions and sizes are restored.

5.5.2 New RSL Project wizard

In order to fulfill the requirements presented in Listing 5.34, a new project wizard must be created for the RSL project. The completion of the wizard will result in a new RSL project that contains a folder named *src*.

```
scenario "R1"
description "The user must be able to create a new RSL project"
scenario "R3"
description "When a new RAISE project is created, it contains a single
folder named 'src'"
```

Listing 5.34: Caption from requirements scenario

To create the new project wizard, we contribute to the *org.eclipse.ui.newWizard* extension point as illustrated in Listing 5.35.

```

<extension point="org.eclipse.ui.newWizards">
  <category
    id="rsl.perspective.category.RSLcategory"
    name="RSL">
  </category>
  <wizard
    category="rsl.perspective.category.RSLcategory"
    class="rsl.perspective.newwizard.RSLProject"
    finalPerspective="rsl.perspective.rslperspective"
    id="rsl.perspective.wizard.NewRSL"
    name="RSL Project"
    project="true">
  </wizard>
</extension>

```

Listing 5.35: RSL project new wizard extension

First, a category named RSL is defined to group all RSL creation wizards. Currently, there is only one creation wizard, defined for the RSL project, but maybe in the future we will want to support user in the creation of other elements like RSL scheme, test cases and so on, and it is easier for the user to identify them if they are grouped. The class creating the new RSL project wizard is *RSLProject* and the RSL project is associated with the RSL perspective created in subsection 5.5.1.

Since all the information needed to create an RSL project is its name and location, a wizard with one page is enough to collect all the information. The page is added in the *addPages* method of the *RSLProject* class presented in Listing 5.36, while the layout of the page is created in *createControl* method of the *RSLProjectCreationPage* class.

```

public void addPages(){
    //Sets the wizard title
    setWindowTitle("New RSL project");
    //create a new wizard page
    String title = "Create a RSL project";
    String description = "Enter a project name";
    page1 = new RSLProjectCreationPage(title, description);
    //add it to the wizard
    addPage(page1);
    //populate wizard page if the user has already selected smth in the workbench
    page1.init(intialSelection);
}

```

Listing 5.36: Adding the page to the New RSL Project Wizard

The important aspect when expecting user input is to validate it and to not allow the user perform the *Finish* action if the RSL project can not be created. A project name and location is not considered acceptable if any of the following situations is encountered:

- the project name is empty

- the project location is empty
- a project with the same name and location already exists
- the project name contains any of the illegal characters:
`{/, \n, \r, \t, \0, \f, ', ?, *, \, <, >, |, ", : }`
- the project location contains any of the illegal characters:
`{ \r, \t, \0, \f, ', ?, *, <, >, |, " }`

The checking of the user input is done in the `updatePageComplete` method inside the `RSLProjectCreationPage` class, and if any of the illegal situations is encountered, the `setPageComplete(false)` is called disabling the activation of the wizard's *Finish* button. When the user input is not considered correct, a message is displayed in the wizard page to inform the user about the problem. For example Listing 5.37 captures the Java code verifying if a project with the same name and location already exists.

```
String fullProjectPath = destinationFolder+"/"+projectName;
File newPrj = new File(fullProjectPath);
if(newPrj.exists()){
    setMessage(null);
    setErrorMessage("The project already exists");
    return;
}
```

Listing 5.37: Verifying the user input in updatePageComplete method

If the user input does not violate the rules described earlier, the wizards' *Finish* button is activated and the user can click on it. As a result a new RSL project with name and location supplied by the user must be created.

The creation of a project is a long running operation and thus it must run in another thread in order not to block the GUI and implicitly the user activity. To accomplish this a new progress monitor is created to monitor the `createRSLProject` execution Listing 5.38.

```
new IRunnableWithProgress(){
    @Override
    public void run(IPressMonitor monitor) throws InvocationTargetException,
        InterruptedException {
        //create the RSL project with the user information
        createRSLProject(userInput, monitor);
    }
}
```

Listing 5.38: Running the project creation in a new thread

The `createRSLProject` method creates the new RSL project with its associated structure, based on the user input and informing the monitor about the progress.

5.6 Extension points

In order to allow other contributors to easily extend eRAISE, we have developed extension points. By using extension points our system does not need to know anything about the other plug-ins, just the fact that somebody has extended it. These extension points have also been used between the four plug-in that compose eRAISE, in order to achieve loose coupling between components and to make the code easier to read and understand. By using the extension points internally we also provide examples for other on how to use the provided extension points.

The *rsl.core* plug-in defines an extension point for broadcasting the results of the RSL test cases to other plug-ins that need this information. The extension point name is *testcaseslisteners* and is defined inside the *plugin.xml* file as presented in Listing 5.39.

```
<extension-point id="testcaseslisteners" name="testcaseslisteners"
schema="schema/testcaseslisteners.exsd"/>
```

Listing 5.39: Declaring the testcaseslisteners extension point

The schema defining the contract of the extension point contains an element named *listener* that has a single attribute of type class Listing 5.40. This attribute specifies that all extensions contributing to the *testcaseslisteners* extension point, must implement the *IRSLTestCaseListener* interface.

```
<element name="listener">
  <complexType>
    <attribute name="class" type="string">
      <annotation>
        <appinfo>
          <meta.attribute kind="java" basedOn=":rsl.typechecker.core.
            IRSLTestCasesListener"/>
        </appinfo>
      </annotation>
    </attribute>
  </complexType>
</element>
```

Listing 5.40: Declaring the testcaseslisteners extension point

The *IRSLTestCaseListener* interface is illustrated in Listing 5.41. It has two methods *testsStarted* that is called whenever the execution of test cases is started and *testFinished(String message, IFile rslfile)* that informs a listener that the tests have finished executing by sending the name of the RSL file that contains the test cases and the output of the smlnj execution.

```
public interface IRSLTestCasesListener {
    public void testFinished(String message, IFile rslfile);
}
```

```

    public void testStarted();
}

```

Listing 5.41: IRSLTestCase interface

In order for the extensions to be notified about the test cases execution, the *rsl.core* plug-in needs to find them first and then call the appropriate methods on them. The listeners of the test cases executions are loaded as captured in Listing 5.42, the first time the test cases are executed. The reason why they are not loaded when the plug-in starts is because we need to conform to the *Lazy Loading Rule* that states that "contributions are loaded only when needed" [Gam04]. The listeners are identified by getting the platform extension registries and collecting all extensions to the *rsl.core.testcaseslisteners* extension point.

```

IExtensionRegistry registry = Platform.getExtensionRegistry();
IExtensionPoint extensionPoint = registry.getExtensionPoint(LISTENER_ID);
Extension[] extensions = extensionPoint.getExtensions();
ArrayList<IRSLTestCasesListener> list = new ArrayList<IRSLTestCasesListener>();
//for each extension to our extension point
for(int index = 0; index < extensions.length; index++){
    IConfigurationElement[] elements = extensions[index].getConfigurationElements();
    //for each configuration element of a extension
    for(int j = 0; j < elements.length; j++){
        try {
            Object listener = elements[j].createExecutableExtension("class");
            if(listener instanceof IRSLTestCasesListener)
                list.add((IRSLTestCasesListener)listener);
        }catch (CoreException e){
            e.printStackTrace();
        }
    }
}

```

Listing 5.42: Loading extensions

When the test cases are executed, the *rsl.core* plug-in notifies the listeners by calling their *testsStarted* method. Since extensions may generate errors inside their *testsStarted* method, we need to protect out plug-in from undesired behaviour. Thus we will wrap the method calls inside the *ISafeRunnable* interface making the platform taking care of the any exception as presented in Listing 5.43. By protecting the plug-in from malicious extensions we conformed with the *Good Fence Rule* that states: "when passing control outside of your code, protect yourself" [Gam04].

```

//for each listener
while(it.hasNext()){
    final IRSLTestCasesListener listener = (IRSLTestCasesListener) it.next();
    ISafeRunnable runnable = new ISafeRunnable() {
        @Override
        public void run() throws Exception {
            listener.testsStarted();
        }
        @Override
        public void handleException(Throwable exception) {

```

```

        //TODO what happens if an listener throws exception?
    } };
    SafeRunner.run(runnable);
}

```

Listing 5.43: Notifying extensions

In exactly the same way, *rsl.core* defines another extension point for listening to the typechecker output. The name of the extension point is *typechecklisteners* and a plug-in that wants to be informed about the output of the typechecking can provide an extension to it. The interface that must be implemented is *ITypeCheckListener*.

The *rsl.testcases* plug-in extends the *rsl.core* plug-in by creating an extension to the *rsl.core.testcaseslisteners* extension point. The test cases listener defined in *rsl.testcases* plug-in is presented in Listing 5.44.

```

<extension point="rsl.typechecker.testcaseslisteners">
  <listener
    class="rsl.testcases.model.TestCasesListener">
  </listener>
</extension>

```

Listing 5.44: testcases extension to rsl.typechecker.testcaseslisteners extension point

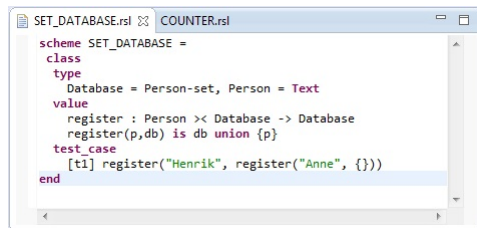
The listener class is *TestCasesListener*, which must implement the *IRSLTestCasesListener* interface as specified in the *rsl.typechecker.testcaseslisteners* extension point contract. Thus after each execution of an RSL specification, the *finish* method inside the *TestCasesListener* is called with the output of the SML run.

This chapter describes the methods that have been used to test eRAISE during the implementation phase. The testing phase relies on a combination of manual and automated testing, each of them further described in a separate section.

6.1 Manual testing

The easiest method for testing a GUI plug-in is to act as the plug-in user and manually click or select GUI items and verify that the expected events are taking place. This way the system is seen as a black box and the focus is on the user actions inside the GUI and the plug-in responses to the user. Manual testing is not a very rigorous method for testing since it is not very fast, it relies on the tester observation skills and most of the time it does not involve complicated scenarios. But manual testing is good for testing the main functionality or pieces of functionality without having to write new code.

In order to test the plug-in without deploying it, the plug-in must be launched as a separate Eclipse application. This way a new runtime workbench is opened containing the plug-ins that are currently being under development. Opening another workbench can take a few seconds and thus the manual testing of an Eclipse plug-in can be quite time consuming.



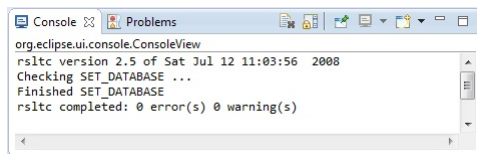
```

scheme SET_DATABASE =
class
type
  Database = Person-set, Person = Text
value
  register : Person <> Database -> Database
  register(p,db) is db union {p}
test_case
  [t1] register("Henrik", register("Anne", {}))
end

```

Figure 6.1: RSL specification opened in the RSL editor

eRAISE was continuously tested during its implementation phase using manual testing. After a new piece of functionality was written, a separate Eclipse application was launched and the new functionality tested. One example is testing that by pressing the **TC** button in the toolbar, the currently opened file in the RSL editor is being type checked. Thus, one file is opened in the RSL editor as presented in Figure 6.1 and the **TC** button is pressed.



```

org.eclipse.ui.console.ConsoleView
rsrtc version 2.5 of Sat Jul 12 11:03:56 2008
Checking SET_DATABASE ...
Finished SET_DATABASE
rsrtc completed: 0 error(s) 0 warning(s)

```

Figure 6.2: Type checker output displayed in the Console View

As a result to this, the Console View gets updated with the message presented in Figure 6.2. The message states that a file with the same name as the one opened in the editor has been type checked. After testing the same thing on different RSL files opened in the editor, it can be concluded that by pressing the **TC** toolbar button, the current active file is being typechecked.

The same way of testing has been applied for all possible user actions, verifying that the plug-in is doing what it was supposed to do.

6.1.1 Input validation

An important aspect when testing is to verify what happens if the input of an action is not the expected one. For example what happens if the typechecking is called on a file that does not contain RSL specifications e.g., an XML file? Or how does the system react when a user tries to create a new project using a name that already exists in the workspace and so on.

Some of the input problems can be avoided by respecting the **Relevance Rule** concerning Eclipse plug-ins development. This rule states: "Contribute only when you can successfully operate" [Gam04]. This means that the availability of a contribution must be limited to the cases where it can be used. Thus, if a file does not contain RSL specifications then the user should not have the option to typecheck it, to SML translate it and so on. This limitation of availability has been accomplished by using the **visibleWhen** construction on GUI elements declaration. For example the RSL buttons in the toolbar are applied on the currently active file in the RSL editor. Therefore the buttons are only visible when the RSL editor is active. And the **visibleWhen** construction used to represent that, is captured in Listing 6.1. It basically states that the buttons should be visible only when the editor with the id **RSLEditor.editor.rsleditor** is active.

```
<visibleWhen checkEnabled="false">
  <with variable="activeEditorId">
    <equals
      value="RSLEditor.editor.rsleditor">
    </equals>
  </with>
</visibleWhen>
```

Listing 6.1: visibleWhen construction used for the toolbar buttons

In what concerns the validation of the user input introduced when creating a new RSL project, this is programatically done inside the **NewProjectWizard** plug-in. When a user enters a name and a location for the project, the input is automatically tested to see if one of the following situations is encountered:

- The project name contains illegal characters
The list of illegal characters for a project name is considered to be:
{/, \n, \r, \t, \0, \f, ', ?, *, \, <, >, |, ", : }
- The project path contains illegal characters The list of illegal characters for a project location is considered to be:
{ \r, \t, \0, \f, ', ?, *, <, >, |, " }
- A project with the same name and path already exists

If any of the three situations is encountered, the wizard signals an error and its **Finish** button is deactivated. Thus the user can not create the project. The user must either try a different input or can choose to cancel the entire operation. Figure 6.3 captures the situation when a user tries to create a project with a name that already exists in the workspace. Please notice that the **Finish** button is not active.

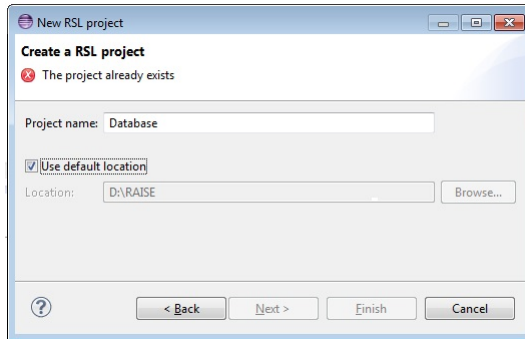


Figure 6.3: New project creation wizard signaling an error

6.2 Automated testing

Besides manual testing, eRAISE has also been tested using automated tests in the form of unit tests. Since eRAISE is written in Java, the framework used to create and run the tests is the JUnit framework [JUN]. The following subsection describes and gives examples of how the tests were created.

6.2.1 JUnit testing

Ideally, for each method inside the source code a series of JUnit tests should be written to verify that the method does what it was intended to do. In what concerns eRAISE, JUnit tests have been written only for the most important pieces of functionality, with the scope of providing a more structured way of testing than the manual one.

For example one important functionality is considered the creation of a new RSL project. Thus the tests questions what happens after the user has entered the data in the *New RSL Project* wizard and clicks the *Finish* button. Concretely, the JUnit test verifies that:

- A new project has been created
- The new project contains a **.project** file
- The new project contains a **src** folder.

The implementation of the JUnit test is shown in Listing 6.2, where the *createRSLProject* method is called to create a new project with the name of *testProject* in the *D:/RAISE* directory.

```

@Test
public void testProjectCreation(){
    String projectName = "testProject";
    String projectPath = "D:/RAISE";
    String[] projectArgs = {projectName, projectPath};

    //call the project creation method
    IProject newProject = RSLProject.createRSLProject(projectArgs);

    //test that the project was created
    Assert.assertNotNull(newProject);

    //test that the project contains .project file
    File projectFile = new File("D:/RAISE/testProject/.project");
    if( !projectFile.exists() )
        Assert.fail();

    //test that the project contains the src folder
    File srcFolder = new File("D:/RAISE/testProject/src");
    if( !srcFolder.exists() )
        Assert.fail();

    newProject.delete(true, null);
}

```

Listing 6.2: JUnit test for the project creation

After the *createRSLProject* method is called, it is verified that the project was created (*Assert.assertNotNull(newProject)*), that it contains the *.project* file (*projectFile.exists()*) and that the project contains the *src* folder (*srcFolder.exists()*). In order for the test to be passed, all the three conditions must be satisfied.

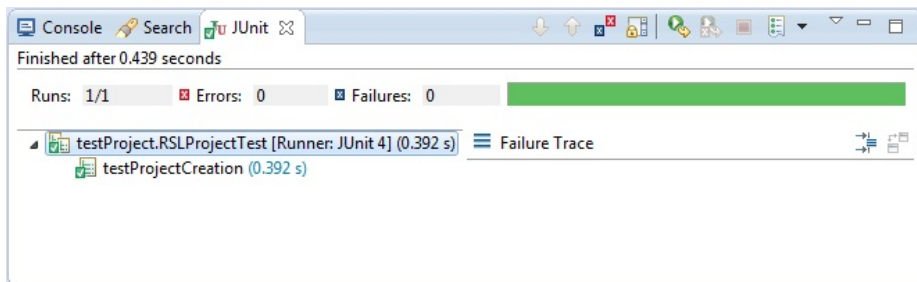


Figure 6.4: JUnit window showing the test result

By executing the test as a JUnit Plug-in Test, a new runtime workbench is opened, the test is executed and the runtime workbench is closed. The *testProjectCreation* test is passed and the JUnit window illustrating this is captured in

Figure 6.4.

Tests similar to the one presented in Listing 6.2 were written to verify the creation of a new project after the SML translation of a RSL file or after the L^AT_EX generation. For example after the SML translation of a RSL specification, a SML project must exist satisfying the following conditions:

- Its name is composed of the RSL project name plus the suffix *SML*.
- It contains the *.project* file.
- It contains a folder with the name *src*.
- It contains a SML file with the same name as the RSL specification and that has the same location inside the SML project, as the RSL file has inside the RSL project.
- It contains a SML file with the same name as the RSL file plus the suffix *_*, and that has the same location inside the SML project, as the RSL file has inside the RSL project.

User guide

This chapter is a user guide for the eRAISE plug-in. It describes the steps that need to be taken inside eRAISE in order to typecheck an RSL specification, translate it to SML, execute its test cases and generate L^AT_EX documentation for it. It also presents the output format and the GUI elements involved in each action, with screenshots for better understanding. The installation guide will be later provided in the eRAISE repository on <https://github.com/kiniry/eRAISE>.

7.1 Writing RSL specification

In order to write a RSL specification the next steps can be followed:

1. Create a new RSL Project
2. Create a new RSL file
3. Edit the RSL file

Each of these steps is further described in a separate subsection.

7.1.1 Create a new RSL project

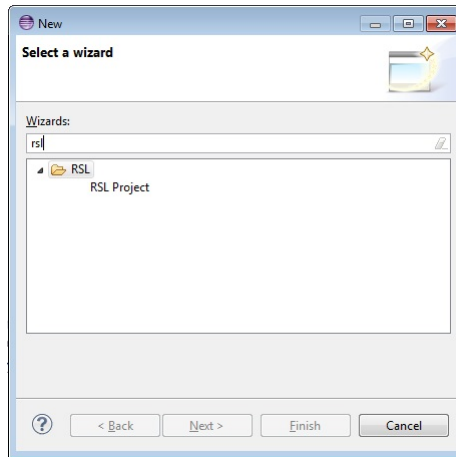


Figure 7.1: New wizard

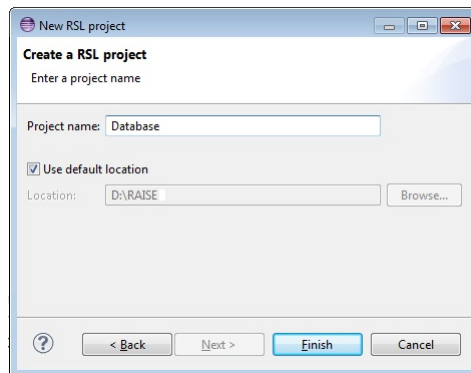


Figure 7.2: New RSL Project

Go to **File** → **New** and click on **Other** (or just press the keyboard shortcut **Ctrl+N**). A new window appears asking to select a wizard. Type in **rsl**, like in Figure 7.1 and select **RSL Project**.

A new window appears and the name of the new project and its location must be filled. Fill in the project name as **Database** and leave the location as default, just like in Figure 7.2 and then press **Finish**.

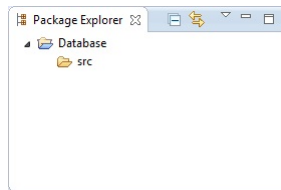


Figure 7.3: The Package Explorer window displaying the Database project

A new RSL project was created and it can be seen in the **Project Explorer** window in the left. Double click on the **Database** folder to see its internal structure. In this moment it only has a subfolder named **src** Figure 7.3.

7.1.2 Create a new RSL file

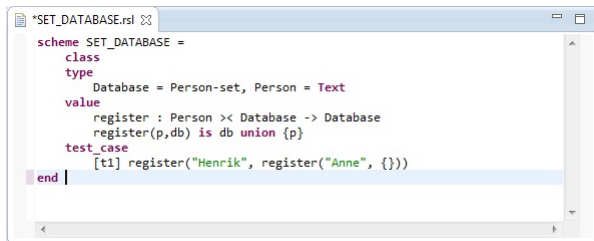
Right click on the **src** folder that was created in the previous step and select **New** → **File**. A new window opens asking for the name of the new file. Write **SET_DATABASE.rsl** and click **Finish**. As a result a new file is created under the **src** folder and opened inside the RSL editor.

7.1.3 Edit the RSL file

Having the **SET_DATABASE.rsl** file opened inside the RSL editor, add the following lines:

```
scheme SET_DATABASE =
  class
    type
      Database = Person-set, Person = Text
    value
      register : Person >< Database -> Database
      register(p,db) is db union {p}
    test_case
      [t1] register("Henrik", register("Anne", {}))
  end
```

The text should look like the one in Figure 7.4. The * symbol before the file name inside the editor means that the file changes has not been saved. Pressing



```

scheme SET_DATABASE =
  class
  type
    Database = Person-set, Person = Text
  value
    register : Person x< Database -> Database
    register(p,db) is db union {p}
  test_case
    [t1] register("Henrik", register("Anne", {}))
end

```

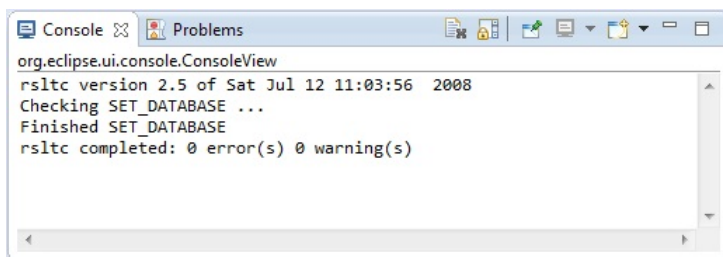
Figure 7.4: The SET_DATABASE.rsl file opened in the RSL editor

Ctrl+S will save the file and automatically trigger the type checker. More about the type checking is presented in section 7.2.

7.2 Type check a RSL specification

The syntax and type checking of a RSL specification can be triggered in many ways:

- Every time a RSL specification is saved it is automatically typechecked.
- Pressing the **TC** button in the toolbar menu, triggers the typechecking of the currently active RSL specification in the editor.
- Right clicking inside the RSL editor or on the file inside the Package Explorer will display the **Type check** context menu item. Clicking it will also trigger a type check on the selected file.



```

org.eclipse.ui.console.ConsoleView
rsltc version 2.5 of Sat Jul 12 11:03:56 2008
Checking SET_DATABASE ...
Finished SET_DATABASE
rsltc completed: 0 error(s) 0 warning(s)

```

Figure 7.5: Type check output displayed in Console window

The result of the type checking is displayed in the **Console** window e.g., by type checking the SET_DATABASE.rsl file, the output presented in Figure 7.5 is displayed.

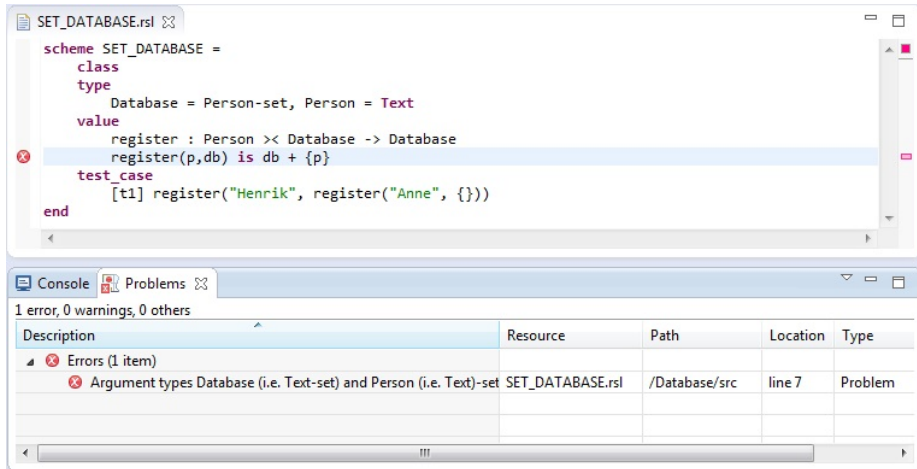


Figure 7.6: Errors displayed in the Problems view and in the RSL editor

If the RSL specification is not correct, then the type checker output contains error messages in the **File:line:column:error message** format. For example replacing the **union** in line 7 in SET_DATABASE.rsl with the mathematical symbol $+$ and saving the file displays the following output in the Console:

```

rsltc version 2.5 of Sat Jul 12 11:03:56 2008
Checking SET_DATABASE ...
D:/eRAISE/src/eRAISE/runtime-EclipseApplication/Database/src/
SET_DATABASE.rsl:7:26: Argument types Database (i.e. Text-set)
and Person (i.e. Text)-set incompatible with '+' type
Int << Int -> Int or
Real << Real -> Real or
Int -> Int or
Real -> Real
Finished SET_DATABASE
rsltc completed: 1 error(s) 0 warning(s)

```

The error is also displayed in the Problems view and the editor is updated to show the RSL line that generated the error. Figure 7.6 illustrates how the Problems and editor are changed after type checking an erroneous specification. The error in the problems view shows the error message, the file name that was type checked, the path to that file relative to the workspace, the line number of the content that generated the error and the type of the problem. By double clicking the error in the Problems view the cursor is moved inside the editor to the RSL line that generated the error.

Replacing `+` back with **union** in `SET_DATABASE.rsl` and saving the file will remove all the editor markers and errors in the Problems view, since the RSL specification is now correct.

7.3 Translate RSL specification to SML

An RSL specification can be translated to SML by using the file's **Translate to SML** context menu. This context menu can be seen by right clicking inside the RSL editor or on the file inside the **Package Explorer** view. As a result of the SML translation, two new SML files are being created. One has the same name as the RSL specification and the other has the same name plus the suffix `_`. These two files are stored in an SML project that has the same name as the project containing the RSL specification plus the suffix **SML**.

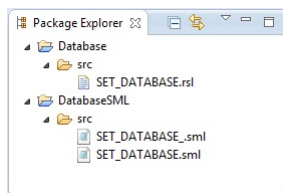


Figure 7.7: Database project and its corresponding SML project

Figure 7.7 captures the new SML project and the two SML files that were created as a result of translating the `SET_DATABASE.rsl` file to SML.

If the RSL specification file has another path inside the RSL project e.g., `files/src`, then the SML files will have the exactly same path, `files/src`, inside the SML project.

Before trying to translate the file to SML, the SML translator calls the type checker, so if the RSL specification contains errors no SML files are created.

7.4 Run test cases

To run the test cases from an RSL specification, one can press the **RT** button in the toolbar menu. Another option is to select the **Run test cases** context menu item after right clicking inside the RSL editor or inside the **Package explorer** view.

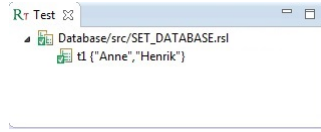


Figure 7.8: The test case result of the SET_DATABASE.rsl specification

By executing the test cases, the tests results are shown in the **RTest** view as in Figure 7.8. The first line is the name of the file, while the test results are displayed as children of the file. For each test case, the name of the test and the result are displayed on the same line separated by a space. If the test has no name, then only its result is displayed.

A test can be in one of the three states:

- success
if a test value is true or any other value that is not false.
- false
if the test value is false.
- error
if the test has generated run time error e.g., division by 0 generates a runtime error

Based on the test result status, an icon is displayed next to its name and result. If the test is false then the icon is blue, if an error was encountered while running the test, then the icon is red, otherwise it is green to show success. The file name has also an icon associated inside the **RTest** view. If the file contains one or more erroneous tests then the icon displayed next to the file name is red. If there are no run time errors, but there are some false tests, the icon is blue. And if all tests are successful then a green icon is displayed in the **RTest** view next to the file name. In Figure 7.8 there is only one test, named **t1** whose value is **{"Anne","Henrik"}**. Since this caused no run time error and it is not **false**, the test is considered a success and thus a green icon is displayed next to it. Since the **SET_DATABASE** specification has only one test and it is successful, then it is also considered a success and a green icon is displayed next to its name.

If the RSL specification has no test cases, but the user has chosen any way to try to execute the file, then no output is displayed in the **RTest** view.

7.5 Generate Latex document

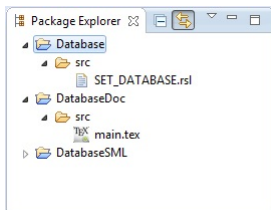


Figure 7.9: The doc project associated to the Database project

The RSL specification can be included in a \LaTeX document by pressing the **Latex** button from the toolbar menu or by choosing the **Generate Latex** item from the editor context menu or from the **Package Explorer**. As a result of this action a new project is created with the same name as the RSL project plus the suffix **Doc**. Inside the newly created project, under the **src** folder, the user can see a **main.tex** file which contains the \LaTeX text for including the RSL specification. Figure 7.9 captures the structure of the project that was created after the **Latex** button was pressed with **SET_DATABASE.rsl** being the active file. The **main.tex** file can be directed compiled with \LaTeX and if no other text is added, the only thing the document will contain is the RSL specification.

7.6 Actions on more than one file

So far in this chapter, all the actions were concerned with only one RSL file: the type check of one file, the SML translation of one file, running the test cases from one file and generating \LaTeX for one RSL file. But sometimes, when projects contain a large number of RSL files, it is nice to be able to execute action on multiple files without having to click on each of them separately. The following subsections describe how to do so.

7.6.1 RSL menu

The **RSL menu** in the main menubar offers the possibility to take actions on all RSL files inside the workspace. The RSL menu has four submenu items presented in Figure 7.10 and further explained:

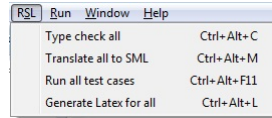


Figure 7.10: RSL menu

- **Type check all**
By clicking this menu item, all RSL files in the workspace are type checked. All type check messages are displayed in the Console, in the order in which the RSL files were type checked.
- **Translate all to SML**
This menu generates the associated SML files for all RSL specifications in the workspace. Once with the SML files creation, the associated SML projects are also created for each RSL project. Informative messages are displayed in the Console for all translations.
- **Run all test cases**
This option executes all RSL test cases available in the workspace. All RSL specifications containing test cases, along with the test cases results are displayed in the **Test** window.
- **Generate Latex for all**
By choosing this menu item, all RSL specifications in the workbench are included in \LaTeX documents. For each RSL project, an associated doc project is created containing one **main.tex** file that includes the RSL specifications in the RSL project.

7.6.2 Context menus on multiple files

The user can apply actions on multiple resources at once, by using the context menus available in the **Project Explorer** window.

By right clicking on a project in the **Project Explorer** window, the user can select to type check, translate to SML, run test cases or generate \LaTeX for all RSL specifications available inside that project.

The user can also select multiple projects or multiple RSL files (by holding the Ctrl key pressed) and apply the four actions on the selection. If the selection comprises e.g., RSL files and one file that is not RSL, then the four context menus will not be visible. The context menus are available only for one or more

projects **or** one or more RSL files are selected. For any other type of resource the RSL context menus are not available.

Future work

This chapter proposes ideas on how eRAISE can be extended and improved. An example is discussed, analyzed and designed using the same methodology that was used for designing and analyzing eRAISE.

8.1 Future work

There are many ways in which eRAISE can be extended, and in this section we will suggest some potentially new tools and features. These new tools can be added as new plug-ins extending the eRAISE extension points. While some new tools can refer to new RAISE functionality, others can just enrich user experience inside Eclipse. If the developers of the new tools need to modify eRAISE, they can do it by downloading its source code from <https://github.com/kiniry/eRAISE> on GitHub.

One possibility is to extend eRAISE with a new series of plug-ins comprising the `rsrtc` tools that were not included in eRAISE.

- Confidence condition generation.

- Display of module dependencies possibly with an interactive display.
- Generation of VCG file to show module dependencies.
- Translation to C++.
- Translation to PVS.
- Translation to SAL.

Another possibility would be to enrich the RSL editor with new features like:

- Mathematical characters inside the editor
This way the user will not have to write the ASCII characters, but use directly the RSL symbols e.g., \rightarrow instead of `->`.
- Type aware completion
This will help users with suggestions while typing inside the RSL editor
- Code folding
Offers the possibility of hiding and showing fragments of code displayed inside the editor.

There are also many Eclipse features that can be added to eRAISE in order to enhance user experience. For example when designing and analyzing eRAISE we also thought that it would be nice to have in the future an Outline view for the RSL modules and a REPL (Read-Evaluate-Print Loop) view, and thus we also included these two in the eRAISE design. Their EBON specification can be found along with the eRAISE specification, but it is marked as future work. However their complexity is not negligible and contributors may need to investigate RSL syntax in detail.

8.1.1 Example

eRAISE was developed using a methodology based on BON. Thus a possibility when developing eRAISE extensions would be to start from the EBON expressed design and build on top of it using the same methodology.

Let us take an example from the ones mentioned above and exemplify how it can be specified using EBON. Let us assume a contribution that proves RSL theories by translating them to SAL and then performing model check on the

SAL well formed translation. The results of the model check would be displayed in a new Eclipse view.

The new system design starts from the domain model, where the SALTranslator, ModelChecker, WFChecker and ProofView notions are added. The WFChecker checks that the SAL obtained file is well formed and the ProofView is the Eclipse view where the results of the theories proving are displayed. All these notions can be grouped in a bigger notion, called *verification*. Thus the new domain model is the one presented in Listing 8.1.

```

cluster verification
  description "Contains all components that contribute to the verification process
    of an RSL project"
cluster_chart verification
  class ProofView
    description "Displays the results of theorems proving"
  class SALTranslator
    description "Translates RSL specification to SAL"
  class ModelChecker
    description "Checks that assertions are valid"
  class WFChecker
    description "Verifies if the SAL file is well formed"
end

```

Listing 8.1: Domain model for the new contribution

The next step involves identifying concepts' behaviour and behavior constrained.

```

class_chart ProofView
indexing
  in_cluster: "verification"
explanation
  "Displays the results of theorems proving"
command
  "Display the output of the model checker"
end
class_chart ModelChecker
indexing
  in_cluster: "verification"
explanation
  "Checks that assertions are valid"
constraint
  "The SML specification must be well formed"
end
class_chart SALTranslator
indexing
  in_cluster: "verification"
explanation
  "Translates RSL specifications to SAL"
constraint
  "The RSL specification has no syntax or type check errors"
end
class_chart WFChecker
indexing
  in_cluster: "verification"

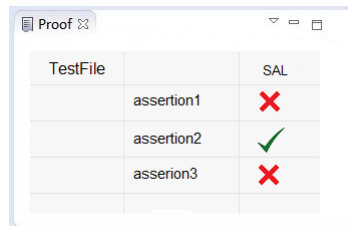
```

```

explanation
  "SAL"
constraint
  "Verifies if the SAL file is well formed"
end

```

Listing 8.2: Class charts inside the verification cluster chart



TestFile	SAL
assertion1	✗
assertion2	✓
asserion3	✗

Figure 8.1: Proof view

Next the user interface is designed to define what a user can do. The ProofView can e.g., be the one presented in Figure 8.1 where an RSL assertion that was successfully proved has a "check mark" next to it, otherwise it has a "fail mark". Once with the GUI design, the requirements are documented using EBON *scenario_chart*.

```

scenario_chart PROOF
  scenario "PRF1"
    description "The name of the RSL file, the assertions and their evaluation
      is displayed in a Proof view when a RSL specification is modified"
  scenario "PRF2"
    description "If the assertion is valid a green check is displayed next
      to its name in the ProofView"
  scenario "PRF3"
    description "If the assertion is not valid a red cross is displayed next
      to its name and the file containing the counter example is stored in
      the same directory as the RSL specification"
  scenario "PRF3"
    description "By clicking the red cross a file is opened containing
      the counterexample file for the assertion"
end

```

Listing 8.3: PROOF scenario chart

For example the ProofView can be updated every time the user modifies the RSL specification. This scenario is captured in Listing 8.3 along with three others, that was elicited analyzing the user interface.

```

event_chart UserActions
  incoming
  event "SAVE: User selects the File->Save menu or presses Ctrl+s"
  event "CROSSCLICK : User clicks the cross button in ProofView"

```

```

end
event_chart UserNotification
  outgoing
  event "PROOFUPDATE: Proof view update with the name of the translated files,
        the name of the assertions and the associated icon"
  event "FILECR: Store the counterexample file in the same directory "
  event "EXDISPLAY: The file containing the counter example is opened in
        the editor"
end

```

Listing 8.4: incoming and outgoing events

Next the incoming and outgoing events are identified as presented in Listing 8.4. These events are used to rewrite the scenario charts as presented in Listing 8.5.

```

scenario_chart PROOF
  scenario "PRF1"
    description "PROOFUPDATE when SAVE"
  scenario "PRF2"
    description "If the assertion is valid implies PROOFUPDATE"
  scenario "PRF3"
    description "If the assertion is not valid implies PROOFUPDATE and FILECR"
  scenario "PRF3"
    description "CROSSCLICK implies EXDISPLAY its associated counterexample"
end

```

Listing 8.5: Scenario chart containing events

Components are identified and added in a static diagram as presented in Listing 8.8.

```

cluster verification
  component
    class ProofView
    class ModelChecker
    class SALTranslator
    class WFChecker
  end
end

```

Listing 8.6: Components inside the static diagram

The components communication is the one presented in Listing 8.7 since the model checking must be performed on a well formed SAL translation. The ModelChecker call the services of ProofView in order to display the results.

```

WFChecker client SALTranslator
ModelChecker client WFChecker
ModelChecker client ProofView

```

Listing 8.7: Components communication inside the static diagram

The last step before code generation is establishing the contracts between components by identifying the information the client needs and the messages it sends to its supplier. The ProofView supplies a feature through which it is notified that it must update the view with a new set of assertions.

```

class WfCheck
  feature
    check: SET[ERROR]
      ->CONTEXT
      require
        context /= Void
      end
  end
class ProofView
  feature
    update
      -> SET[assertions]
  end
class SALTranslator
  feature
    translate
      ->context: CONTEXT
      require
        context /= Void
      end
  end
end

```

Listing 8.8: Components inside the static diagram

Finally the implementation can be started from the code skeleton generated by Beetlz and from the list of scenarios previously identified.

Of course, this is one approach on how to develop new plug-ins. There are many other possibilities, and the developer can choose whatever suits her most. E.g., another possibility would be to develop the new plug-ins using the RAISE formal specification method. In order to write and test the RSL specifications describing the new tools, the developers can use eRAISE. This way extensions for eRAISE will be designed and analyzed inside eRAISE.

Conclusions

This chapter draws the conclusions of the project. The first section summarizes the achievements while the last one presents the overall impressions with the project.

9.1 What was achieved

The goal of this master thesis was twofold. Firstly, to create an Eclipse based development environment for RAISE that is easy to extend and secondly to test EBON's applicability on plug-ins development, using the tool under development as a case study.

The paper started by investigating RAISE and Eclipse in order to provide a general overview of their capabilities, and continued with analyzing EBON and its available tools.

Based on the BON methodology, eRAISE was designed to be loosely coupled and easy to extend. The methodology used to design and analyze the system has made the subject of a paper that was published by the author of this thesis and her supervisors.

Starting from the Java skeleton that was generated from the EBON documentation, eRAISE was implemented as a set of four plug-ins, easily extendable through the concepts of extensions and extension points. Furthermore, the plug-ins implementation follows the Eclipse House Rules [Gam04], developed by the Eclipse community. The RAISE functionality that was integrated in Eclipse is syntax and type checking of RSL specifications, translating RSL specifications to SML, executing RSL test cases and generating L^AT_EX documentation. In addition some Eclipse functionality was added in order to enhance user experience: view for displaying test cases, RSL editor, errors view, RSL menu for handling all files in the workspace and toolbar buttons.

eRAISE has been tested and provided with a user guide that gives an overview of its functionality.

The report provides also ideas on how can eRAISE be extended and improved. From these ideas, an example is taken, designed and analyzed using the same methodology that was used for designing and analyzing eRAISE.

9.2 Conclusions

The master theses project was a great opportunity to work on a big project that involved many new technologies. The project offered many technical challenges that were solved by applying theory, analyzing other projects and by learning from others experience. The biggest challenge was the lack of up to date documentation for Eclipse 4.2 since this version was released right before the thesis was started.

Since eRAISE is easily extendable and easy to use, we hope that it will be the starting point of many contributions and that it will help RAISE become a widely used method in software development.

APPENDIX A

Article

A Rigorous Methodology for Analyzing and Designing Plug-ins

Marieta V. Fasia
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: marietafasie@gmail.com

Anne E. Haxthausen
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: aeha@dtu.dk

Joseph R. Kiniry
DTU Compute
Technical University of Denmark
DK-2800 Lyngby, Denmark
Email: jkin@dtu.dk

Abstract—Today, GUI plug-ins development is typically done in a very ad-hoc way, where developers dive directly into implementation. Without any prior analysis and design, plug-ins are often flaky, unreliable, difficult to maintain and extend with new functionality, and have inconsistent user interfaces. This paper addresses these problems by describing a rigorous methodology for analyzing and designing plug-ins. The methodology is grounded in the Extended Business Object Notation (EBON) and covers informal analysis and design of features, GUI, actions, and scenarios, formal architecture design, including behavioral semantics, and validation. The methodology is illustrated via a case study whose focus is an Eclipse environment for the RAISE formal method’s tool suite.

I. INTRODUCTION

Plug-ins, especially in the realm of plug-ins that wrap existing research command-line tools, are notoriously badly designed. Academics simply do not have the resources and expertise to execute on the design and implementation of a quality plug-in. Partly this is due to the fact that there are few examples of best practices in the area, and partly it is because plug-in development is viewed as the dirtiest of the dirty-but-necessary jobs of “selling” systems technology.

Eclipse plug-in development is a tricky world. Concepts like features, plug-ins, extension points, windows, views, etc. abound. Enormous, poorly documented APIs are prolific in the Eclipse ecosystem. To implement even the most basic of features sometimes takes hours of digging to find the right three lines of code, and then those lines must change when a new major version of Eclipse comes out. This is a frustrating world for researchers who want to package their demonstrable, useful tools for the Eclipse IDE.

This work is an attempt to help resolve these issues. We provide a *rigorous step-wise methodology through which one can do the analysis, architecture design, and user interface (UI) design of a plug-in for an arbitrary integrated development environment (IDE)*.

The methodology used is based upon the Business Object Notation (BON), an analysis and design methodology promoted by Walden and Nerson in the mid-90s within the Eiffel community [1]. Ostroff, Paige, and Kiniry formalized parts of the BON language and reasoned about BON specifications [2], [3], [4], [5]. Fairmichael, Kiniry, and Darulova developed the BONc and Beetz tools for reasoning about BON specifications

and their refinement to JML-annotated Java.¹ Finally, Kiniry and Fairmichael have extended BON in a variety of ways to produce Extended BON (EBON), which permits one to add new domain-specific syntax and semantics to the core BON language [6].

For the reader who has never heard of EBON, think of it as the subset of UML that might actually have a clear, unambiguous semantics. EBON’s core features are that it is *seamless*, insofar as you use the same specification language for everything from domain analysis to formal architecture specification and its behavior, *reversible* insofar as code generation and reverse engineering to and from code to EBON is straightforward and tool-supported, and *contracted* as formal abstract state-based contracts (invariants, pre, and postconditions) are the fundamental notion used to specify system behavior. EBON has both a textual and a graphical syntax, a formal semantics expressed in higher-order logic, a formal semantics of refinement to and from OO software, and tool support for reasoning about specifications, expressing specifications textually or graphically, generating code from models and models from code, and reasoning about refinement to code.

The methodology is illustrated on a case study that develops an Eclipse environment for the RAISE formal method and specification language (RSL) [7]. The project is called *eRAISE* and it is currently under development at DTU. The RAISE tool suite (*rsrtc*) [8], [9] consists of a type checker and some extensions to it supporting activities such as pretty printing, translation to other languages, generation of proof obligations, and execution of test cases. *rsrtc* has a command-line interface that exposes different capabilities selected via switches, but is also used from Emacs using menus and key-binding. However, although it is easy to use for the user comfortable with command-line tools or Emacs, we expect that the creation of a modern Eclipse-based development environment for *rsrtc* would broaden its appeal to mainstream software engineers and better enable its use for university-level pedagogy.

II. ANALYSIS AND DESIGN METHOD

The EBON methodology as applied to Eclipse plug-in development has six steps described shortly in the following. These steps can either be performed in sequence or in some iterative manner. More details on the steps and the full

¹See <http://tinyurl.com/brgrcz> for more information.

```

system_chart eRAISESystem
cluster RSLPerspective
description "The Eclipse RAISE perspective. It contains all
components and functionality relevant for a RAISE project"

cluster_chart RSLPerspective
class Console
description "Displays the output of components"
...

class_chart Console
command "Displays informative or error messages",
constraint "Delete content before displaying a new message"

```

Listing 1. Excerpts of a system chart describing the eRAISE system.

```

scenario "TypeCheckAllMenu"
description "The user can type check all RSL files in the
workspace. Success or failure messages will be displayed along
with the list of errors in case of a failure"

```

Listing 2. Scenario for a menu in eRAISE.

specification of our case study will be available in a technical report [11].

Step 1: Domain Modeling. In the first step the most important entities and high level classifiers related to the system domain are identified, explained and documented. The identified notions are documented as classes, which can be grouped under clusters and all these make up a unique system. Listing 1 illustrates a caption of the eRAISE System specified in EBON notation. The domain model also describes how concepts behave and how their behavior is constrained.

Step 2: User Interface. In this step, for each user action relevant for the plug-in, a mock-up user interface is drawn, and the requirements for the actions are documented in EBON *scenario_chart* elements. As an example, Listing 2 presents the requirements for one of the menus in the eRAISE case study.

Step 3: Events. This step identifies the external actions that make the system react and the system's outgoing responses. The external actions are captured as *incoming events* and the possible responses as *outgoing events* in EBON *event charts*. For the eRAISE case study, one of incoming events is shown in Listing 3. One of the possible system responses to this action is captured in Listing 4.

Step 4: Components. This step looks *inside* the system at the components that constitute its architecture. The high level classifiers described in the system domain model captured in step 1 are transformed into concrete data types.

Step 5: Components Communication. First, by inspecting the events from step 3 and the scenarios from step 2, it is identified which components interact with each other. Then component interfaces are described using parameterized classes that contain formally specified features.

Step 6: Code Generation. In the last step a tool named Beetlz [12] is applied to automatically generate JML-annotated, Javadoc documented Java code from the EBON specifications created in the previous steps.

III. RELATED WORK

There is little published work that focuses on methodologies specific to plug-in development. E.g., Lamprecht et

```

event "TYPECHECKALL: User clicks RSL menu and then clicks on
Type Check all option or presses Ctrl+Alt+T"
involves TypeChecker, Console

```

Listing 3. An incoming event in eRAISE.

```

event "CONSOLEUPDATE: Success or failure messages displayed
in console"
involves Console, TypeChecker

```

Listing 4. An outgoing event in eRAISE.

al. reflect over some simplicity principles elicited by many years' experience in plug-in development [10], but do not provide a methodology. We speculate that there is not much published work because plug-in development was not the focus of scientists until recently. Moreover, it is a fair question whether or not plug-in development is any different from normal systems development where a GUI is involved. We believe that plug-in development is different from normal GUI development as plug-ins must integrate into the larger framework of the IDE, deal with non-GUI events, and work in arbitrary compositions.

REFERENCES

- [1] K. Waldén and J.-M. Nerson, *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice-Hall, Inc., 1995.
- [2] J. Lancaric, J. Ostroff, and R. Paige, "The BON CASE tool," Details available via http://www.cs.yorku.ca/~ciffel/bon_case_tool/, Mar. 2002.
- [3] J. R. Kiniry, "The Extended BON tool suite," 2001, available via <http://ebon.sourceforge.net/>.
- [4] R. Paige, L. Kaminskaya, J. Ostroff, and J. Lancaric, "BON-CASE: An extensible CASE tool for formal specification and reasoning," *Journal of Object Technology*, vol. 1, no. 3, 2002, special issue: TOOLS USA 2002 Proceedings. Available online at <http://www.jot.fm/>.
- [5] R. F. Paige and J. Ostroff, "Metamodelling and conformance checking with PVS," in *Proceedings of Fundamental Aspects of Software Engineering*, ser. Lecture Notes in Computer Science, vol. 2029. Springer-Verlag, Apr. 2001, also available via <http://www.cs.yorku.ca/techreports/2000/CS-2000-03.html>.
- [6] J. R. Kiniry, "Kind theory," Ph.D. dissertation, Department of Computer Science, California Institute of Technology, 2002.
- [7] The RAISE Language Group, *The RAISE Specification Language*, ser. BCS Practitioner Series. Prentice Hall, 1992.
- [8] "RAISE Tool User Guide," 2008. [Online]. Available: http://www.iist.unu.edu/newrh/III/3/1/docs/rsrtc/user_guide/html/ug.html
- [9] C. George, "The Development of the RAISE Tools," in *Formal Methods at the Crossroads. From Panacea to Foundational Support*, ser. Lecture Notes in Computer Science, B. K. Aichernig and T. Maibaum, Eds. Springer Berlin Heidelberg, 2003, vol. 2757, pp. 49–64. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-40007-3_4
- [10] S. Naujokat, A. Lamprecht, B. Steffen, S. Jorges, and T. Margaria, "Simplicity principles for plug-in development: The jabc approach," in *Developing Tools as Plug-ins (TOP1)*, 2012 2nd Workshop on, June 2012, pp. 7–12.
- [11] M. V. Fasie, "An Eclipse based Development Environment for RAISE," Master's thesis, DTU Compute, Technical University of Denmark, to appear May 2013.
- [12] E. Darulová, "Beetlz - BON software model consistency checker for Eclipse," Master's thesis, University College Dublin, 2009.

APPENDIX B

eRAISE domain model

```
system_chart eRAISESystem
    indexing
        author: "Marieta Vasilica Fasie"
cluster RSLPerspective
    description "The Eclipse RAISE perspective. It contains all components and
    functionality relevant for a RAISE project"
end

cluster_chart RSLPerspective
    cluster core
        description "Comprises functionality for typechecking an RSL specification,
        translating it to SML, executing the test case and for generating the Latex
        documentation"
    cluster editor
        description "Groups all editor specific functionality"
    cluster testcases
        description "Contains the elements displaying the RSL test cases"
    cluster wizard
        description "Contains all components that are dealing with the creation of
        a new RSL project"
end

cluster_chart core
    class Console
        description "Displays the output of different components e.g. TypeChecker,
        SMLTranslator"
    class ResourceHandler
        description "Handles the actions done on workspace resources"
    class IRSLTestCaseListener
        description "The contract used by the test cases listeners"
    class SMLTranslator
        description "Translates RSL specifications to SML code"
    class LatexGenerator
        description "Integrates RSL specification in Latex"
```

```

class TestRunner
  description "Executes the SML files"
class TypeChecker
  description "The RSL syntax and type checker"
cluster guihandlers
  description "Comprises all UI handlers"
end

cluster_chart guihandlers
class TCHandler
  description "Notified when the 'Type check' button is pressed"
class TCallHandler
  description "Notified when 'Type check all' menu is selected"
class SMLHandler
  description "Notified when the 'SML translate' menu is selected"
class SMLAllHandler
  description "Notified when SML translate all menu is selected"
class RTHandler
  description "Notified when the 'Run test cases' button is pressed"
class RTAllCasesHandler
  description "Notified when 'Run all test cases' menu is selected"
class LatexHandler
  description "Notified when the 'Generate Latex' button is pressed"
class LatexAllHandler
  description "Notified when 'Generate Latex for all' menu is selected"
end

cluster_chart editor
class ConsoleToProblems
  description "Connects the Console and the Problems View"
class Problem
  description "Represents an RSL type check error"
class ProblemsView
  description "Displays any problem existing in the current workspace"
class RSLEditor
  description "The RSL text editor"
-- Future work ---
class OutlineView
  description "Displays the structure of the RSL specification"
class REPLView
  description "Read Evaluate Print Loop view. It evaluates expressions during
    a debug process, based on the current stack state. When the compiler
    reaches
    a breakpoint, the programmer can write expressions based on the currently
    active variables in the stack. The expressions will be evaluated based
    on their current value and the output will be displayed in the display
    view"
-- end future work --
cluster config
  description "Contains components that configure the editor for the RSL
    specification"
end

cluster_chart config
class ColorManager
  description "Manages the colors displayed inside the RSL editor"
class IRSLSyntaxColors
  description "Stores the colours used inside the RSL editor"
class MathWordDetector
  description "Identifies mathematical words inside the RSL editor"
class RSLCodeScanner
  description "Scans the RSL source in search for key words, strings and
    white spaces"
class RSLConfiguration

```

```

        description "Registers the source code scanners"
class RSLDocumentProvider
    description "Document provider for the RSL files"
class RSLDoubleClickStrategy
    description "Decides what text is selected when the user double clicks
        inside the editor"
class RSLPartitionScanner
    description "Scans the RSL source code based on the rules specified in
        the constructor"
class RSLWordDetector
    description "Defines rules for accepting a character as being part of a
        keyword"
class WhiteSpaceDetector
    description "Defines what characters can be considered whitespaces"
end

cluster_chart testcases
    cluster model
        description "Comprises the classes creating the test cases model that
            is displayed inside RTestView"
    cluster ui
        description "Contains the classes displaying the test cases results"
end

cluster_chart model
    class RSLTestCaseModel
        description "Stores the model that will be displayed in RTestView"
    class RSLTestFile
        description "Represents the first level in the RTestView"
    class TestCase
        description "Represents the second level in the RTestView"
    class TestCasesListener
        description "Implements the test cases listener"
end

cluster_chart ui
    class RTestView
        description "Defines the 'Test' View content as a TreeViewer"
    class TestCaseContentProvider
        description "Provides the content for the TestView"
    class TestCaseLabelProvider
        description "Provides the labels for the TestView"
end

cluster_chart wizard
    class NewRSLProjectWizard
        description "RSL new project wizard"
    class RSLPerspective
        description "Groups the RSL associated views and actions"
    class RSLProjectPage
        description "Page collecting the user input"
end

-----
-- classes inside the core
-----
class_chart Console
indexing
    in_cluster: "core"
explanation

```

```

        "Displays the output of different components e.g. TypeChecker, SMLTranslator"
command
    "Display informative or error messages",
    "Clear console content"
end

class_chart ResourceHandler
indexing
    in_cluster: "core"
explanation
    "Handles the actions done on workspace resources"
query
    "Are there any RSL files in this project?"
command
    "Add this file to this project",
    "Add this project in the workspace",
    "Move file from this source project to destination project"
end

class_chart IRSCTestCaseListener
indexing
    in_cluster: "core"
explanation
    "The contract used by the test cases listeners"
command
    "Modify SML file"
end

class_chart SMLTranslator
indexing
    in_cluster: "core"
explanation
    "Translates RSL specifications to SML code"
command
    "Translate this context into SML"
constraint
    "All RSL specification must be type checked first"
end

class_chart LatexGenerator
indexing
    in_cluster: "core"
explanation
    "Integrates RSL specification in Latex"
command
    "Integrate this context in Latex"
end

class_chart TestRunner
indexing
    in_cluster: "core"
explanation
    "Executes the SML files"
command
    "Execute the test cases from this context"
end

class_chart TypeChecker
indexing
    in_cluster: "core"
explanation
    "The RSL syntax and type checker"
query
    "Is this context and entity pair type correct?"
end

```

```
-----  
-- classes inside the guihandlers cluster  
-----  
class_chart THandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when the 'Type check' button is pressed"  
end  
  
class_chart TCallHandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when 'Type check all' menu is selected"  
end  
  
class_chart SMLHandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when the 'SML translate' menu is selected"  
end  
  
class_chart SMLAllHandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when SML translate all menu is selected"  
end  
  
class_chart RHandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when the 'Run test cases' button is pressed"  
end  
  
class_chart RTAllHandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when 'Run all test cases' menu is selected"  
end  
  
class_chart LatexHandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when the 'Generate Latex' button is pressed"  
end  
  
class_chart LatexAllHandler  
indexing  
    in_cluster: "guihandlers"  
explanation  
    "Notified when 'Generate Latex for all' menu is selected"  
end  
  
-----  
-- classes inside editor  
-----  
class_chart ConsoleToProblems  
indexing
```

```

        in_cluster: "editor"
explanation
    "Listens to the Console in order to update the ProblemsView"
end

class_chart Problems
indexing
    in_cluster: "editor"
explanation
    "Represents an RSL type check error"
end

class_chart ProblemsView
indexing
    in_cluster: "editor"
explanation
    "Displays all type check errors in the current workspace"
command
    "Display the set of problems"
end

class_chart RSLEditor
indexing
    in_cluster: "editor"
explanation
    "The RSL editor"
end

-- Future work ---
class_chart OutlineView
indexing
    in_cluster: "editor"
explanation
    "Displays the structure of the RSL specification"
command
    "Display the signature"
constraint
    ""
end

class_chart REPLView
indexing
    in_cluster: "editor"
explanation
    "Read Evaluate Print Loop view. It evaluates expressions during a
    debug process, based on the current stack state. When the compiler
    reaches a breakpoint, the programmer can write expressions based on
    the currently active variables in the stack. The expressions will be
    evaluated based on their current value and the output will be displayed
    in the display view"
command
    "Delete content"
constraint
    ""
end
-- end future work --

-----
-- classes inside the editor.config
-----

class_chart ColorManager
indexing
    in_cluster: "config"

```

```
explanation
    "Manages the colors displayed inside the RSL editor"
end

class_chart IRSLSyntaxColors
indexing
    in_cluster: "config"
explanation
    "Stores the colours used inside the RSL editor"
end

class_chart MathWordDetector
indexing
    in_cluster: "config"
explanation
    "Identifies mathematical words inside the RSL editor"
end

class_chart RSLCodeScanner
indexing
    in_cluster: "config"
explanation
    "Scans the RSL source in search for key words, strings and white spaces"
end

class_chart RSLConfiguration
indexing
    in_cluster: "config"
explanation
    "Registers the source code scanners"
end

class_chart RSLDocumentProvider
indexing
    in_cluster: "config"
explanation
    "Document provider for the RSL files"
end

class_chart RSLDoubleClickStrategy
indexing
    in_cluster: "config"
explanation
    "Decides what text is selected when the user double clicks inside the editor"
end

class_chart RSLPartitionScanner
indexing
    in_cluster: "config"
explanation
    "Scans the RSL source code based on the rules specified in the constructor"
end

class_chart RSLWordDetector
indexing
    in_cluster: "config"
explanation
    "Defines rules for accepting a character as being part of a keyword"
end

class_chart WhiteSpaceDetector
indexing
    in_cluster: "config"
explanation
    "Defines what characters can be considered whitespaces"
```

```

end

-----
-- classes inside the testcases.model cluster
-----

class_chart RSLTestCaseModel
indexing
  in_cluster: "model"
explanation
  "Stores the model that will be displayed in RTestView"
query
  "Are there any test cases in the RSL specification?"
command
  "Add an RSL specification to the model",
  "Remove an RSL specification from the model"
end

class_chart RSLTestFile
indexing
  in_cluster: "model"
explanation
  "Represents the first level in the RTestView"
end

class_chart TestCase
indexing
  in_cluster: "model"
explanation
  "Represents the second level in the RTestView"
end

class_chart TestCasesListener
indexing
  in_cluster: "model"
explanation
  "Implements the test cases listener"
end

-----
-- classes inside the testcases.ui cluster
-----

class_chart RTestView
indexing
  in_cluster: "ui"
explanation
  "Defines the 'Test' View content as a TreeViewer"
command
  "Display the set of test results"
end

class_chart TestCaseContentProvider
indexing
  in_cluster: "ui"
explanation
  "Provides the content for the TestView"
end

class_chart TestCaseLabelProvider
indexing
  in_cluster: "ui"
explanation
  "Provides the labels for the TestView"
end

```

```
-----  
-- classes inside the wizard  
-----  
class_chart NewRSLProjectWizard  
indexing  
    in_cluster: "wizard"  
explanation  
    "RSL new project wizard"  
end  
  
class_chart Perspective  
indexing  
    in_cluster: "wizard"  
explanation  
    "Groups the RSL associated views and actions"  
end  
  
class_chart RSLProjectPage  
indexing  
    in_cluster: "wizard"  
explanation  
    "Page collecting the user input"  
end
```

APPENDIX C

UI mock-ups

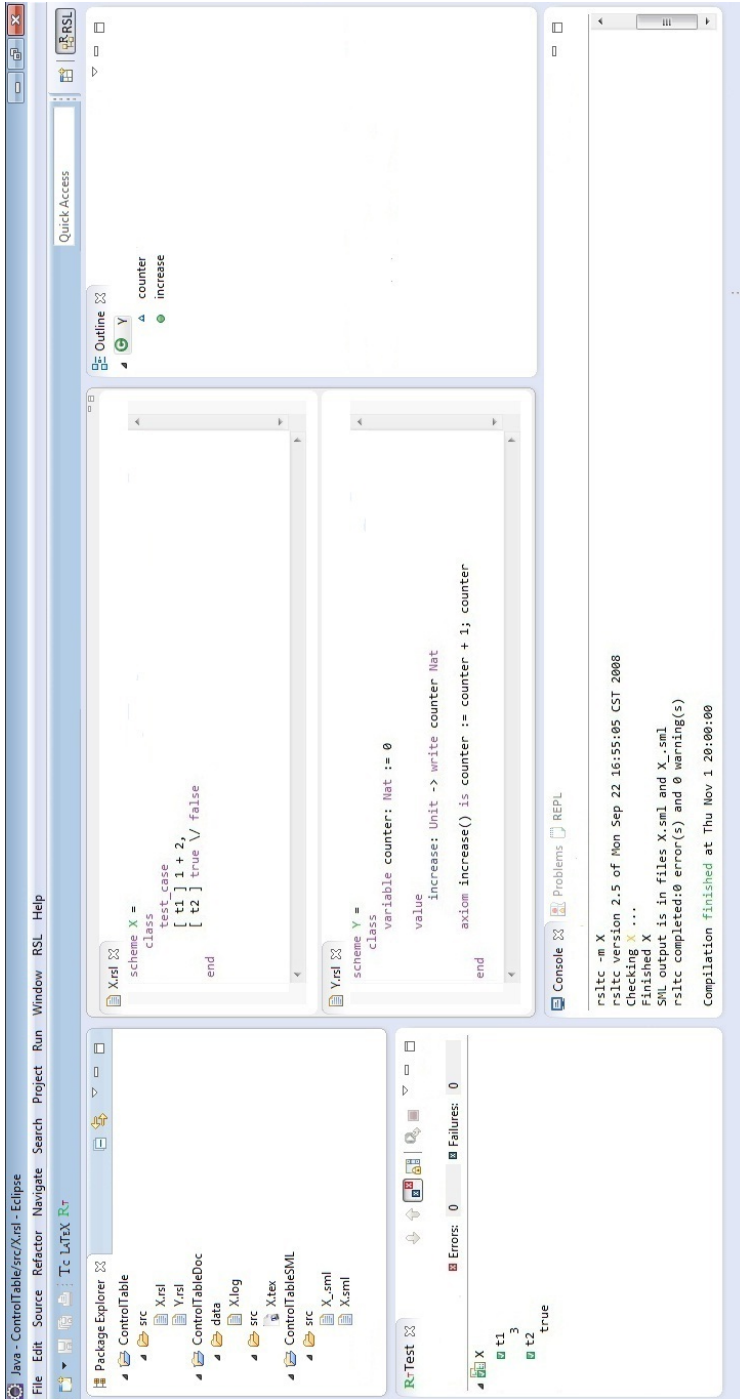


Figure C.1: RSL perspective

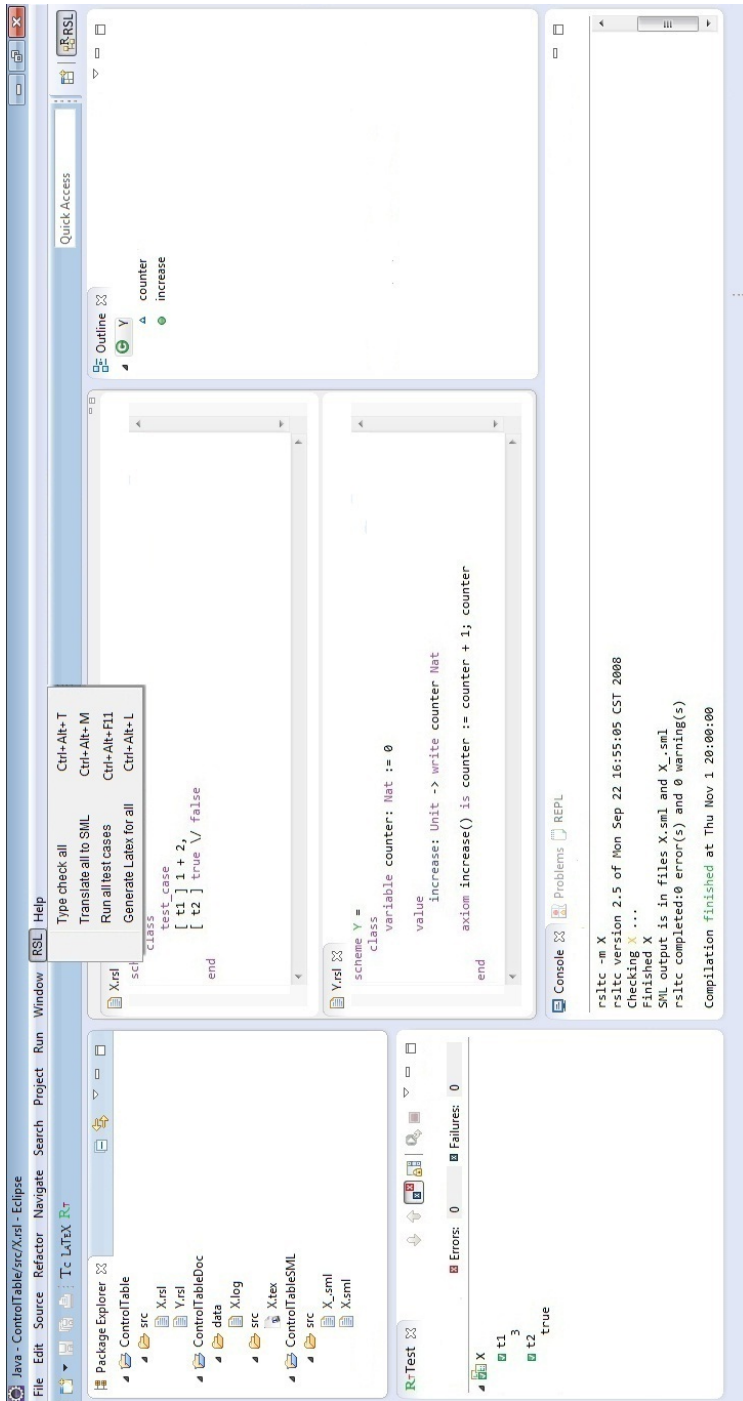


Figure C.2: RSL menu

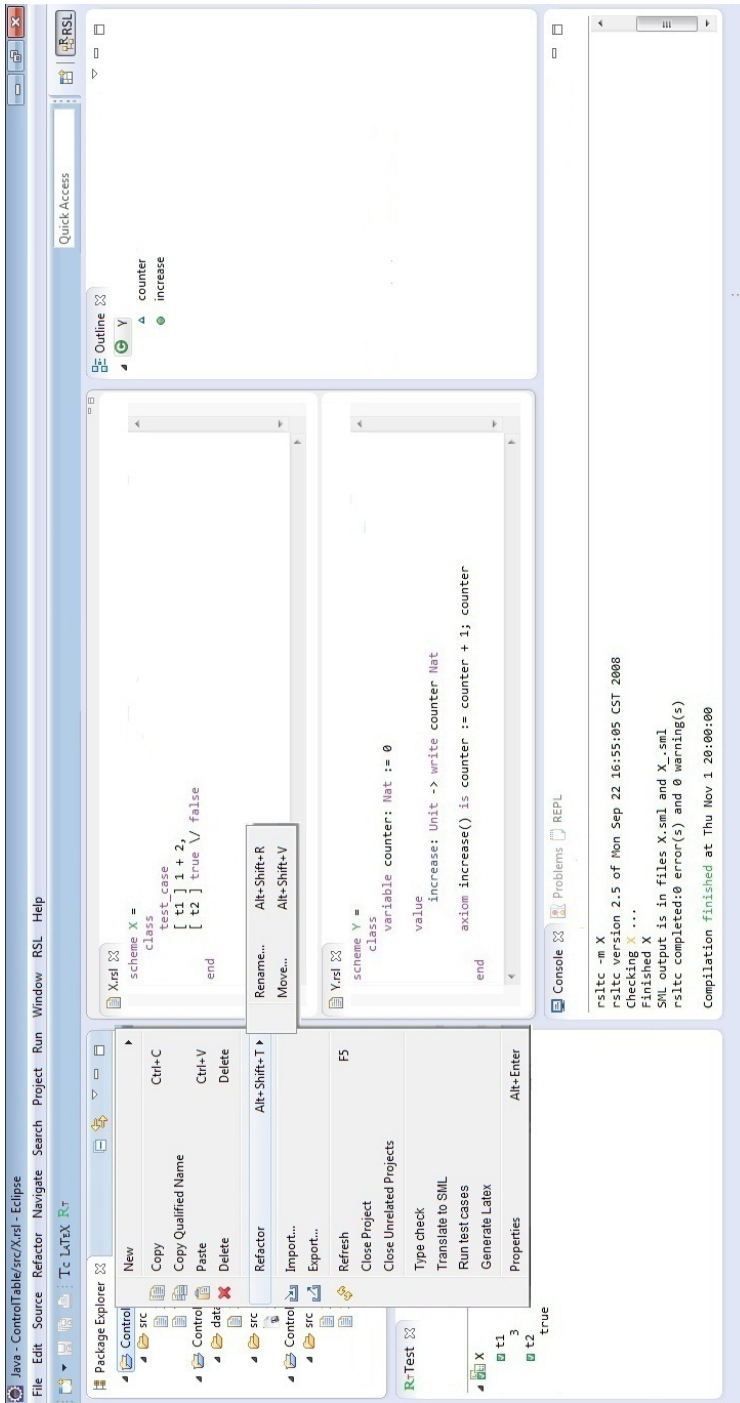


Figure C.3: RSL project context menu

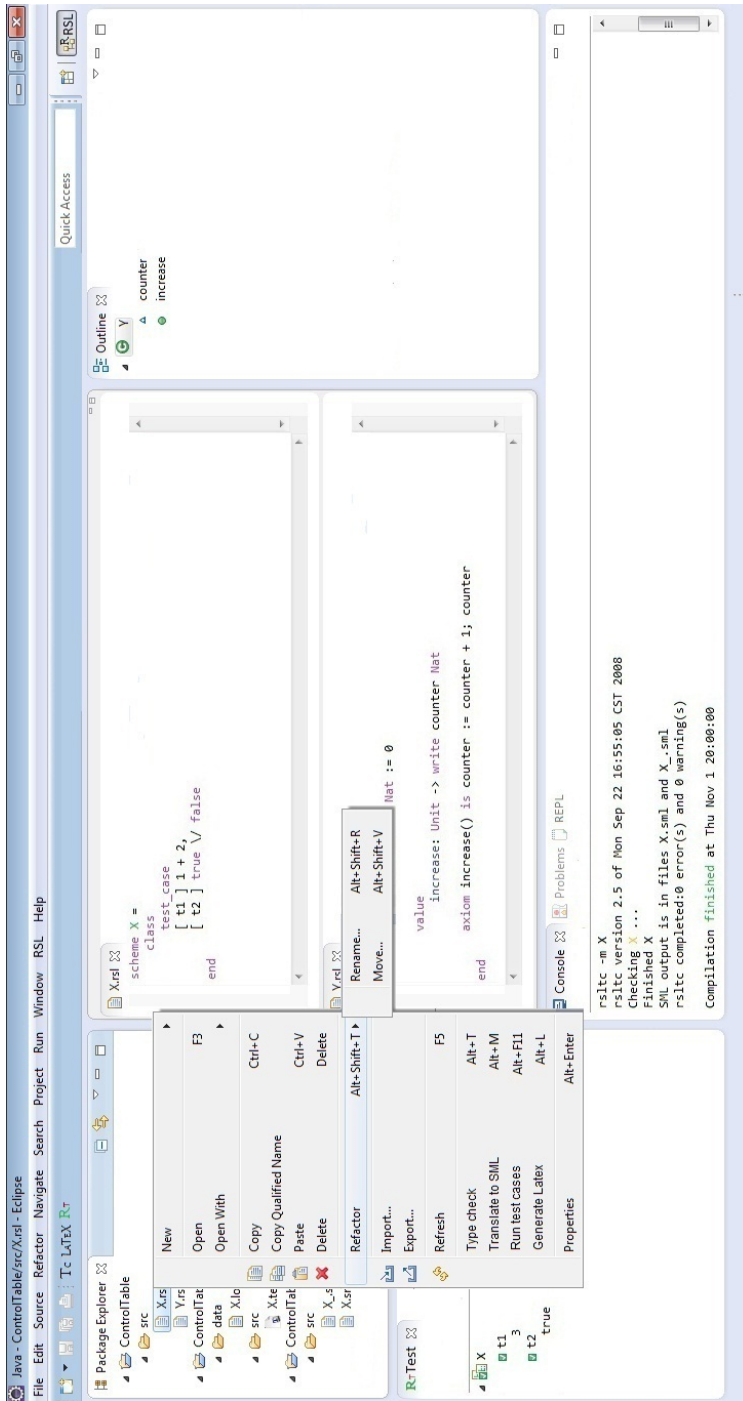


Figure C.4: RSL file context menu

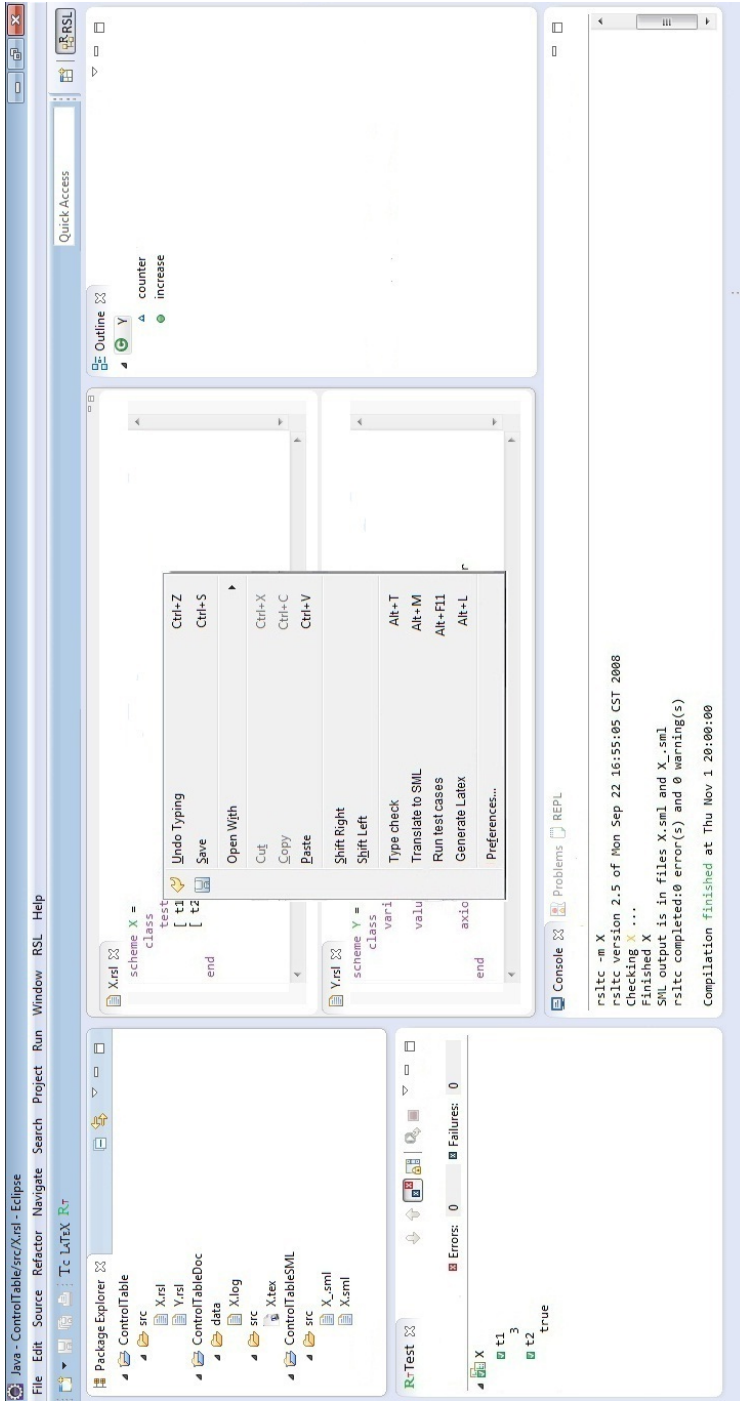


Figure C.5: RSL editor

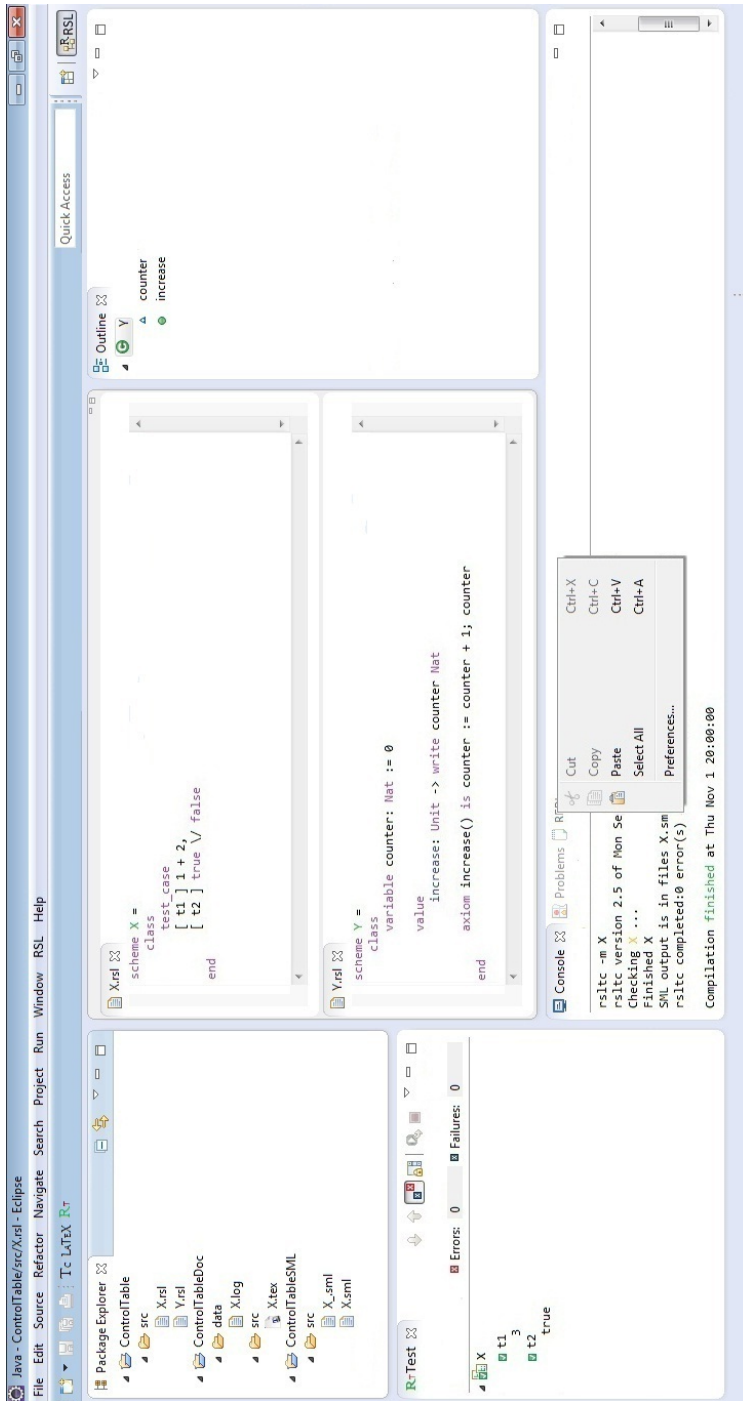


Figure C.6: RSL Console view

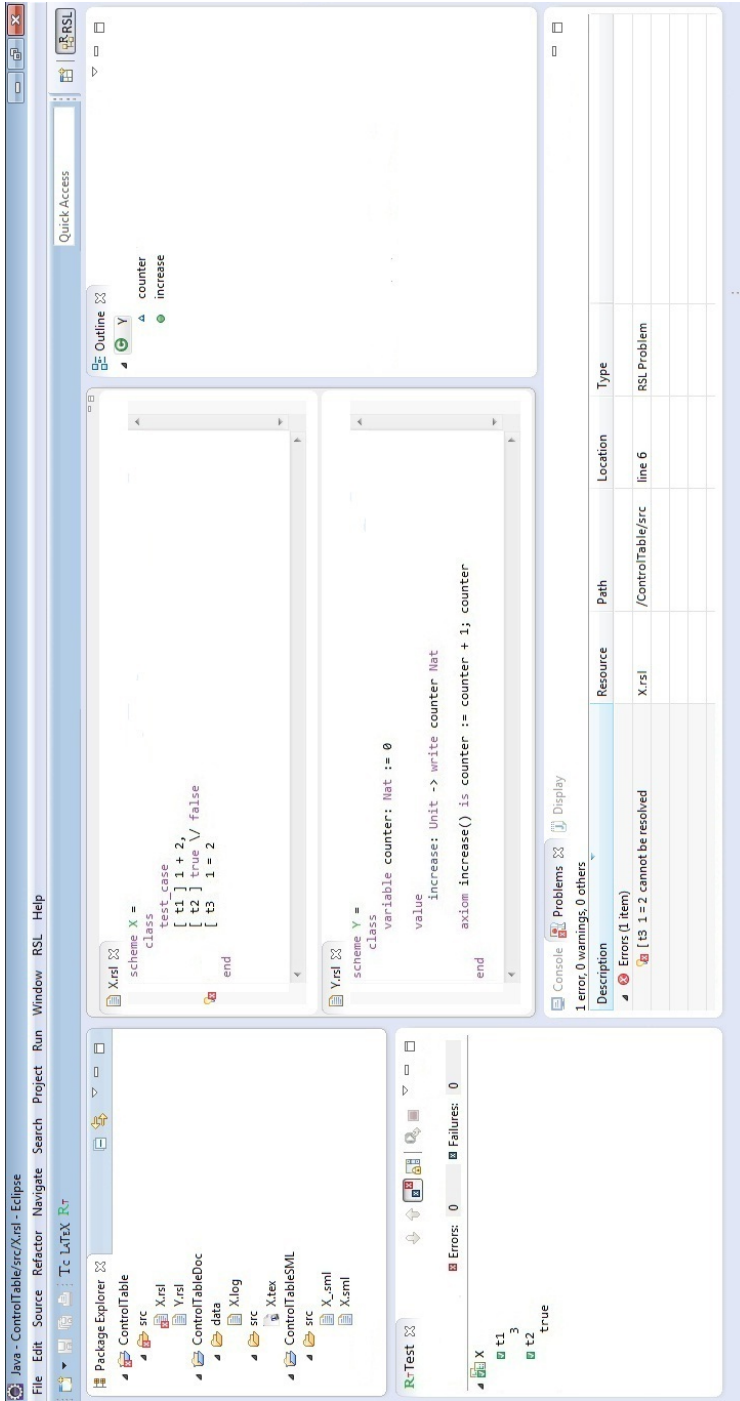


Figure C.7: RSL Problems view

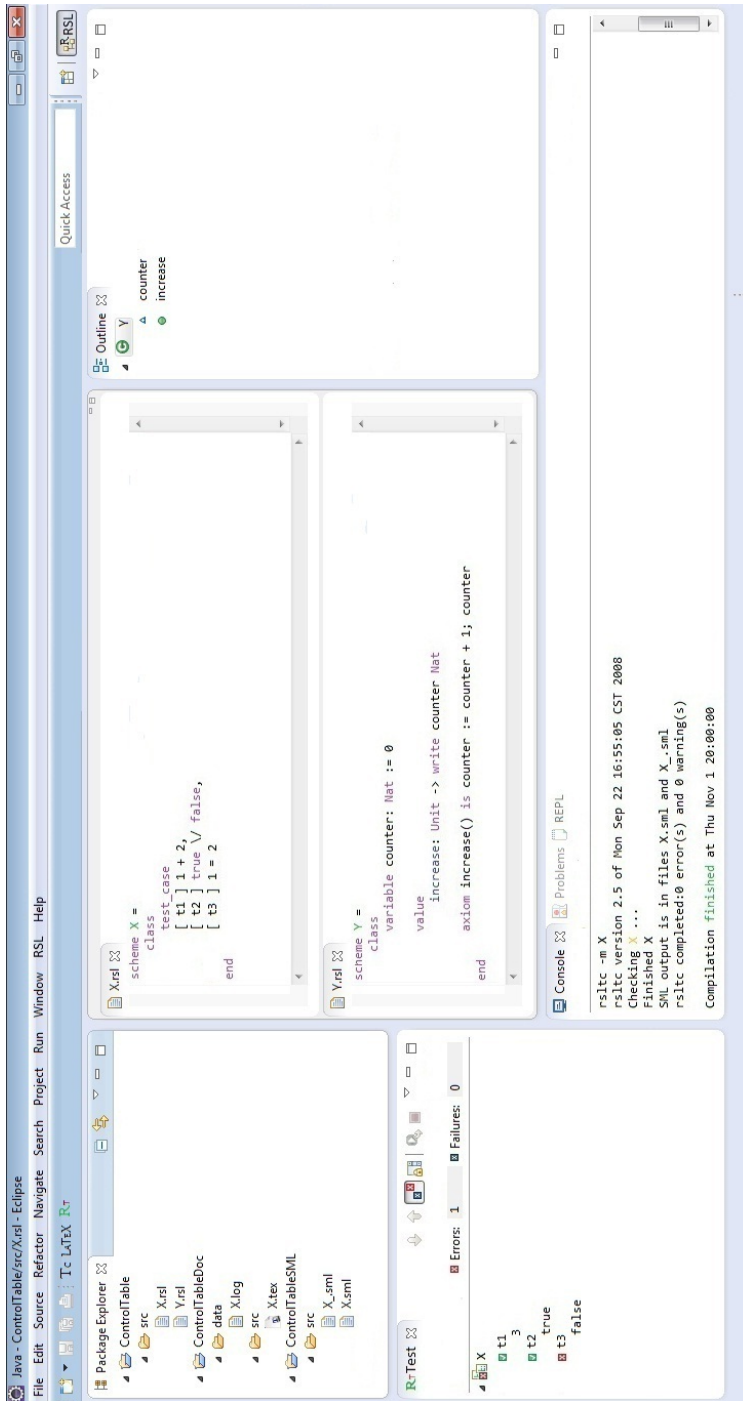


Figure C.8: RSL RTest view

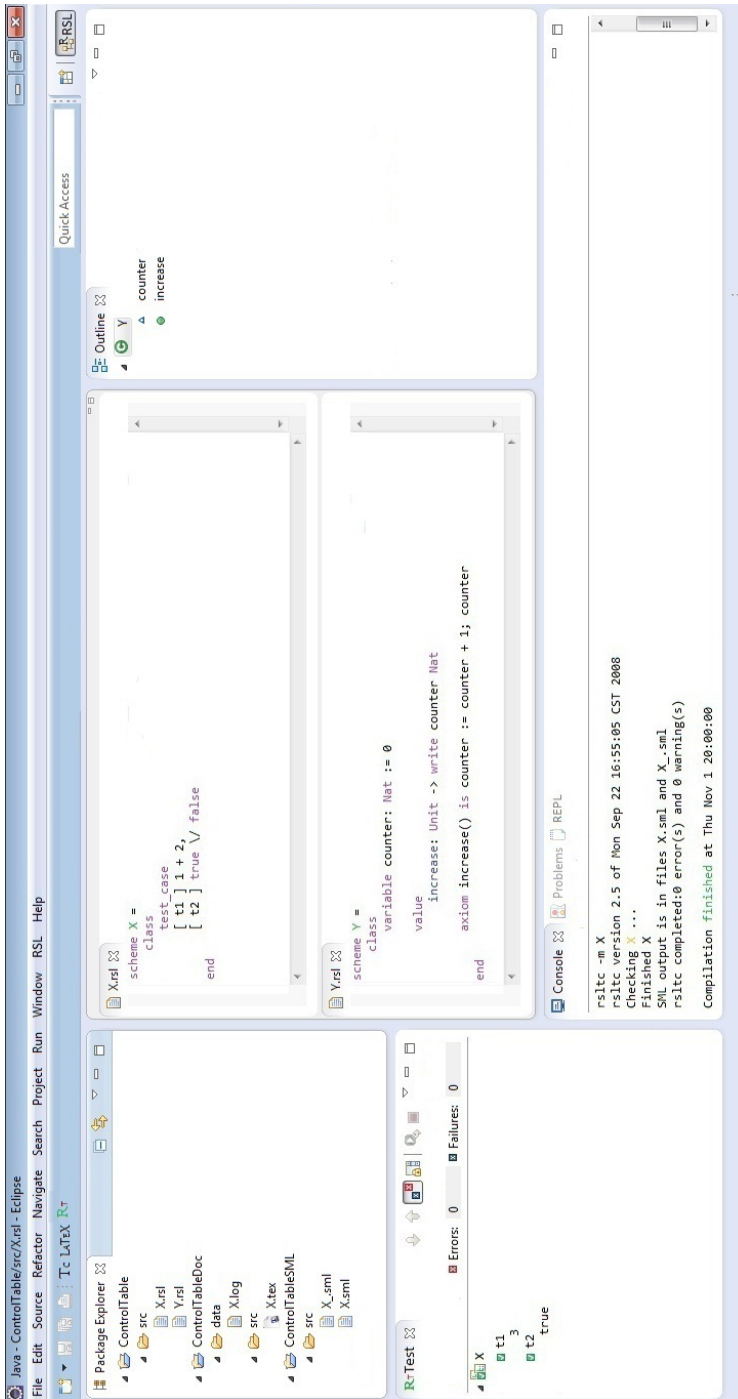


Figure C.9: RSL Outline view

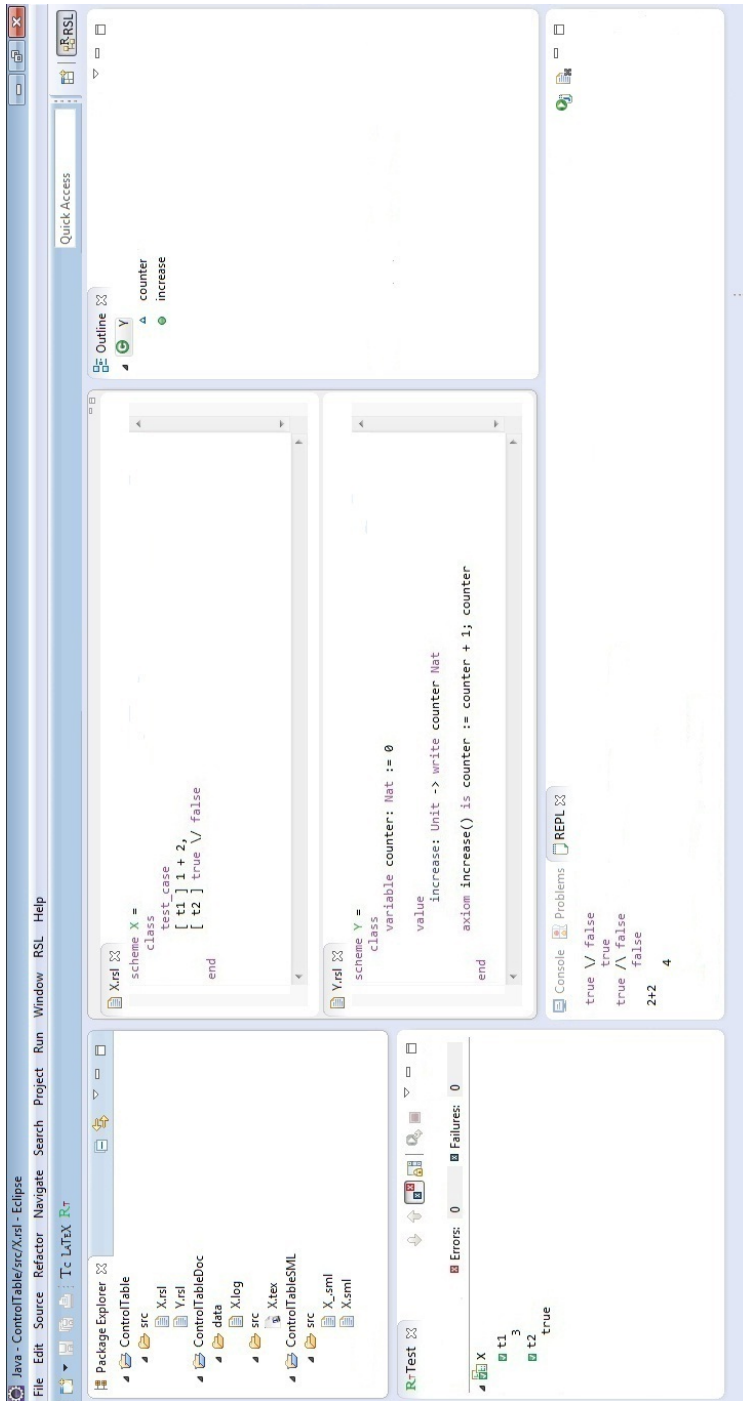


Figure C.10: RSL REPL view

APPENDIX D

Scenarios

```
scenario_chart RSLPERSPECTIVE
  scenario "R1"
    description "The user must be able to create a new RAISE project"
  scenario "R2"
    description "The user must be able to import an existing RAISE project."
  scenario "R3"
    description "When a new RAISE project is created, it contains a single
      folder named 'src'"
  scenario "R4"
    description "The user must be able to add an existing RSL file to an
      existing RSL project"
  scenario "R5"
    description "The user must be able to add a new RSL file to an existing
      RSL project"
  scenario "R6"
    description "All RSL actions and views are stored inside the RSL
      perspective"
  scenario "R7"
    description "A RAISE project can have associated one SML project
      that has the same name plus the suffix 'SML'. This project is created
      when the first SML file is created"
  scenario "R8"
    description "A generated SML file is stored in the SML project
      associated to the RAISE project that contains the translated
      RSL file"
  scenario "R9"
    description "A RAISE project can have associated one documentation
      project that has the same name plus the suffix 'DOC'. This project
      is created when the first documentation file is created"
  scenario "R10"
    description "A generated Latex file is stored in the Documentation
      project associated to the RSL project that contains the translated RSL
      file"
  scenario "R11"
```

```

        description "The documentation project contains a 'main.tex'
        file that includes all the other tex files in the documentation
        project"
    end

scenario_chart PROJECT_EXPLORER_FILE
    scenario "PRJF1"
        description "The user can TYPECHECK one RSL file. This implies
        CONSOLEUPDATE, PROBLEMSUPDATE and EDITORUPDATE"
    scenario "PRJF2"
        description "The user can SMLTRANSLATE one RSL file. This implies
        CONSOLEUPDATE, PROJECTEXPLORERUPDATE, PROBLEMSUPDATE and EDITORUPDATE"
    scenario "PRJF3"
        description "The user can RUNTESTS on one RSL file. This implies
        RTESTUPDATE, PROJECTEXPLORERUPDATE, PROBLEMSUPDATE and CONSOLEUPDATE"
    scenario "PRJF4"
        description "The user can GENERATELATEX for one RSL file. This implies
        CONSOLEUPDATE and PROJECTEXPLORERUPDATE"
    end

scenario_chart PROJECT_EXPLORER_PRJ
    scenario "PRJ1"
        description "The user can TYPECHECKPRJ RSL files in a project.
        This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    scenario "PRJ2"
        description "The user can SMLTRANSLATEPRJ RSL files in a project.
        This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    scenario "PRJ3"
        description "The user can RUNTESTSPRJ cases in a project. This
        implies RTESTUPDATE, PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    scenario "PRJ4"
        description "The user can GENERATELATEXPRJ for all RSL files in
        a project. This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    end

scenario_chart MENU
    scenario "TCALLMenu"
        description "The user can TYPECHECKALL RSL files in the workspace.
        This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    scenario "SMLaLLMenu"
        description "The user can SMLTRANSLATEALL RSL files in the workspace.
        This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    scenario "RunAllMenu"
        description "The user can RUNALLTESTS cases in the workspace.
        This implies RTESTUPDATE, PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    scenario "LatexAllMenu"
        description "The user can GENERATELATEXALL for all files in the
        workspace. This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
    end

scenario_chart TEST
    scenario "TEST1"
        description "The evaluation of test cases results in CONSOLEUPDATE,
        PROBLEMSUPDATE and EDITORUPDATE"
    scenario "TEST2"
        description "Tests results are shown in a separate view"
    end

scenario_chart PROBLEMS
    scenario "PRB1"
        description "The user can see the problems existing in the workspace
        with PROBLEMSUPDATE and EDITORUPDATE. "
    scenario "PRB2"
        description "For each problem the description, resource, path,

```

```
                location and type are specified"
end

scenario_chart CONSOLE
  scenario "CNS1"
    description "The user can see the success or failure messages
with CONSOLEUPDATE"
  end

scenario_chart EDITOR
  scenario "EDT1"
    description "The user can TYPECHECK the active file. This implies
PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
  scenario "EDT2"
    description "The user can SMLTRANSLATE the active RSL file.
This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
  scenario "EDT3"
    description "The user can RUNTESTS cases in the active RSL file.
This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
  scenario "EDT4"
    description "The user can GENERATELATEX files for the active
RSL file. This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
  scenario "EDT5"
    description "The user SAVE the current RAISE file and the TYPECHECK
is automatically run."
  scenario "EDT6"
    description "The user clicks on the typecheck button to TYPECHECK
the currently active file."
  scenario "EDT7"
    description "The user clicks on the run test cases button to RUNTESTS
in the currently active file."
  scenario "EDT8"
    description "The user clicks the Latex button to GENERATELATE
X for the currently active file"
  scenario "EDT9"
    description "RSL keywords and mathematical words are written
with a different colour in the editor"
  end

end
-- Future work --
scenario_chart REPL
  scenario "REPL1"
    description "The user can write ASCII characters in the REPL view"
  scenario "REPL2"
    description "The user can REPLEXECUTE"
  scenario "REPL3"
    description "The output of a user evaluation is REPLUPDATE"
  scenario "REPL4"
    description "The user can CHANGECONTEXT"
  scenario "REPL5"
    description "The user can clear the entire content of the REPL"
  end

end

scenario_chart OUTLINE
  scenario "OTL1"
    description "The user can see the signatures for the entire
project in a dedicated window"
  end

end
-- end future work --
```

Events

```
event_chart UserActions
  incoming
  explanation
    "External events triggering representative system behaviour"
  part "1/2"

  event "TYPECHECK: User clicks the type check button or user right clicks on a
  RSL file(or inside editor) and then clicks on Type Check option or user
  presses Alt+T" involves TypeChecker, Console, ConsoleToProblems,
  ProblemsView
  event "SMLTRANSLATE: User presses translate to SML button or user right
  clicks on a RSL file(or inside editor) and then clicks Translate
  to SML option or user presses Alt+M" involves SMLTranslator, TypeChecker,
  Console, ConsoleToProblems, ProblemsView, ResourceHandler, RTestView
  event "RUNTESTS: User clicks the run test cases button or user right clicks
  on RSL file(or inside editor) and then clicks on Run Test Cases
  option or presses Alt+F11" involves SMLTranslator, TypeChecker,
  Console, ConsoleToProblems, ProblemsView, ResourceHandler, RTestView
  event "GENERATELATEX: User clicks Latex button or user right clicks
  on a RSL file(or inside editor) and then clicks on Generate Latex or
  user presses Alt+L" involves LatexGenerator, Console, ResourceHandler
  event "TYPECHECKALL: User clicks RSL menu and then clicks on Type Check
  all option or presses Ctrl+Alt+C" involves ResourceHandler, TypeChecker,
  Console, ConsoleToProblems, ProblemsView
  event "SMLTRANSLATEALL: User clicks RSL menu and then clicks on Translate
  all to SML option or presses Ctrl+Alt+M" involves ResourceHandler,
  SMLTranslator, TypeChecker, Console, ConsoleToProblems, ProblemsView,
  ResourceHandler, RTestView
  event "RUNALLTESTS: User clicks RSL menu and then clicks on Run all test
  cases option or presses Ctrl+Alt+F11" involves ResourceHandler,
  SMLTranslator, TypeChecker, Console, ConsoleToProblems, ProblemsView,
  ResourceHandler, RTestView
  event "GENERATELATEXALL: User clicks RSL menu and then clicks on Generate
  Latex for all option or presses Ctrl+Alt+L" involves ResourceHandler,
```

```

    LatexGenerator, Console
event "TYPECHECKPRJ: User right clicks on a RSL project and selects
type check option" involves ResourceHandler, TypeChecker, Console,
ConsoleToProblems, ProblemsView
event "SMLTRANSLATEPRJ: User right clicks on a RSL project and then
clicks on translate to SML option" involves ResourceHandler, SMLTranslator,
TypeChecker, Console, ConsoleToProblems, ProblemsView, ResourceHandler,
RTestView
event "RUNTESTSPRJ: User right clicks on a RSL project and then clicks
on Run test cases option" involves ResourceHandler, SMLTranslator,
TypeChecker, Console, ConsoleToProblems, ProblemsView, ResourceHandler,
RTestView
event "GENERATELATEXPRJ: User right clicks on a RSL project and then
clicks on Generate Latex option" involves ResourceHandler, LatexGenerator,
Console
event "IMPORT: User imports a RAISE project or a RSL file" involves
ResourceHandler, TypeChecker, Console, ConsoleToProblems, ProblemsView
event "SAVE: User presses File and then Save option or Ctrl+s" involves
TypeChecker, Console, ConsoleToProblems, ProblemsView
-- future work --
event "REPLEXECUTE: User presses CTRL+enter in REPL view" involves
REPLView, SMLCompiler, SMLTranslator, TypeChecker
event "CHANGECONTEXT: User writes the name of the context and presses
enter to change context in REPL" involves
REPLView
event "SHOWONLYFAILURES: User presses the Show only failures in RTest
View" involves RTestView
-- end future work --
end

event_chart UserMessages
  outgoing
  explanation
    "Internal events triggering responses meant to inform the user.
    The list presented here concerns those events that are not explicitly
    requested by the user"
  part "2/2"

  event "PROBLEMSUPDATE: Problems view update" involves
    TypeChecker, Console, ConsoleToProblems, ProblemsView
  event "CONSOLEUPDATE: Success or failure messages displayed in console"
    involves Console, TypeChecker, SMLTranslator
  event "EDITORERRUPDATE: Display error message in editor" involves
    Editor, TypeChecker
  event "RTESTUPDATE: Displays the test cases interpretation" involves
    RTestView
  event "PROJECTEXPLORERUPDATE: Files and folders are added in the workspace"
    involves ResourceHandler
  -- future work --
  event "OUTLINEUPDATE: Outline view update" involves
    Editor, OutlineView, TypeChecker
  event "REPLUPDATE: Displays the evaluation result or an error in REPL window"
    involves REPL
  -- end future work --
end

```

APPENDIX F

Static diagram

```
static_diagram SystemArchitecture
--Shows the architecture of the eRAISE system
component
  cluster core
  component
    class Console
    class ResourceHandler
    class SMLTranslator
    class LatexGenerator
    class TestRunner
    class TypeChecker
    cluster guihandlers
    component
      class THandler
      class SMLHandler
      class RHandler
    end

    LatexGenerator client Console
    TypeChecker client Console
    LatexGenerator client Console
    SMLTranslator client TypeChecker
    THandler client TypeChecker
    TestRunner client SMLTranslator
    SMLHandler client SMLTranslator
    RHandler client TestRunner
    TestRunner client ResourceHandler

  end

  cluster editor
  component
    class ConsoleToProblems
    class Problems
```

```
class ProblemsView
class RSLEditor
--Future work--
class REPLView
end
class OutlineView
end
--end future work--
cluster config
component
    class ColorManager
end

ConsoleToProblems client ProblemsView
RSLEditor client config
end

cluster testcases
component
    cluster ui
    component
        class RTestView
    end
    cluster model
    component
        class RSLTestCaseModel
        class TestCasesListener

        TestCasesListener client RSLTestCaseModel
    end
    ui client model
end

cluster wizard
component
    class RSLProjectPage
    class NewRSLProjectWizard
    class RSLPerspective

    NewRSLProjectWizard client RSLProjectPage
end

editor client core
testcases client core

end
```

APPENDIX G

Components' interfaces

```
static_diagram Interfaces
--Shows the interfaces provided by the components
component
  cluster core
  component
    class Console
      feature
        update
          -> LIST[CHAR]
        clear
      end
    class ResourceHandler
      feature
        getRSLfiles: SET[FILE]
          ->project: PROJECT
          require
            project /= Void
          end
        addFile
          ->file: FILE
          ->directory: DIRECTORY
          require
            file /= Void;
            directory /= Void
          ensure
            file member_of directory
          end
        addProject
          ->directory: DIRECTORY
          ->project: PROJECT
          require
            directory /= Void;
            project /= Void
          ensure
```

```

        directory member_of project
    end
    moveFile
    ->scrProject: PROJECT
    ->destProject: PROJECT
    ->file: FILE
    ensure
        file /= Void;
        file member_of destProject;
        file not member_of srcProject
    end
end
class SMLTranslator
    feature
        translate
        ->context: CONTEXT
        require
            context /= Void
        end
    end
class LatexGenerator
    feature
        integrate
        ->context: CONTEXT
        require
            context /= Void
        end
    end
class TestRunner
    feature
        execute
        ->context: CONTEXT
        require
            context /= Void
        end
    end
class TypeChecker
    feature
        typeCheck: SET [CHAR]
        ->context: CONTEXT
        require
            context /= Void
        end
    end
end
cluster guihandlers
    component
        class TCHandler
        class SMLHandler
        class RTHandler
    end
end
end
cluster editor
    component
        class ConsoleToProblems
        class Problems
        class ProblemsView
            feature
                update
                -> problems: SET [PROBLEM]
            end
        end
        class RSLEditor
        -- Future work --
        class REPLView
    end
end

```

```
        feature
            clear
        end
    class OutlineView
        feature
            update
                -> SIGNATURE
            end
        -- end future work --
    end
end
cluster testcases
component
    cluster ui
    component
        class RTestView
            feature
                update
                    -> restResults: SET[TESTRESULT]
                end
            end
        end
    end
    cluster model
    component
        class RSLTestCaseModel
        class TestCasesListener
        end
    end
end
cluster wizard
component
    class RSLProjectPage
    class NewRSLProjectWizard
    class RSLPerspective
end
end
```

APPENDIX H

Generated Java code

```
//***** package core *****  
  
public /*@ nullable_by_default @*/ class SmlTranslator {  
    public void translate(/*@ non_null @*/ Context context){}  
}  
  
public /*@ nullable_by_default @*/ class TypeChecker {  
    public /*@ pure @*/ Set<Char> typeCheck(/*@ non_null @*/ Context context){}  
}  
  
public /*@ nullable_by_default @*/ class LatexGenerator {  
    public void integrate(/*@ non_null @*/ Context context){}  
}  
  
public /*@ nullable_by_default @*/ class ResourceHandler {  
    public /*@ pure @*/ Set<File> getRSLfiles(/*@ non_null @*/ Project project){}  
    public void addFile(/*@ non_null @*/ Directory directory, /*@ non_null @*/ File file){}  
    public void addProject(/*@ non_null @*/ Project project, /*@ non_null @*/  
        Directory directory){}  
    public void moveFile(Project destProject, File file, Project scrProject){}  
}  
  
public /*@ nullable_by_default @*/ class Console {  
    public void clear(){}
```

```
    public void update(List<Char> null){}
}

public /*@ nullable_by_default @*/ class TestRunner {

    public void execute(/*@ non_null @*/ Context context){}
}

//***** package ui *****

public /*@ nullable_by_default @*/ class RTestView {

    public void update(Set<Testresult> restResults){}
}

//***** package model *****

public /*@ nullable_by_default @*/ class TestCasesListener {
}

public /*@ nullable_by_default @*/ class RSLTestCaseModel {
}

//***** package guihandlers *****

public /*@ nullable_by_default @*/ class RTHandler {
}

public /*@ nullable_by_default @*/ class TCHandler {
}

public /*@ nullable_by_default @*/ class SmlHandler {
}

//***** package editor *****

public /*@ nullable_by_default @*/ class ConsoleToProblems {
}

public /*@ nullable_by_default @*/ class Problems {
}

public /*@ nullable_by_default @*/ class RSLEditor {
}

public /*@ nullable_by_default @*/ class ProblemsView {

    public void update(Set<Problem> problems){}
}

// future work
public /*@ nullable_by_default @*/ class Replview {

    public void clear(){}
}

//future work
public /*@ nullable_by_default @*/ class Outlineview {

    public void update(Signature signature){}
}

//***** package wizard *****
```

```
public /*@ nullable_by_default @*/ class RSLPerspective {
}

public /*@ nullable_by_default @*/ class RSLProjectpage {
}

public /*@ nullable_by_default @*/ class NewRSLProjectWizard {
}

//***** package Kernel *****

public /*@ nullable_by_default @*/ class Value {
}

public /*@ nullable_by_default @*/ class Any {
}

public /*@ nullable_by_default @*/ class String {
}

public /*@ nullable_by_default @*/ class File {
}

public /*@ nullable_by_default @*/ class None {
}

public /*@ nullable_by_default @*/ class Boolean {
}

//***** package Numbers *****

public /*@ nullable_by_default @*/ class Float implements Number {
}

public /*@ nullable_by_default @*/ class Number implements int {
}

public /*@ nullable_by_default @*/ class Integer implements Number {
}

//***** package Structures *****

public /*@ nullable_by_default @*/ class Tuple {
}

public /*@ nullable_by_default @*/ class Tree {
}

public /*@ nullable_by_default @*/ class Array {
}

public /*@ nullable_by_default @*/ class Table {
}

public /*@ nullable_by_default @*/ class Set {
}

public /*@ nullable_by_default @*/ class List < T > {
}
```

APPENDIX I

Prioritized scenarios

```
-- type check
scenario "PRJF1"
    description "The user can TYPECHECK one RSL file. This implies CONSOLEUPDATE,
    PROBLEMSUPDATE and EDITORUPDATE"
scenario "CNS1"
    description "The user can see the success or failure messages with CONSOLEUPDATE"
scenario "PRJ1"
    description "The user can TYPECHECKPRJ RSL files in a project. This implies
    PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario
"TCALLMenu"
    description "The user can TYPECHECKALL RSL files in the workspace.
    This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario
"EDT1"
    description "The user can TYPECHECK the active file. This implies
    PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"

-- sml translate
scenario "PRJF2"
    description "The user can SMLTRANSLATE one RSL file. This implies
    CONSOLEUPDATE, PROJECTEXPLORERUPDATE, PROBLEMSUPDATE and EDITORUPDATE"
scenario "R8"
    description "A RAISE project can have associated one SML project
    that has the same name plus the suffix 'SML'. This project is created
    when the first SML file is created"
scenario "R9"
    description "A generated SML file is stored in the SML project associated
    to the RAISE project that contains the translated RSL file"

scenario "PRJ2"
    description "The user can SMLTRANSLATEPRJ RSL files in a project.
    This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "SMLaLLMenu"
```

```

    description "The user can SMLTRANSLATEALL RSL files in the workspace.
    This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "EDT2"
    description "The user can SMLTRANSLATE the active RSL file. This implies
    PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"

-- run tests
scenario "PRJ3"
    description "The user can RUNTESTSPRJ cases in a project. This implies
    RTESTUPDATE, PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "PRJF3"
    description "The user can RUNTESTS on one RSL file. This implies
    RTESTUPDATE, PROJECTEXPLORERUPDATE, PROBLEMSUPDATE and CONSOLEUPDATE"
scenario "RunAllMenu"
    description "The user can RUNALLTESTS cases in the workspace. This implies
    RTESTUPDATE, PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "EDT3"
    description "The user can RUNTESTS cases in the active RSL file. This
    implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"

-- generate latex
scenario "PRJ4"
    description "The user can GENERATELATEXPRJ for all RSL files in a project.
    This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "PRJF4"
    description "The user can GENERATELATEX for one RSL file. This implies
    CONSOLEUPDATE and PROJECTEXPLORERUPDATE"
scenario "LatexAllMenu"
    description "The user can GENERATELATEXALL for all files in the workspace.
    This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"
scenario "EDT4"
    description "The user can GENERATELATEX files for the active RSL file.
    This implies PROBLEMSUPDATE, CONSOLEUPDATE and EDITORUPDATE"

-- editor
scenario "PRB1"
    description "The user can see the problems existing in the workspace
    with PROBLEMSUPDATE and EDITORUPDATE. "
scenario "PRB2"
    description "For each problem the description, resource, path, location
    and type are specified"
scenario "ED6"
    description "RSL keywords and mathematical words are written with a
    different colour in the editor"
scenario "EDT5"
    description "The user SAVE the current RAISE file and the TYPECHECK
    is automatically run."

-- test display
scenario "TEST1"
    description "The evaluation of test cases results in CONSOLEUPDATE,
    PROBLEMSUPDATE and EDITORUPDATE"
scenario "TEST2"
    description "Tests results are shown in a separate view involves
    RTESTUPDATE"

-- wizard
scenario "R1"
    description "The user must be able to create a new RAISE project"
scenario "R2"
    description "The user must be able to import an existing RAISE project."
scenario "R3"

```

```
    description "When a new RAISE project is created, it contains a single
folder named 'src'"
scenario "R4"
    description "The user must be able to add an existing RSL file to an
existing RSL project"
scenario "R5"
    description "The user must be able to add a new RSL file to an existing RSL project"
scenario "R6"
    description "All RSL actions and views are stored inside the RSL perspective"
scenario "R14"
    description "A RAISE project can have associated one documentation
project that has the same name plus the suffix 'DOC'. This project is
created when the first documentation file is created"
scenario "R16"
    description
"The documentation project contains a 'main.tex' file that includes
RSL files in the documentation project"

scenario_chart RSLPERSPECTIVE
scenario "R6"
    description "When a RAISE project is imported, it is automatically
TYPECHECKPRJ"
scenario "R10"
    description "A RAISE project can have associated one PVS project
that has the same name plus the suffix 'PVS'. This project is created
when the first PVS file is created"
scenario "R11"
    description "A generated PVS file is stored in the PVS project
associated to the RAISE project that contains the translated
RSL file"
scenario "R12"
    description "A RAISE project can have associated one SAL project
that has the same name plus the suffix 'SAL'. This project is created
when the first SAL file is created"
scenario "R13"
    description "A generated SAL file is stored in the SAL project
associated to the RAISE project that contains the translated
RSL file"

end
```

RSL key words

The list of RSL keywords presented alphabetically from left to right, up to down.

Bool	Char	Int	Nat
Real	Text	Unit	abs
any	as	axiom	card
case	channel	chaos	class
do	dom	elems	else
elseif	end	extend	false
for	forall	hd	hide
if	in	inds	initialise
int	len	let	local
object	of	out	post
pre	read	real	rng
scheme	skip	stop	swap
then	ti	true	type
test_case	until	use	value
variable	while	with	write

Table J.1: RSL key words

Key words that replace mathematical symbols:

all	exists	union	inter
isin	always	is	

Table J.2: RSL key words that replace mathematical symbols

APPENDIX K

Colours used inside RSL editor

The colours used inside the RSL editor

Token	Colour	RGB value
Keywords	pink	(127,0,85)
Texts	green	(0,128,0)
Characters	green	(0,128, 0)
Comments	red	(128, 0, 0)
Others	black	(0,0,0)

Table K.1: Colours used inside RSL editor

APPENDIX L

SML run-time errors

The SML execution of the RSL test cases, can generate run-time errors. Each error has a message associated and the complete list of the messages is further presented. The list is taken from [rsl08], and inside the message x represents values that are part of the message, constants or variables.

Invalid integer literal x
Division by zero
Modulo zero
Integer exponentiation with negative exponent x
Cannot compute $0 ** 0$
Invalid real literal x
Zero raised to non-positive power x
Negative number x raised to non-integer power y
 hd applied to empty set
Cannot select from empty set
 hd applied to empty list
 tl applied to empty list
List applied to index outside index set
Cannot select from empty list hd applied to empty map
Map applied to value outside domain
Nondeterministic enumerated map
Maps do not compose

Cannot select from empty map
List x applied to non-index y
Text x applied to non-index y
Map x applied to non-domain value y
x union y has non-disjoint domains
Cannot compute function equality
Destructor x applied to wrong variant
Reconstructor x applied to wrong variant
Argument of x(y) not in subtype
Precondition of x(y) not satisfied
Result z of x(y) not in subtype
Case incomplete for value x
Value x of c not in subtype
Initial value x of v not in subtype
Value x of v not in subtype
chaos encountered
stop encountered
swap encountered

Bibliography

- [Cer05] Gary Cernosek. A brief history of Eclipse, 2005.
- [Dar09] Eva Darulová. Beetlz - BON software model consistency checker for Eclipse. Master's thesis, University College Dublin, 2009.
- [Ecl] Eclipsepedia - the eclipse.org wiki. http://wiki.eclipse.org/Main_Page.
- [Ecl13] Eclipse - The Eclipse Foundation open source community website, 2013.
- [EGt] Egit. <http://www.eclipse.org/egit/>.
- [FK13] Haxthausen Anne E. Fasie, Marieta V. and Joseph R. Kiniry. A Rigorous Methodology for Analyzing and Designing Plug-Ins. pages 49 – 50, may 2013. Available via <http://www.conference-publishing.com/list.php?Event=ICSEWS13TOPI>.
- [for97] Formal Software Specification Using RAISE, 1997.
- [Gam04] Erich Gamma. *Contributing to Eclipse : principles, patterns, and plug-ins*. Addison-Wesley, Boston, 2004.
- [Geo03] Chris George. The Development of the RAISE Tools. In Bernhard K. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin Heidelberg, 2003.

- [Gro95] The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995. Available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book.
- [Hax99] Anne Haxthausen. Lecture Notes on The RAISE Development Method, 1999.
- [hel13] Eclipse Juno 4.2 documentation, 2013.
- [JUn] Junit. <http://www.junit.org/>.
- [Kin] KindSoftware. The BONc home page.
- [Kin01] Joseph R. Kiniry. The Extended BON tool suite, 2001. Available via <http://ebon.sourceforge.net/>.
- [Kin02] Joseph R. Kiniry. *Kind Theory*. PhD thesis, 2002.
- [LOP02] Jason Lancaric, Jonathan Ostroff, and Richard Paige. The BON CASE tool. Details available via http://www.cs.yorku.ca/~eiffel/bon_case_tool/, March 2002.
- [Myl] Mylyn.
- [NLS⁺12] S. Naujokat, A. Lamprecht, B. Steffen, S. Jorges, and T. Margaria. Simplicity principles for plug-in development: The jabc approach. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 7–12, June 2012.
- [Pai99] Ostroff Jonathan S. Paige, Richard F. A Comparison of the Business Object Notation and the Unified Modelling Language. Technical Report CS-1999-04, York University, 1999.
- [PKOL02] Richard Paige, Liliya Kaminskaya, Jonathan Ostroff, and Jason Lancaric. BON-CASE: An extensible CASE tool for formal specification and reasoning. 1(3), 2002. Special issue: TOOLS USA 2002 Proceedings. Available online at <http://www.jot.fm/>.
- [PO01] Richard F. Paige and Jonathan Ostroff. Metamodelling and conformance checking with PVS. In *Proceedings of Fundamental Aspects of Software Engineering*, volume 2029, April 2001. Also available via <http://www.cs.yorku.ca/techreports/2000/CS-2000-03.html>.
- [Rep] Reps, Thomas and Tciclbaum, Tim .
- [rsl08] RAISE Tool User Guide, 2008.
- [Sch05] Friedrich Wilhelm Schroer. The GENTLE compiler construction system, 2005. Details available via <http://gentle.compilertools.net/>.

- [SML] Standard ML of New Jersey. <http://www.smlnj.org/>.
- [Sub] Eclipse Subversive - Subversion (SVN) Team Provider.
- [The92] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.
- [top] Topi 2013: 3rd workshop on developing tools as plug-ins. <http://se.inf.ethz.ch/events/topi2013/Welcome.html>.
- [UNU] International Institute for Software Technology. United Nations University.
- [Wal] Kim Waldén. The business object notation home page.
- [WN94] Kim Walden and Jean-Marc Nerson. Seamless object-oriented software architecture - analysis and design of reliable systems, September 1994.
- [WN95] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. 1995.