A plugin for refactoring Event-B models

Akulina Lyapunova

Dissertation 2016

Erasmus Mundus MSc in Dependable Software Systems

Department of Computer Science,

Maynooth University,

Co. Kildare, Ireland.

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department: Dr. Adam Winstanley

Supervisor: Dr. James F. Power

10 June 2016

Declaration

I hereby certify that this thesis, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work. This thesis contains fewer than 22,000 words.

____

Akulina Lyapunova

10.06.2016

Acknowledgements

I would like to thank my supervisor Dr. James F. Power for his help, patience and guidance during the project. His optimism and high intelligence became an extra source of motivation for me. I would also like to thank Dr. Diarmuid O'Donoghue for his helpfulness and dedication to the Erasmus Mundus MSc in Dependable Software Systems program. Finally, I would like to thank Marie Farrell for her support and advices, which made it possible to have all this work done.

Abstract

This thesis describes a plugin which was created for developers who intend to design their models using the Event-B language. Event-B is a formal language which uses mathematical techniques for system modelling and verification. The accuracy of the model is ensured by proof obligations. The main disadvantage of the Event-B is that it doesn’t have many well-developed modularization constructs and it is not easy to combine specifications in Event-B with those written in other formal languages. Developers can use the plugin described in this thesis if they want to make changes in the existing model such as element renaming or merging, without writing new elements from the scratch. Developers can use their knowledge of the institution theory and specification based operators to interact with the plugin.

Software development requires from the developer not only an accurate and clear structuring of the system, as well as efficient tests for finding bugs, but also a strong mathematical proof. All these components allow software to be reliable and eliminate the possibility of system failure. The more complicated the system is, the more difficult it becomes to make sure that it works correctly and, in this context, mathematical proof can help to show the absence of bugs.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Developing dependable systems is one of the most important targets in software development nowadays. Technologies are involved in nearly everyone's daily life. People use smartphones, laptops, and different vehicles to solve their everyday problems. They rely on their gadgets which sometimes can work improperly and cause trouble. In those situations where the systems should be reliable and should work stably all the time, software, on which these systems are based, has to be verified. Verified software allows the user to be sure that it will work accurately in any situation. Reliable software usually is very expensive and is used in situations when any software fault can cause huge money losses or endanger people's lives. To decrease the danger of money losses or security risks, scientists analyze all the possible variants of the system's behavior and software developers design systems which work stably all the time. But analyzing the system is very time-consuming and difficult work, and requiring the developer to be sufficiently concentrated all the time is sometimes impossible.

To solve this problem and eliminate the human factor special tools have been developed. They allow the developer to verify their software easier and faster. One of these tools is the Rodin Platform, which provides effective support for refinement and mathematical proof [2]. Event-B [4] is not the only language which can be used in verification, but is one of the most popular ones. The practical work of this thesis is based on creating a Plug-in which manipulates Event-B elements with the help of formal expressions. The Rodin Platform [5] is Eclipse-based, so the Plug-in is written in Java. The work in this thesis should also help people who are familiar with the theory of institutions and specification-based operators (SBOs), but not familiar with Event-B syntax, to more easily manipulate Event-B elements by using SBOs inside the Rodin Platform.

1.1 Motivation

During the last couple of decades, the popularity of the software increased vastly. No one can imagine the work of banks, schools, hospitals without using modern technologies. Almost every single household appliance works under the control of software. Even the timetables for public transport are not created manually any more. People trust the software to control airplanes and train traffic. While life has become more convenient, it has also become more dangerous at the same time [9].

If software development is quite difficult work, then reliable software development is much more difficult. It requires not only accurate and clear structure and efficient tests to find the bugs, but also strong mathematical proofs. Software verification is very important, because the software testing can show the presence of the bugs, but never their absence [7]. Software verification can be done using formal methods. While studying formal methods, I met the problem that knowing only theory of institutions is not enough to start working with tools which provide software verification. One should not only understand the logic of the verification, but although study the syntax and logic of the verification language and tool one uses. Each platform has its own features and to study them all, a person would have to spend lots of time. It is not often easy, because developer may want to try another language and another tool and should start studying formal language and verification tools from the very beginning. If there is code written in some formal language, it becomes another problem to translate this code to another formal language. Event-B is quite popular language, but there are some known concerns regarding the using of Event-B formal language:

- it doesn't have well-developed modularization constructs and it is not easy to combine specifications in Event-B with those written in other formalisms [8];
- The Rodin Platform, which maintains the Event-B language is quite big and has lots of features, so it becomes quite difficult to start using it without knowing Event-B;
- If the developer is familiar with the theory of institutions, but not familiar with the Event-B, it will be hard to start working with Rodin Platform.

This theme for the thesis was chosen to decrease developers' efforts in writing dependable software and to make creating it as easy as possible. The best way of doing this is to make it possible to use the Rodin Platform to write some specification-based operators and manipulate with elements, written in Event-B [1]. This will allow the user to save time and to speed up the software development. Developers, using formal methods to prove their programs will have the opportunity to get a flexible tool which can be used to write proof obligations not only in Event-B, but also with the help of specification-based operators, and then to manipulate Event-B elements without knowing its syntax. The main currency nowadays is time, so studying lots of formal languages and understanding the main features of the tools, which use these languages, requires lots of time that can be spent on better software design.

1.2 Overall project objective

This thesis will provide a solution for the problem, which has been described in the previous part. The benefit for developers will be in the opportunity to use the Rodin Platform to make changes in Event-B models not only using the Event-B formal language, but also using the specification-based operators. One of the most important features of the created plugin is the renaming Event-B model elements (e.g., machines, events, variables), as this becomes a huge problem, when the model is quite big and the renamed object is used in different places. Renaming can also cause problems in proof obligations, one of the specific parts of formal languages, so it was important to keep this part valid and avoid errors caused by the renaming. The second big part of the plugin is merging two machines together, combing their variables, events, etc. This feature will allow two machines to combine their properties into one machine without creating new machine from scratch. Despite the fact that separate renaming (refactoring) and composition plugins already exist, our new plugin will combine these two features and will be scriptable using by specification-based operators. The developed plugin will simplify the verification of software in formal languages and save time during the development of reliable software systems [9].

1.3 Research question

The main question of the project is **RQ1**, *how can software verification be simplified for the developers?* This question is divided in two separate research questions, **RQ1.1** and **RQ1.2**.

- **RQ1**: how can software verification be simplified for the developers?
  - **RQ1.1**: how can developers, using different formal languages use the Rodin Platform without the necessity of studying the Event-B language?
  - **RQ1.2**: how can renaming and machine composition be implemented without any errors in the corresponding proof obligations?

These questions describe the main idea of the current project. Understanding of what is done in this field and studying the existing plugins and projects will be the first step in answering these questions. The existing plugins may help in designing the new one in a simpler and more efficient way. With a firm understanding of the work done in this research field, the work that should be done will become more precise. The resulting developed plugin will provide developers with short and easy to use tool which will allow them to use Event-B models without spending long time studying and understanding all the Rodin Platform features.

1.4 Solution

To solve the above stated questions, the plugin for the Rodin Platform was designed and developed. This plugin will simplify the development of reliable software systems and help developers to start using the Rodin Platform simply and without huge effort. The plugin will use the existing Event-B models and manipulate their underlying representation. The plugin supports input validation and indicates if some problems can occur during the execution of the operations (renaming or composition [11]). The plugin is developed in Java and uses the Event-B abstract syntax tree in the syntax analysis part of the development. This approach allows user to make necessary changes without the necessity of parsing the XML-files, which Rodin uses to store the Event-B project. The plugin is addressed to developers who want to verify software and who are familiar with the theory of institutions and specification-based operators.

1.5 Structure of the thesis

The project is separated into six chapters:

Chapter 1: the current chapter, introduces the initial problem, describes the motivation of the current project, states the research questions and solution of the problem in brief. This part also describes the current problems of using the Rodin Platform.

Chapter 2: this chapter presents the background research, it names and describes in detail two plugins that are most closely related to the current project and shows their advantages and disadvantages. This information is used in the next chapter a comparison between this project and existing plugins.

Chapter 3: this part defines the key requirements for the project and describes the design and implementation phases of the development process.

Chapter 4: this chapter represents the evaluation the work and states if all the key requirements were met and includes the comparison with existing plugins, mentioned in chapter 2.

Chapter 5: this part describes the case study.

Chapter 6: the last chapter of the thesis contains conclusions, where the value of the work is described and the future possible work is outlined.

Chapter 2

Background and related work

Formal reasoning is a very ‘strict’ type of reasoning, and it helps to find answers and make decisions between the conflicting sentences, ideas or opinions of different people. Formal reasoning is based on a certain form of arguments, which are declared to be true. Those arguments which contradict these arguments, become false accordingly. The conclusion, which is based on true statements, is supposed to be true as well. Formal reasoning usually reasons in terms of formal logic, not simple words. There are several languages, based on formal reasoning, in existence. One of the most popular languages based on formal reasoning is Event-B. It was developed by Jean-Raymond Abrial (France) and is based on the B language [16]. The main difference between B and Event-B is that Event-B has simpler notation, it is easy to learn and use and it has more features. It is used in many industrial projects and allows users to create systems and verify them.

2.1 Event-B

Event-B is a notation for formal modelling based around an abstract machine notation [6]. It allows the user to verify difficult real-life tasks. There are some examples of using verification in daily life: smart-grid modelling and railway interlocking models. Verification of systems is used to ensure the safety of people or to avoid costs caused by the improper operation of the system. The main advantage of using Event-B is that all development errors in the model can be easily found since in incomplete and inaccurate models some proofs cannot be done.

2.1.1 Contexts and machines

Event-B models consist of two main parts: contexts and machines. The context shows all static parts of the model, while the machine represents the dynamic parts of the model. These two main parts allow the user to create efficient models and to describe the behavior of the system. The key feature of Event-B is that the primary model can be really simple, but with the help of refinements, it can be gradually improved and become sufficiently complicated. The term “refinement” applies to the dynamic parts of the model, so-called machines. One of the most famous introductory example of Event-B modelling is the “Controlling Cars on a Bridge” model [3]. It describes a set of traffic lights for cars crossing the bridge from the mainland to an island and vice versa. The first model developed for this study case was really simple, it had island and bridge joined together and only two colors in the traffic lights: red and green, while the final model had not only traffic lights, but also car censors and all three parts of the case study – island, bridge and mainland. This example shows the idea of the refinement – gradual improvement of the model using refinements.

2.1.2 Events

The main part of machines in Event-B is the event. At the beginning of development there is just one event in each created machine – the INITIALISATION event. This event is used for initializing any variables shared between the events. No one model can work properly without this event. When developing the final model, different events can be created to describe the model. Each event should describe one action in real life. In the given example “Controlling Cars on a Bridge” there are different events describing “A car is leaving the mainland and entering the Island-Bridge”, “A car leaving the Island-Bridge and re-entering the mainland”, etc. The more precise a model becomes, the more events it usually includes. Events can either have no guards, or they can be also simple and guarded (keyword where) or they can be parameterized and guarded (keywords **any** and **where**) [19].

2.2 Theory of institutions and SBOs

Despite the fact that Event-B is a quite popular language and is used in industry, it has a great disadvantage - it doesn’t have well-developed modularization constructs and it is not easy to combine specifications in Event-B with those written in other formalisms [8]. Modularization constructs are the basis of the general theory of institutions. What is an institution? The concept of institution is introduced to formalize the informal notion of “logical system” [24]. Institutions enable abstracting away from syntactic and semantic detail when working on language structure “in-the-large”; for example, language features can be defined for building large structures from smaller ones, possibly involving parameters, without a commitment to any particular logical system. This applies to both specification languages and programming languages. Institutions also have applications to such areas as database theory and the semantics of artificial and natural languages [19]. A specification is the main modelling unit in an institution, but specification language is not a programming language, this is a collection of sentences about programs [19]. For a (pure) logical programming language, the specification is also a program [19].

The key concepts in the theory of institutions are:

- A specification is the main modelling unit in an institution.

In terms of Event-B, a specification is referred to as a component: i.e. it is the description for either a machine or a context. In the theory of institutions, a specification consists of a signature along with a set of sentences over that signature.

- The signature of a specification is the set of names used in that specification.

For an Event-B machine this is the set of (global) variables and event names. For an Event-B context this is the set of constant and set names. We don't worry about the names of invariants, guards and axioms, since these are just labels for information and can't be seen by other specifications.

- The sentences in a specification are just the predicates that define things.

In Event-B machines these are the invariants, guards and actions; in Event-B contexts they are the axioms.

2.3 Existing Event-B plugins

The Rodin Platform, which supports the Event-B language, is a simple and easy-to-use tool, but its' functionality is limited. This limitation doesn't allow users to manipulate Event-B elements and use all the features of this language. Developers from all over the world try to make the use of this tool as simple as is possible. They provide plugins [18] which allow user not only to edit the text of Event-B program (Camille editor), but also to create visual representations of the models (UML-B plugin), animate proofs (ProB animator), rename model elements and make compositions of several models. These plugins allow developers to create very precise and smart models and discover new features of the Event-B language. More details about two plugins closely related to the plugin described in this thesis are given in the following subsections.

2.3.1 Refactoring framework

The initial work of the Refactoring framework was done by Sonja Holl [17]. The author identified the problems related to refactoring in formal specification languages like B (and Event-B) due to the presence of proofs. The refactoring of such languages would need to be very good in order to avoid breaking proofs. This plugin was created to provide a possibility of renaming elements of the models written in Event-B. Users can rename not only machines and contexts, but also variables, variants, invariants, events, etc. via this plugin. To do so, the user should right click on the element he wants to rename and choose the new name. This simplicity allows the user not to worry about the proof obligations becoming inconsistent, because the plugin works very carefully and renames elements in such a way that proofs don't break.

The plugin has several updates, the latest version is 1.3.0 and based on Rodin 3.0.x. During the dozens of updates, the plugin became powerful tool with good functionality and user-friendly interface. It includes not only renaming for main parts of the Event-B model such as machines and contexts, but also small parts of these parts, such as variables, invariants, events, constants, axioms, etc. It also allows users to keep the proofs valid during the renaming. During its execution, the plugin operates with three trees: a dependency tree to match all the dependencies between the renaming object and other objects in model (such as variables in invariants or theorems), an abstract syntax tree of the Event-B language to get access to all of the elements of the model, and a proof tree to make changes in proof obligations without breaking them. Even though the plugin's main functionality is simple renaming, it allows users not to waste their time on creating new elements with other names and deleting unnecessary elements which can cause the crash of the whole system. Figures 2.1-2.4 contain screenshots of the user interface of this plugin showing the main flow of the interaction between user and the system. Figure 2.1 represents the plugin call, caused by right-clicking on the renaming element, and Figure 2.2 shows the modal window asking user to input the new name. After the valid input of the new name, the plugin checks if any problems can appear during the renaming (Figure 2.3) and then starts renaming (Figure 2.4).

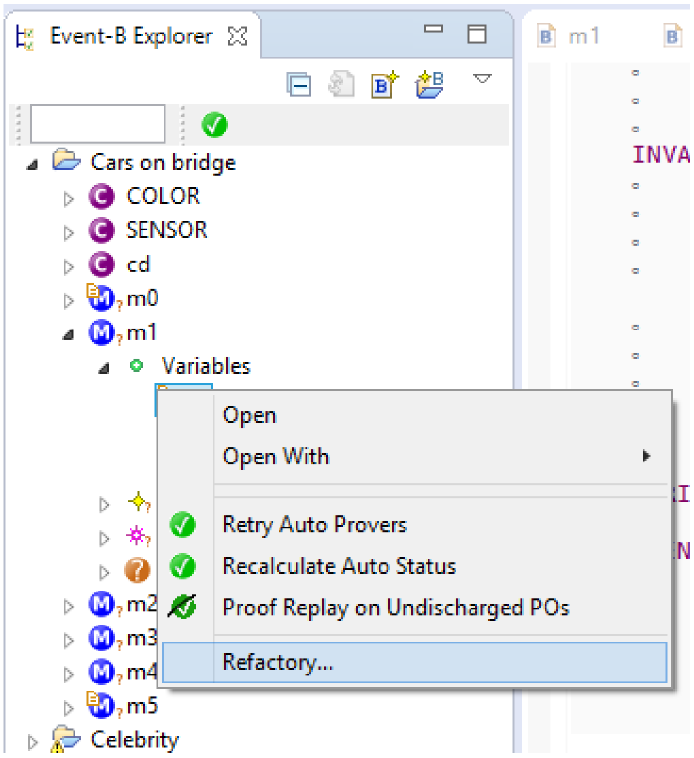
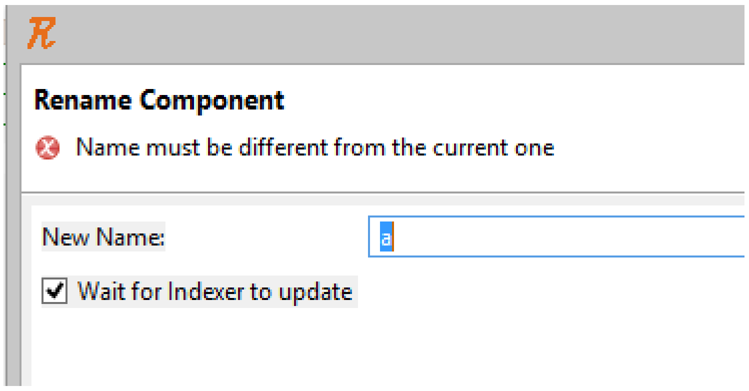


Fig. 2.1 Refactoring (renaming) plugin call



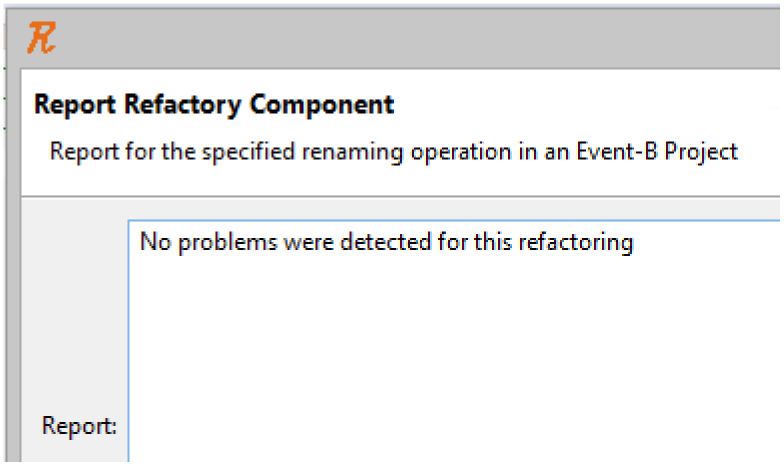
Rename Component

Name must be different from the current one

New Name: a

[✓] Wait for Indexer to update

Fig. 2.2 New name input

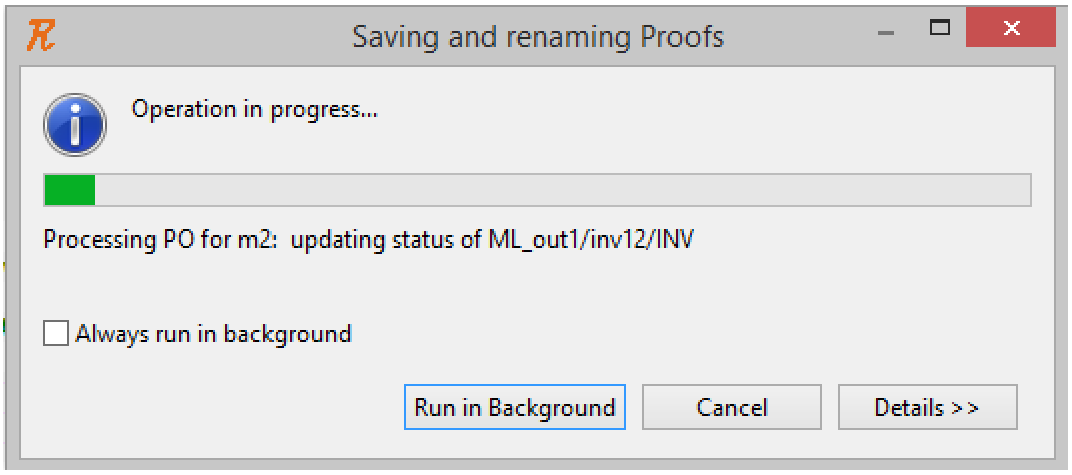


Report Refactory Component

Report for the specified renaming operation in an Event-B Project

Report: No problems were detected for this refactoring

Fig. 2.3 Problem report


Saving and renaming Proofs

Operation in progress...

Processing PO for m2: updating status of ML_out1/inv12/INV

Always run in background

Fig. 2.4 Renaming in progress

2.3.2 Feature composition plugin

A feature composition plugin was developed by Ali Gondal (University of Southampton) and compatible with Rodin 2.0. This plugin allows user to build a composition model of the input models. The new model is also an Event-B model and is saved in Rodin database. This smart plugin highlights potential conflicts when joining models such as declarations of the same events or variables in both models. It also allows the user to resolve conflicting situations by removing the repeating/redundant information in different models. The composition editor also provides option for merging events [12].

The composition of several models created by this plugin allows the user to obtain a model with the necessary properties without long manual development. The modal window of the feature composition plugin shows all available elements such as variables, invariants, events, etc. which can be used in the final composition model. The user can select/deselect these elements depending on planned idea of the final model. Besides allowing for the possibility of making a composition, the plugin allows the user to merge two or more events by creating a new event. However, the plugin doesn't allow the user to compose variants and theorems. The tool is also capable of composing features at different refinement levels. The composite feature is a typical Event-B model and is automatically checked by the Rodin static checker for any errors [10]. This plugin is a prototype project of the feature composition tool and the developers claim it will be improved in future versions. It was created because the feature composition tools that existed before didn't use all of the capabilities of the Event-B language and only provided user with small number of features. The developers claim that new versions of this plugin will be able to deal with proof obligations and create compositions based on existing proof obligations.

2.4 Summary

In this chapter we talked about Event-B language features, the Rodin Platform and its plugins which can be used to manipulate Event-B models. Two existing plugins were described to show key points of the renaming and composition of Event-B models. Some basics of the institutional theory were given to represent the advantages of using SBOs while operating with different formal languages. As the main problem of Event-B is its' 'non-standard' syntax, the next chapter represents the solution which is based on implementing the Event-B editor which supports SBOs and has a functionality of two plugins described above.

Chapter 3

Solution

3.1 Requirements specification

The requirements for the generated plugin are based on the research questions, introduced in chapter 1 and existing projects, which were mentioned in chapter 2. There are 5 requirements and these are described below.

R1 – The plugin should be easy to use for all the users of Rodin Platform

Despite the fact that all verification languages are based on the same principles and use the same logical laws, the syntax of these languages can vary significantly. The user of the plugin should have some experience of working with Event-B and be able to manipulate its elements. The plugin will help any user to rename any element of the existing model without difficult and time-consuming manipulations. Even if the user hasn't been working with the Rodin Platform for a long time, he will be able to call the plugin and write commands, based on specification-based operators. If the Rodin Platform user is more advanced, he will be able to learn specification-based operators and combine Event-B specifications with specifications from other languages easily.

R2 – The plugin should allow the user to rename Event-B elements and compose machines

The main features of the plugin are renaming of Event-B elements and composing of machines. As there are existing plugins with the same functionality, but working separately, the goal of this plugin development was to combine features of these two plugins into one plugin and to make unusual way of communication with the system. The implementation of the specification-based operators in the Rodin Platform will help to make the Event-B language more 'standard' and will allow the user to combine specifications in Event-B with specifications from other formal languages. This standardization will allow developers using different formal languages to work on one project simultaneously.

R3 – The plugin should not allow the user to make changes if the input is incorrect

The main idea of this part is to prevent program execution if the user input is incorrect. As Event-B provides the proofs of the system, incorrect data input can make these proofs inconsistent. The plugin should not only be user-friendly and simplify model development, but it should also keep the system stable and not allow the execution of operations which can make proof obligations inconsistent. Despite the fact that the Rodin core catches some exceptions, the plugin should now allow the user to overcome Rodin limitations and make harmful changes in the working model.

R4 – The plugin should give the user information about any incorrect data input

This part is very important in software development. The feedback from the program shows the user any possible mistake he could have made. There are several common mistakes that could be identified by the plugin. First of all, the user can write an element name incorrectly and in this case the plugin displays the message that the element with the given name doesn't exist. Another way of making a mistake is to specify the same name for the renaming element as it already has or to not specify a new name at all. A third possible mistake is misspelling the key words or placing them in the wrong place in the command. As the Event-B language is case-sensitive, all the commands should be written in lower-case and the names of renaming elements should match the elements' name in the model. In any other case the plugin will not be able to identify the elements.

R5 – The plugin should manipulate Event-B models with the help of specification-based operators

As was mentioned above, the main disadvantage of the Event-B language is that it doesn't have well-developed modularization constructs and it is not easy to combine specifications in Event-B with those written in other formal languages. The main feature of this plugin is the support of the specification-based operators. This feature allows developers to not worry about the Event-B language structure. The plugin executes all the operations using the Event-B abstract syntax tree, so all the dependencies between elements are considered and in case of valid input, no one machine will break during the renaming. Within the project, not all of the specification-based operators were implemented in the plugin, only part of them, but even this part could show the advantages of the chosen approach.

After the renaming, elements in the tree will change their names automatically, without the necessity of refreshing the model tree. The one thing the user will need to do is to open the renamed element (or machine/context if the element is inside them) and save it manually. After saving, some errors can occur in the model tree, but after the full build of the workspace, all errors will disappear. If the user changes the name of the element which is used somewhere in the model (e.g. a variable can be used in invariants or events), after the renaming it will be changed in all occurrences. This is the main advantage of renaming with the help of this plugin. During the manual renaming of the elements, the user could forget to change the name somewhere and this will cause errors. Manual renaming is very time-consuming as well.

3.2 Overall Project Concept

To meet requirements described above, a new plugin has been developed. The user interface of this plugin is a multi-page editor. The multi-page editor allows the user not only to write commands into one tab, but also to get feedback from the plugin in another tab. Answering the research question **RQ1.1** (Section 1.3), the idea of the solution is to develop a tool which can manipulate Event-B elements with the help of SBOs. Any formal language developer, familiar with the theory of institutions, will be able to use this plugin to work on an Event-B model. The plugin allows the user to rename not only the machines and contexts of an Event-B model, but also their elements, such as carrier sets, axioms, events, invariants, etc. Another useful feature of the plugin is machine composition, which allows the user to combine variants, invariants and events of one machine with those of another machine.

SBOs can be applied to any formal specification. They are specification-independent and this is their main advantage. These operators are 'standard' and all operators in formal languages are based on them. To work with another language, the developer should find analogues of these 'standard' operators in another language and compare them with those in the language he works with, which seems to be difficult work. The main concept of this plugin is to make it possible to work on an Event-B model using basic SBOs such as **with** and **and**. These key words will allow the user to make changes into Event-B models without deep knowledge of the Event-B syntax. This project also will become a starting point for Event-B 'standardization'. The more standard the formal language is, the easier it becomes to work with it. The simple 'standard' syntax for this plugin requires from the user only the correct inputting of names of elements that one works with.

3.3 System implementation

This section is one of the biggest sections in this thesis. It describes the main phase of the development - making decisions about the implementation of the key features of the plugin. In the next few paragraphs some of the implementation decisions will be discussed. We will pay attention to the most important decisions which were made during the development, such as plugin type, the selection of SBOs, etc.

3.3.1 Short summary of the system implementation decisions

One of the most important phases in software development is a design phase. After the phase of analysis, when the developer examines existing solutions and find all their advantages and disadvantages, he should find his own approach of solving the problem or decide to use some existing solution. During the design phase of the current project, two existing plugins were examined and their advantages and disadvantages were discovered. Among the advantages of the refactoring (renaming) plugin we can name the using of three types of trees – a dependency tree, an abstract syntax tree and a proof tree. This is an ideal solution for the problem of renaming elements and saving the consistency of proof obligations. This solution was kept in this project's plugin with some rework. Despite the fact that second plugin's functionality meets the requirements of our work, the solution used in the existing plugin is quite unusual and complicated. This was the reason to make our own decision and design a brand new solution for composition problem.

As a result, the user is able to make changes in Event-B elements with the help of our plugin without the necessity of installing and searching installed plugins within the Rodin Platform. He also doesn't have to click several buttons and select/deselect necessary elements to be included into the machine composition. The functionality of two existing plugins was combined and implemented in multi-page editor plugin. Key words **copy**, **with** and **and** allow the user to manipulate Event-B elements and get a feedback from the plugin within one editor (in two different tabs). We chose these three key words, because they cover all the necessary functionality and play an important role in institution theory, being the two main SBOs. Each word is responsible for certain operations. The word **copy** is in charge of creating a copy of the existing machine, the word **with** is responsible for renaming both machines/contexts and their elements, and the word **and** helps the user to create a composition of two machines, having elements from both of them.

The access to Event-B elements is directed through the abstract syntax tree. This tree is the foundation of the EMF (Eclipse Modelling Framework) model [28], which includes all the elements of Event-B formal language. This approach allows the plugin to decrease the time of executing the code compared to making changes in the model through the simple XML-parsing of Event-B project files. The number of these files increases dramatically in big projects for industrial use. When using any application, the user expects an immediate response from the program and the back-end of the current plugin manages to handle the information in a reasonable amount of time. When manipulating elements using the AST, there is no need to worry about missing any action or guard where the element is used, which is important for the current problem related to proof obligations.

We implemented the front-end part of the plugin using Standard Widget Toolkit (SWT) – one of the most popular Java plugin APIs. The user interface of the plugin is quite simple, it is a multi-page editor with two tabs – one for entering commands and another for output results from the plugin. The editor supports three basic SBOs. The key word **copy** creates a new file containing the newly created machine, while two other words don't create new files, they only make changes in existing ones. With this design we can answer research question **RQ1.1** as having a possibility to use SBOs in this plugin to manipulate Event-B elements without deep knowledge of the Event-B language syntax. Our plugin uses a multi-page editor in order to have access to Event-B elements without using any buttons or checkboxes and it provides efficient interaction between the user and the Event-B model. The solution of using three types of trees, which was mentioned above, can help to answer the research question **RQ1.2**. When the proof tree is created, it becomes easy to change names of elements and create compositions without making proof obligations inconsistent. As the Rodin tool and plugin are written in Java, it is possible to run them on any operation system, having a JVM.

3.3.2 SBOs and CASL

During the background research on the basics of the institution theory and specification-based operators, the main features of the specification language CASL [22, 23] were examined. CASL, the Common Algebraic Specification Language, has been designed by CoFI, the Common Framework Initiative for algebraic specification and development. CASL is an expressive language for specifying requirements and design for conventional software [20]. As the foundation of the plugin functionality is specification-based operators and CASL is well-structured formal language, this section of the thesis is dedicated to the comparative analysis of those key words and operators of the CASL language that are used in the plugin. All examples used for the analysis are small and simple, but large and complex specifications are easily built out of simpler ones by means of (a small number of) specification-building operations [21]. Operators, used in this part belong to Structuring Specifications [21]. Combined together, they allow the user build complex and structured programs.

3.3.2.1 Union

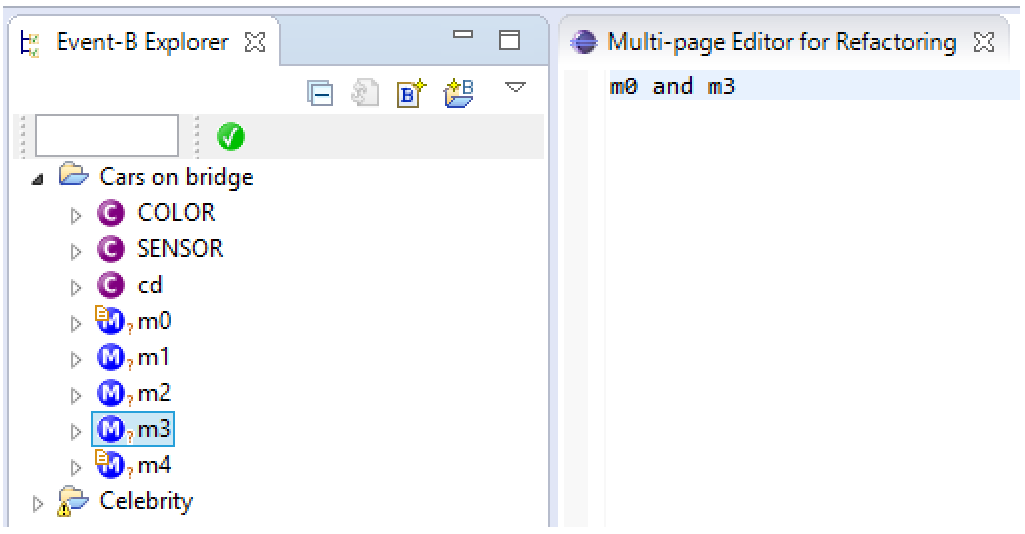
The first operator, which can be used to structure specifications is the union operator **and** [25]. Figure 3.1 represents simple piece of program, creating a union of two specifications – List_ Selectors and Generated_Set. Union is generally used to combine two self-contained specifications. Union of specifications is obviously associative and commutative [21].

```
spec   LIST_SET [sort Elem] =
       LIST_SELECTORS [sort Elem]
and    GENERATED_SET [sort Elem]
end
```

Fig. 3.1 CASL syntax for making union of two specifications

There is a principle 'same name, same thing' [21], existing in the CASL language. The main idea of this principle is that if two specifications have elements with the same name, they will not be duplicated. If these two elements have different content, one of these elements should be extended with the help of another CASL operator.

Let's have a look at the created plugin and syntax for manipulating Event-B elements. It is less formal than CASL syntax and doesn't need key words **spec** and **end**, but it has the same key word for creating a composition of two machines (Figure 3.2).



```
m0 and m3
```

Fig. 3.2 The plugin syntax for creating a composition of two machines

As can be seen from the above two screenshots, the newly created syntax for manipulating Event-B machines matches CASL syntax, which means that Event-B elements control becomes more formal and well-structured.

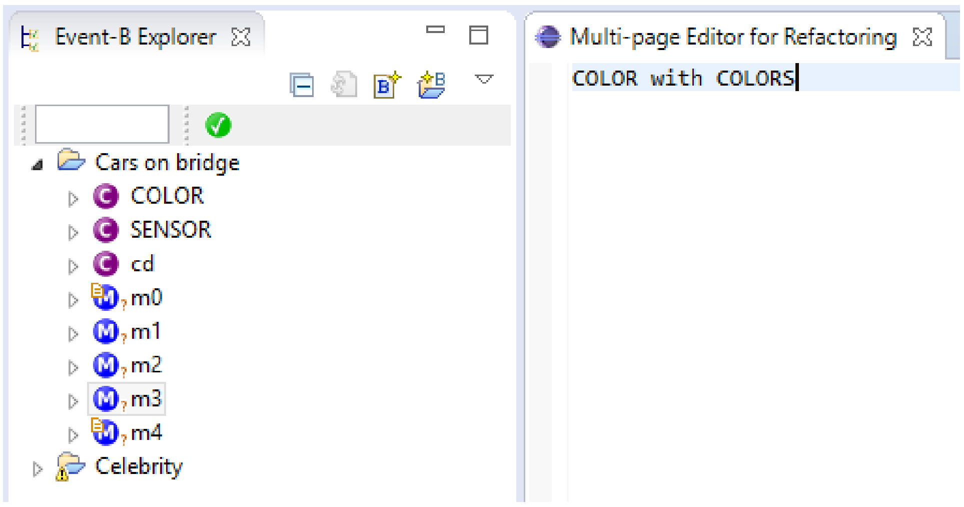
3.3.2.2 Renaming

Renaming may be used to avoid unintended name clashes, or to adjust names of sorts and change the notation for operations and predicates [21]. While the ‘same name, same thing’ principle is used in the union operation, it still can happen that during the combining of two specifications, this principle leads to unintended name clashes [21]. This can happen when two specifications with two elements with same name are intended to be combined. If both of these elements should remain in the final specification (they have the same name, but are not the same ‘thing’), then one of these elements should be renamed to avoid unintended name clashes. Figure 3.3 represents the syntax for renaming elements in the CASL language.

```
spec  STACK [sort Elem] =
      LIST_SELECTORS [sort Elem] with sort List ↦ Stack,
                                      ops  cons ↦ push_onto__,
                                           head ↦ top,
                                           tail ↦ pop
end
```

Fig. 3.3 CASL syntax for renaming elements of one specification

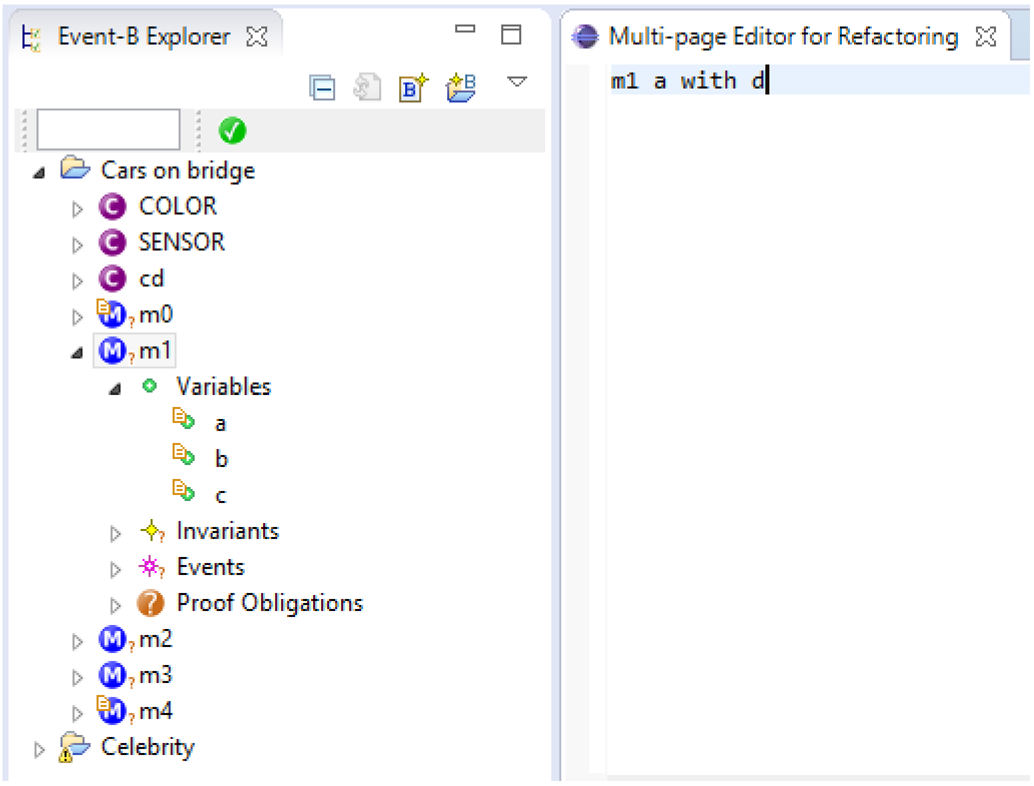
In this piece of code the key word **with** is responsible for renaming the element sort *List* into *Stack*, operation *cons* renamed into *push_onto_* and finally selectors *head* and *tail* are renamed into *top* and *pop*, respectively [21]. Let’s have a look at the renaming operation in the created plugin. It is implemented using the key word **with** as well. The syntax for this operator is similar to the key word **and** and requires the name of the existing machine/context and the new name, which should not be null or the same. Figure 3.4 represents the syntax of the plugin, which can be used to rename a context.



```
COLOR with COLORS
```

Fig. 3.4 The plugin syntax for renaming context COLOR

This syntax allows the user to rename both machines and contexts, but not internal elements. To rename Carrier Set/Constant/Axiom in a context or Variable/Invariant/Event in a machine, the syntax, represented on the Figure 3.5 should be used.



```
m1 a with d
```

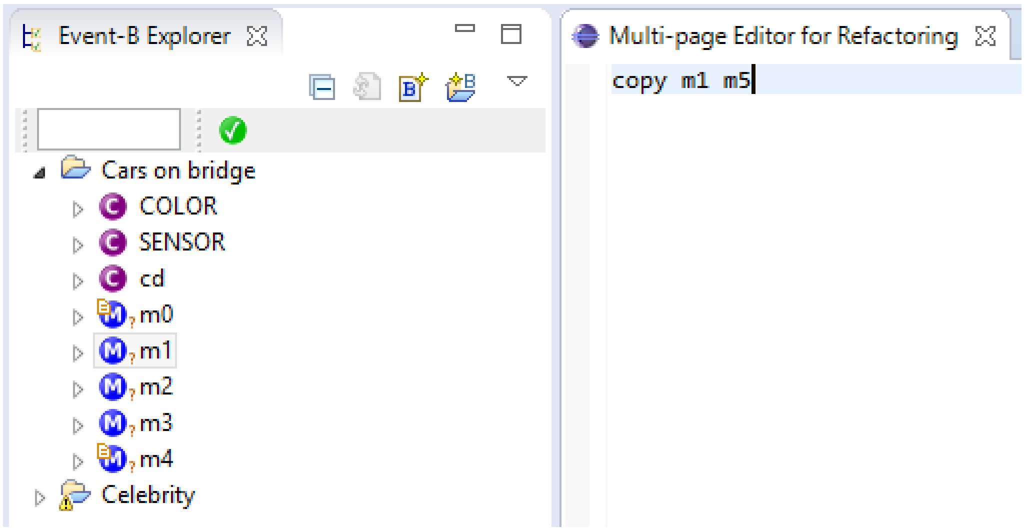
Fig. 3.5 The plugin syntax for renaming Variable *a* in machine m1

The syntax presented on the screenshot above can be used to rename Variable *a* in machine m1. It is clear that the syntax for renaming elements in CASL and renaming an element inside machines/contexts by the created plugin are very similar. Of course, some differences can be noticed as well. In CASL it is possible to rename several elements at the same time, while in our plugin only one element could be renamed during the execution of one operation. On the other hand, some benefits of our plugin include the possibility of changing machines and contexts, which is impossible to do in CASL.

3.3.2.3 Copying

This operator is not represented in CASL specification language, but in our opinion it is important to have a possibility of creating a copy of the existing machine, because sometimes the user needs to have several machines with similar properties, but having insignificant differences in the structure. The creating of such machines from scratch could be very complicated and time-consuming, considering the fact that models, used in industrial projects may have hundreds lines of code. The copying operation creates copy of the existing machine with another name. This machine could be used in future manipulations such as composition and renaming. Changes in the newly created machine will not affect the original one, which can be very useful in creating closely related machines.

As it was with the renaming operator, only one machine can be created during the execution of this operator. To create another machine, a new command should be written. This feature allows the user to avoid making silly mistakes in machine names and reduces the effort of searching for these mistakes. Figure 3.6 represents the syntax of copying a machine in the plugin.



```
copy m1 m5
```

Fig. 3.6 The plugin syntax for creating a copy of machine m1

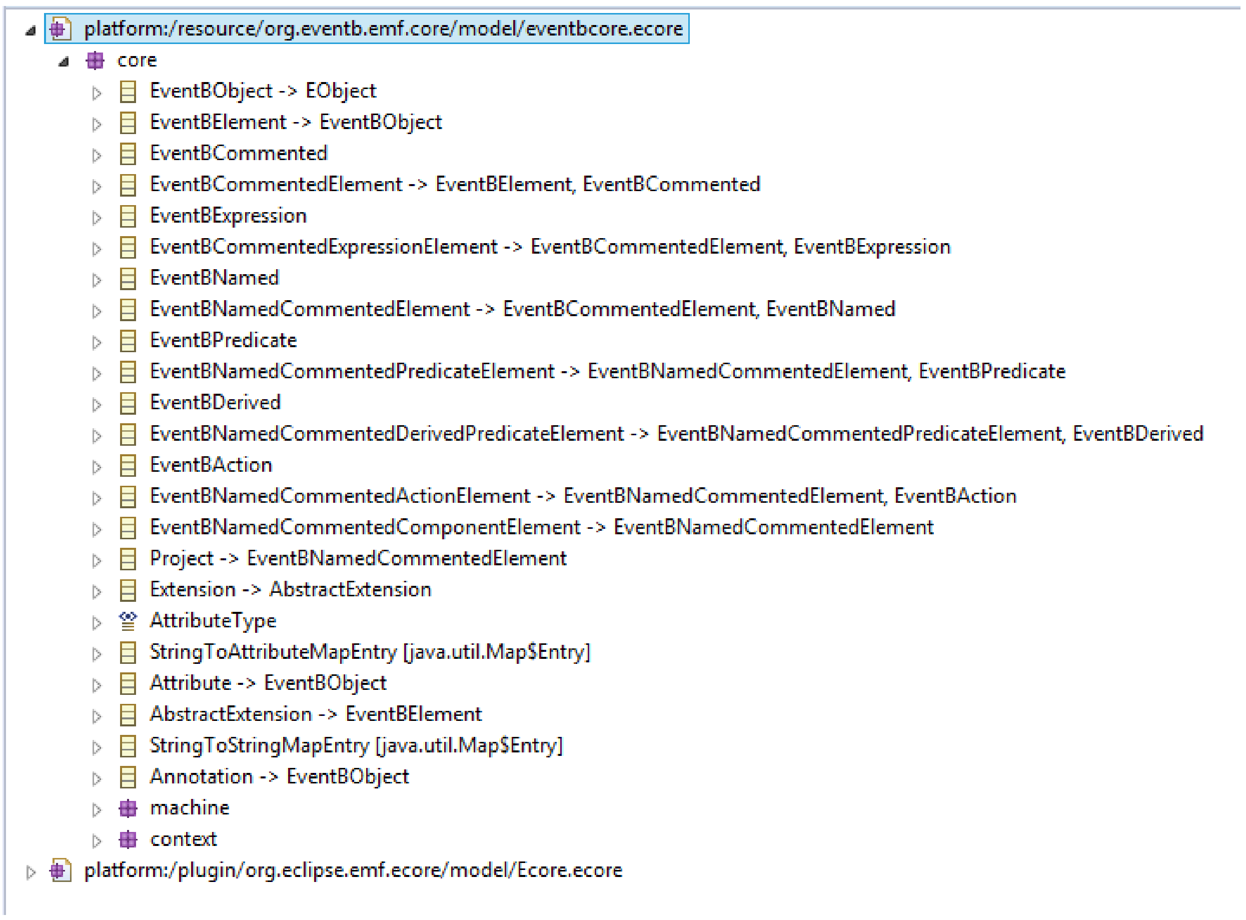
3.3.2.4 Summary

As can be seen from the overview of these three main operators, the first step of Event-B ‘standardization’ has been done. The new plugin allows the user to not only create a composition of two machines, rename elements of Event-B models and copy existing machines, but also makes it possible to manipulate these elements, having basic knowledge of CASL language. All the operators are quite simple, but powerful enough to make significant changes in Event-B models. The next step of this ‘standardization’ should be done by implementing the rest of operators such as then, hide (or reveal) and construction ‘local… within… ’. The implementation of all listed operators will fully cover structuring the specifications.

3.3.3 Back-end development

The back-end development includes all the functionality features of the plugin. During the plugin development, several techniques were used. The main framework used for creating the structure of Event-B is the Eclipse Modelling Framework (EMF). EMF is a modelling framework which allows the developer to create models of systems and describe relationships between components. Although the description of EMF and the main features of its models are close to UML diagrams, a model in EMF is less general and not quite as high level as the commonly accepted interpretation [27]. This framework unifies Java, XML and UML and its main benefit is the decreasing the costs of development. As Event-B is a well-structured formal language, its EMF model is logical and well-structured as well. This model was used to define all structural elements of Event-B and find dependencies and all relationships between Event-B components.

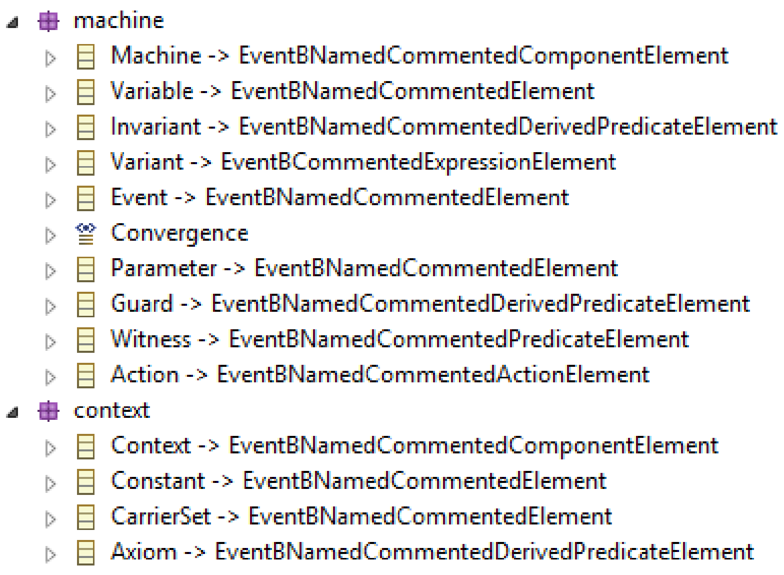
The following screenshot (Figure 3.7) represents Event-B .ecore model built with the help of EMF.



```
platform:/resource/org.eventb.emf.core/model/eventbcore.ecore
  core
    EventBObject -> EObject
    EventBElement -> EventBObject
    EventBCommented
    EventBCommentedElement -> EventBElement, EventBCommented
    EventBExpression
    EventBCommentedExpressionElement -> EventBCommentedElement, EventBExpression
    EventBNamed
    EventBNamedCommentedElement -> EventBCommentedElement, EventBNamed
    EventBPredicate
    EventBNamedCommentedPredicateElement -> EventBNamedCommentedElement, EventBPredicate
    EventBDerived
    EventBNamedCommentedDerivedPredicateElement -> EventBNamedCommentedPredicateElement, EventBDerived
    EventBAction
    EventBNamedCommentedActionElement -> EventBNamedCommentedElement, EventBAction
    EventBNamedCommentedComponentElement -> EventBNamedCommentedElement
    Project -> EventBNamedCommentedElement
    Extension -> AbstractExtension
    AttributeType
    StringToAttributeMapEntry [java.util.Map$Entry]
    Attribute -> EventBObject
    AbstractExtension -> EventBElement
    StringToStringMapEntry [java.util.Map$Entry]
    Annotation -> EventBObject
    machine
    context
platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore
```

Fig. 3.7 Event-B EMF model

As can be seen from the model, there are not so many special classes, but all of them define Event-B components. The structure of Event-B components is represented in Figure 3.8.



```
machine
  Machine -> EventBNamedCommentedComponentElement
  Variable -> EventBNamedCommentedElement
  Invariant -> EventBNamedCommentedDerivedPredicateElement
  Variant -> EventBCommentedExpressionElement
  Event -> EventBNamedCommentedElement
  Convergence
  Parameter -> EventBNamedCommentedElement
  Guard -> EventBNamedCommentedDerivedPredicateElement
  Witness -> EventBNamedCommentedPredicateElement
  Action -> EventBNamedCommentedActionElement
context
  Context -> EventBNamedCommentedComponentElement
  Constant -> EventBNamedCommentedElement
  CarrierSet -> EventBNamedCommentedElement
  Axiom -> EventBNamedCommentedDerivedPredicateElement
```

Fig. 3.8 Event-B main components

Besides the possibility of creating models of systems, EMF allows the user to generate source code from the model. This feature is very useful when the developer intends to create his own programming language, as it creates all the necessary classes and relationships between them. Because of this bi-directionality, EMF has successfully bridged the gap between modelers and Java programmers [27]. The created EMF model lets us not only fully understand the structure of Event-B, but also helps to build the strategy for manipulating Event-B components. This was very helpful due to the fact that Event-B stores its data in XML files and it is not clear at the beginning how to make changes to components or add something new into Event-B model.

One of the backend design approaches used in the project was the breaking of the text into tokens. The text entered into the editor, was split into separate words and these words were checked by the plugin. If they matched the key words defined by plugin, certain operations were performed, otherwise a warning was shown.

3.3.4 Front-end development

As the plugin is one of the means of connecting the user and the system internal structure, its interface should be clear and easy to use. It should not contain redundant panels or tabs; the names of the tabs should match the tab functionality, etc. For creating the plugin, the Standard Widget Toolkit (SWT) library was used. This library contains all visual elements of the applications such as buttons, labels, modal windows and so on. As was mentioned above, the foundation of the created plugin is a multi-page editor. A newly created file from this editor is stored in the project folder and has .mpe extension. It has two tabs containing information about the current operation. The user enters certain commands into the first tab and when he switches to the second tab, the method activates and the feedback from the plugin appears on the second tab. The user doesn’t have to press anything else to execute operations.

If no errors occur during the program execution, the plugin will display the successful result on the screen and make any necessary changes in the model. The user can see changes performed by the plugin immediately after the execution. Renamed or composed machines will appear in the Event-B explorer tree. The copying of a machine takes more time and thus a Progress bar in the Rodin Platform will indicate how much of the work has been done. The renaming opertion is a very important capacity of this plugin. As this operation involves changes inside the model, some errors can temporarily appear in other machines or contexts or even in proof obligations. To fix this problem, the user should open the error machine/context and simply save it. After saving, all errors should disappear. These errors are related to the changes inside models, but they don’t affect the correctness of the model and don’t make proof obligations inconsistent. As proof obligations are the most important part of the Event-B language, the ability of keeping them correct makes this plugin important in the work with manipulating Event-B elements. As can be seen from the next section, all the commands for this multi-page editor are simple and easy to remember and use.

How does the Refactoring plugin work?

To run the plugin, the user should choose it in the main menu of the Rodin Platform. To do so press File – New – Other… – Refactory Wizards – Multi-page Editor File. The modal window should appear. It contains two fields available for changing. The first field is designed for the name of Container. This name can be selected only from existing projects in the current workspace. The message above this field shows the user if the entered name is valid. The second field is used for the name of the creating file. It has value 'Commands.mpe' by the default. This name will appear on the first tab of the editor. If the user entered correct names for both fields, the button Finish will become active. After clicking this button, the main window of the editor will appear. The editor is case-sensitive, so all the key words should be entered in a lower case, while all the names of model elements should match existing elements in the current model. All the commands should be entered by the user in the first tab. The results of the execution will appear in the second tab called "*Results*".

To copy the existing machine, the user should type "**copy** [machine_1] [machine_2]" command, where copy is a key word, machine_1 is a name of existing machine and machine_2 is a name of newly created machine. There are some restrictions put on this command. Firstly, the machine_1 name should obligatorily be the name of existing machine. The input of the wrong name will cause an error "*The file [machine_1] doesn't exist*". Secondly, the machine_2 name should not exist. The input of the existing machine will cause an error "*The file [machine_2] already exists, choose another name*". If everything entered correctly, the message "*The machine [machine_1] exists. The file [machine_2] doesn't exist, start copying*".

To rename the existing machine/context, the user should enter the following command: "[machine_1/context_1] **with** [machine_2/context_2]". As with the **copy** key word, all restrictions remain the same. If the [machine_1/context_1] entered incorrectly, an error "*No such machine or context exist*" will occur. If the [machine_2/context_2] is null or equal to the existing name, the following error will occur: "*Select another name. The name cannot be null or be equal to existing name*". If everything entered correctly, the message "Context [machine_1/context_1] exists. Renaming of the context [machine_2/context_2] is done!" will appear.

To rename components of the machine/context enter command "[machine/context] [component_1] **with** [component_2]", where [machine/context] is the name of parent machine/context of the element, [component_1] is existing component intended to be renamed and [component_2] is a new name of the component. Restrictions for this variation of renaming remain the same as for the renaming of machines/contexts.

To create a composition of two machines, the user should first copy the first machine of the composition, as all the changes that will be made into it. After this step is done, the following command should be entered: "[machine_1] **and** [machine_2]", where [machine_1] and [machine_2] are the existing machines which are intended to be merged. This command will copy into [machine_1] all missing variants, invariants and events, existing in [machine_2]. The mandatory condition for this command is the existence of both machines.

3.4 Dependability attributes

Before we start to evaluate dependability of the project, let's have a look at the definition of dependability to learn what dependability actually is. According to the dictionary [29], dependability is an ability to perform as and when required. In other words, dependability shows the user's trust in the system. It represents the coincidence between the system behavior and user's expectations. Dependability has several attributes, such as system's **availability**, **reliability**, and its **maintainability**, and **maintenance support performance**, and, in some cases, other characteristics such as **durability**, **safety** and **security** [29]. In this thesis attention will be paid to the first three attributes: availability, reliability and maintainability of the system.

3.4.1 Availability attribute

Availability is an ability to be in a state to perform as required [29]. According to this definition, the system should be in an active state during the whole process of its use. This means that execution of one particular operation should take a satisfactory amount of time. Among all the operations performed by the plugin, only one operation takes a significant amount of time. This operation is copying. Copying is time-consuming operation, because new machine is created from scratch, so it takes some time to copy all variables, invariants and events, build the syntax tree, and make sure that the new machine is available and ready to use. The more elements the original machine has, the more time it will take to recreate it. Despite the fact that copying takes some time, it is possible to continue working on the rest of the elements in the model. They will be available for changing even during the copying operation. However, the manual creation of the machine will definitely take much more time from the user, and could cause problems due to missing any elements or creating new proof obligations, which takes significant amount of time.

One more point about availability within different operating systems can be noticed. As the Rodin Platform is Java-based open source project, it could be run on all operating systems having a JVM. Respectively, the plugin's jar-file could be put in the Plugins folder in the directory where Rodin was installed. The plugin works fine with Windows and Linux operating systems, which makes it possible to work on Event-B projects from different machines, having the same amount of operations and functionality. Of course, the user can rely only on himself during the execution, the system doesn't allow the user to avoid all possible errors, but some errors from the user side are caught and shown in the Results tab of the multi-page editor.

To conclude, the availability attribute of the plugin was shown for two points of view – within the system itself and within several operating systems. According to the information above, the plugin may be considered as reliable software, which provides the user with access to internal elements of an Event-B model all the time.

3.4.2 Reliability attribute

Reliability is the ability to perform as required, without failure, for a given time interval, under given conditions [29]. The correspondence to this attribute could be traced according to the requirements **R2**, **R3** and **R4**, mentioned in Section 3.1. The requirement **R2** describes the first factor of reliability – *correctness*. It sounds like "The plugin should allow the user to rename Event-B elements and compose machines" and describes the main functionality of the system. The correctness of the system is provided by execution of the system according to the specification. The requirement **R3** sounds like "The plugin should not allow the user to make changes if the input is incorrect". This requirement confirms the second factor of the reliability – *robustness*. This factor helps to prevent damages if the use of the system is outside the specification. The plugin shows a warning message about the incorrect input, but doesn't make changes in the system. The third requirement **R4**, which states that "The plugin should give the user information about incorrect data input", provides the third factor of the reliability – *security*. This requirement prevents not only damage caused by deliberate use of the system outside the specification, but also unintended incorrect use of the system by the user.

Incorrect data input

Despite the fact that all factors of reliability were considered in system requirements, the situation when the user enters incorrect data still can occur. It is almost impossible to avoid incorrect data input from the user, as the names of all elements in Event-B model are the responsibility of the user. There is no spelling check in the multi-page editor, so the user should be very careful while entering new or existing names. Even in the situation when the user entered incorrect new name and noticed it after the execution, he can always delete incorrect elements from the model and execute the command with correct data again. The high speed of the operations' execution allows the user to call them quite frequently without any delays.

One of the most frequent examples of incorrect input is the violation of the language syntax rules. The created plugin supports a scriptable language, which has certain rules required for calling proper methods. These rules are defined for each operation and the misspelling even of one word or incorrect order of words in command can cause errors in the execution. As was mentioned in the previous sections, the operation of renaming can be written in two variations: for machines/contexts and for their elements. It is very important to keep the correct order of the words in this operation as these two variations of the renaming call two different types of renaming. The same situation holds for two other operations. They can only be called if the command is written properly. Even wrong case of commands causes errors in the execution. The user is responsible for correct call of functions and name specifying, which influence on correct execution of the whole system and changes in the structure of Event-B model.

3.4.3 Maintainability attribute

Maintainability is an ability to be retained in, or restored to a state to perform as required, under given conditions of use and maintenance [29]. This is a characteristic, which can be evaluated after some time of the use of software. Technologies nowadays change so fast, so it is quite difficult to say how much will requirements to software change after certain period of time. Some of them could become outdated and will be required to change, some new requirements could appear. One of the possible solutions of this problem could be an open source platform.

Open source

Software development nowadays is completely different compare to software development even 10 years ago. A lot of open source projects appear and all developments interested in particular field of development can take part in it. This allows developers from all over the world to work on important problems for humanity and maintain projects [30], which haven't any sponsorship from big companies. We believe that open source projects increased the maintainability of software. This project will be maintained by developers, interested in formal languages standardization and searching for the current development decisions. The plugin is stored in jar-file and could be easily used by any developer. It can be changed or completed by other operations and developed till the full functional scriptable language. Both the developers and the project will have benefits of the open source status of this projects.

Chapter 4

Overall project evaluation

This chapter is dedicated to the evaluation of the contribution given by this project. This can be done by reviewing requirements mentioned in Section 3.1. A comparison of the current project with existing plugins with the same functionality will also be conducted, and the benefits of the created plugin will be discussed.

4.1 Requirements

The requirements were presented in Section 3.1 and introduced for our project. These requirements were built on the basis of research questions and existing plugins responsible for operations on Event-B model elements. The fulfillment of these requirements should help us to reach a goal of 'standardization' of Event-B and to improve the existing functionality of different plugins. We will repeat all requirements in this Section to have quick access to them.

R1 – The plugin should be easy to use for all the users of Rodin Platform

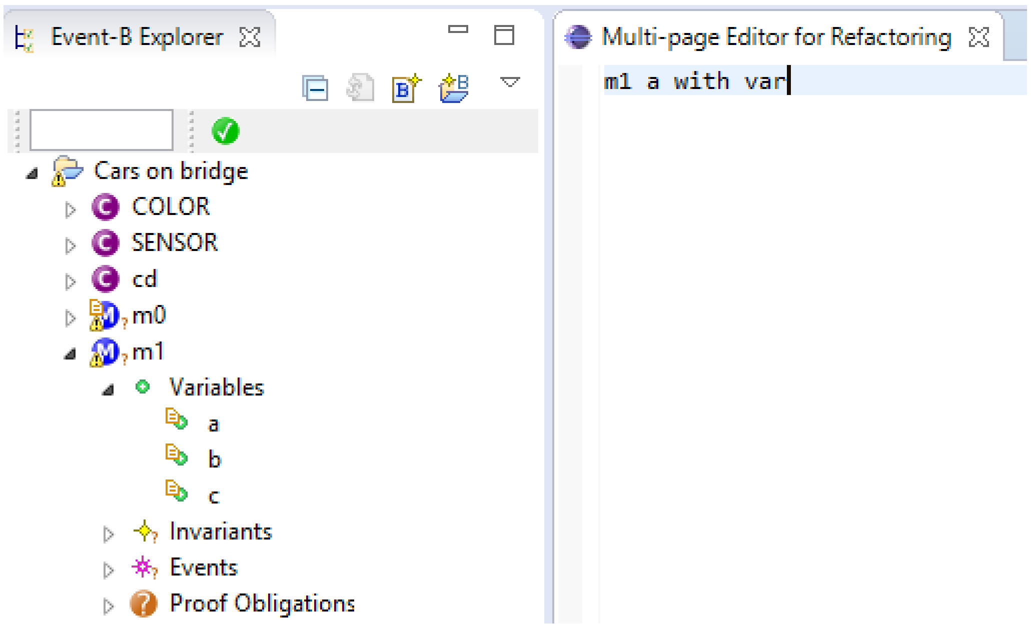
R2 – The plugin should allow the user to rename Event-B elements and compose machines

R3 – The plugin should not allow the user to make changes if the input is incorrect

R4 – The plugin should give the user information about incorrect data input

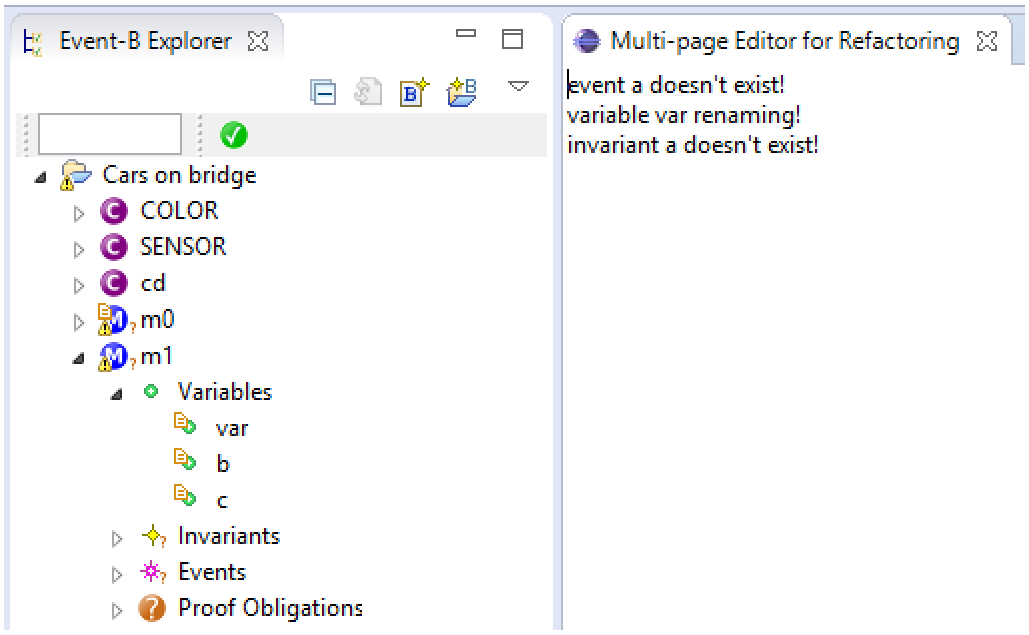
R5 – The plugin should manipulate Event-B models with the help of specification-based operators

The first requirement **R1** made our project different from other existing plugins. It doesn't have lots of buttons, input fields and checkboxes, but its functionality combines functionalities of two existing plugins and adds something new and unused before. As this project is based on an open source Platform, the plugin is available for all users of the Rodin Platform, working on different operating systems. The second requirement **R2** was concerned with the functionality of the created plugin and was not an easy requirement, because there are two different plugins in existence, each of them responsible for just one operation. It was required that we consider how to implement this functionality in that way so the execution will not be time-consuming. Figures 4.1-4.4 can support the fulfillment of this requirement. Figures 4.1 and 4.2 represent the renaming of the variable before and after execution.



```
m1 a with var
```

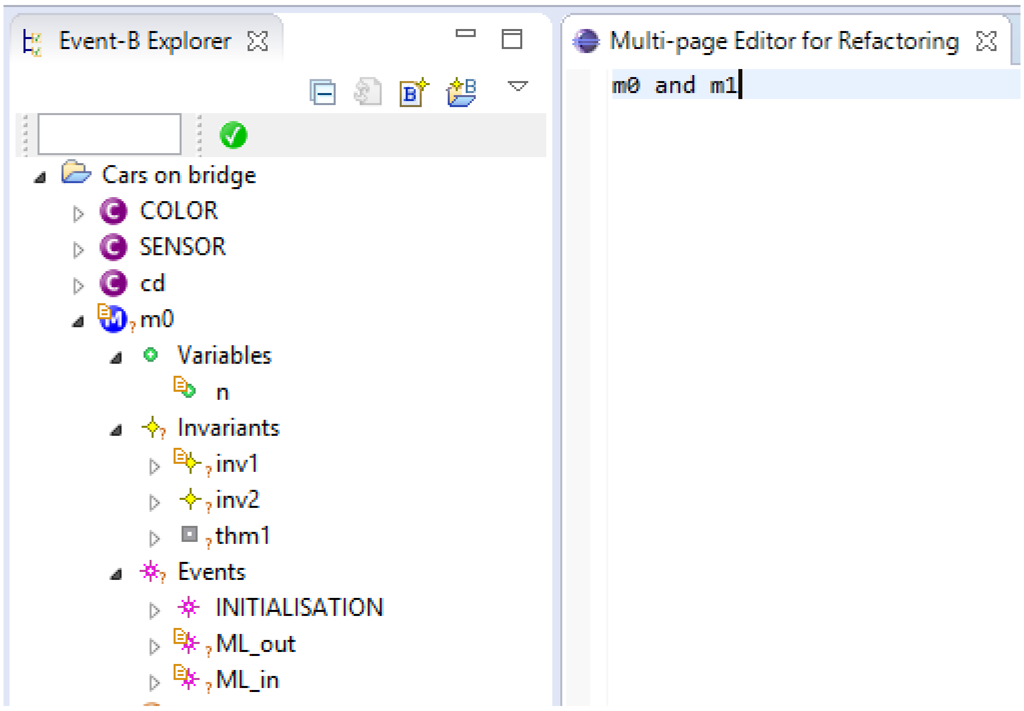
Fig. 4.1 Renaming of the variable a before the execution



```
event a doesn't exist!
variable var renaming!
invariant a doesn't exist!
```

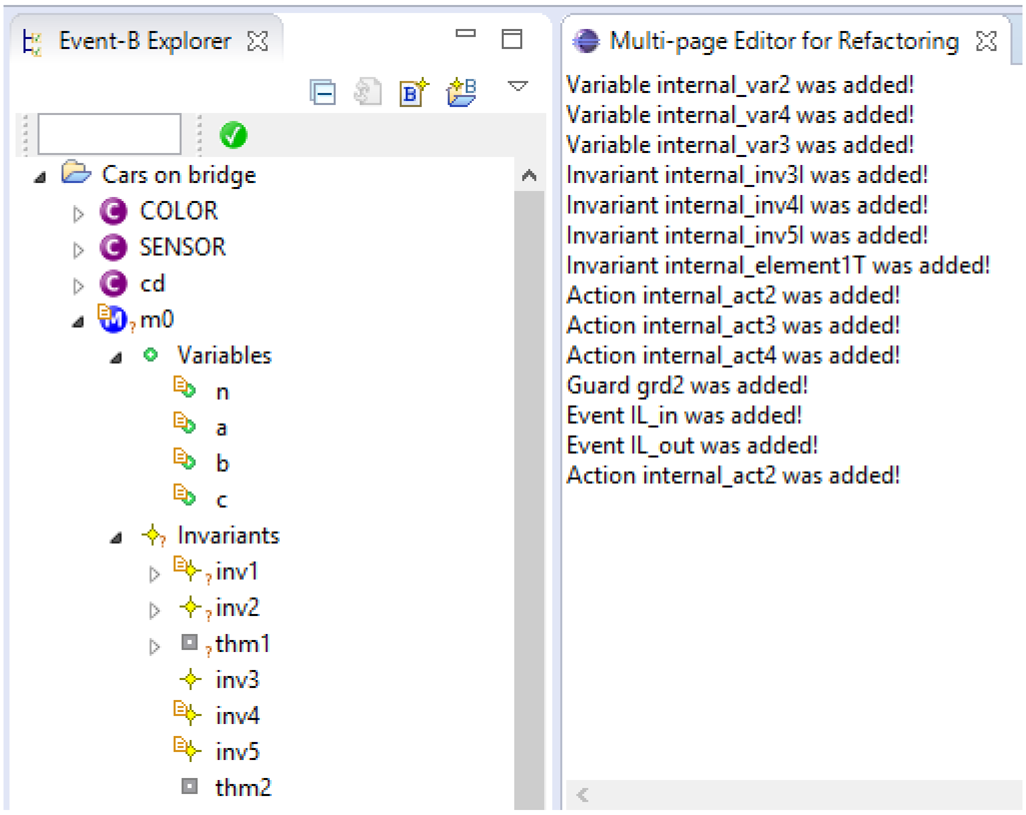
Fig. 4.2 Renaming of the variable a after the execution

Figures 4.3 and 4.4 represent creating composition of machines m0 and m1.



```
m0 and m1
```

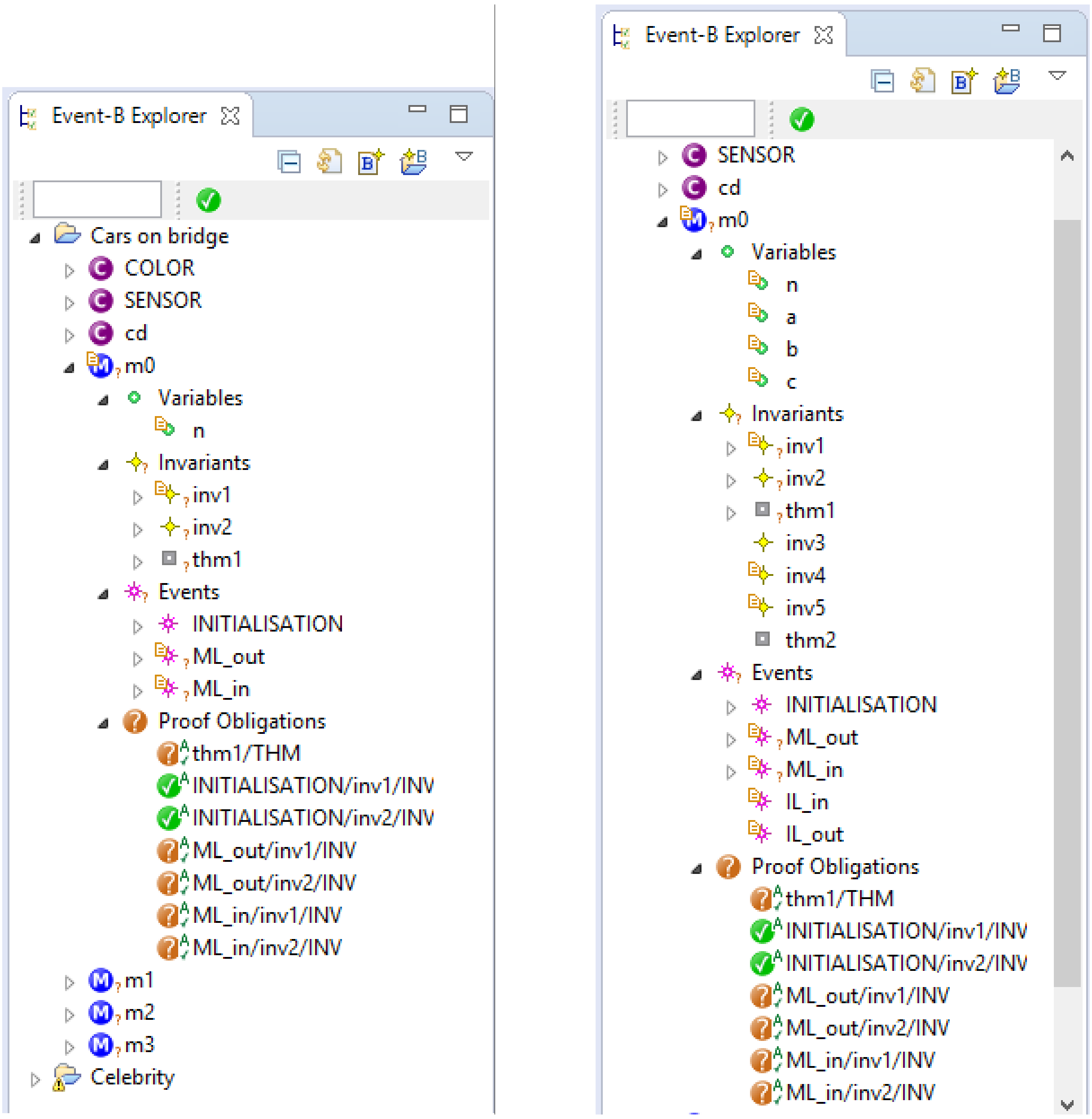
Fig. 4.3 Machine m0 before the creating of composition



```
Variable internal_var2 was added!
Variable internal_var4 was added!
Variable internal_var3 was added!
Invariant internal_inv3l was added!
Invariant internal_inv4l was added!
Invariant internal_inv5l was added!
Invariant internal_element1T was added!
Action internal_act2 was added!
Action internal_act3 was added!
Action internal_act4 was added!
Guard grd2 was added!
Event IL_in was added!
Event IL_out was added!
Action internal_act2 was added!
```

Fig. 4.4 Machine m0 after creating of composition

To confirm that the composition was created by the command in our plugin, screenshots of machine m0 before and after creating composition are attached (Figures 4.5 a and b).


```
Event-B Explorer
Cars on bridge
    COLOR
    SENSOR
    cd
    m0
        Variables
            n
        Invariants
            inv1
            inv2
            thm1
        Events
            INITIALISATION
            ML_out
            ML_in
        Proof Obligations
            thm1/THM
            INITIALISATION/inv1/INV
            INITIALISATION/inv2/INV
            ML_out/inv1/INV
            ML_out/inv2/INV
            ML_in/inv1/INV
            ML_in/inv2/INV
    m1
    m2
    m3
Celebrity
```

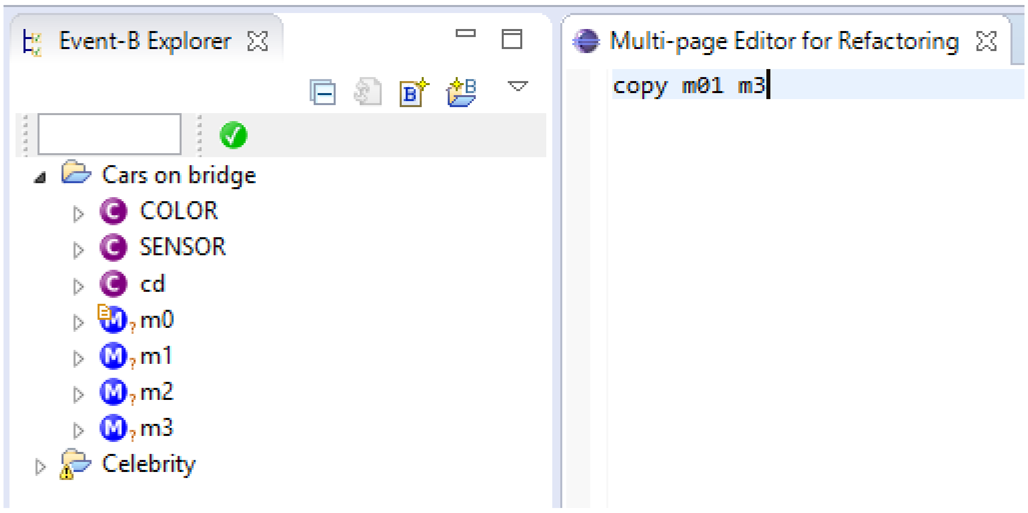
(a)

```
Event-B Explorer
SENSOR
cd
m0
    Variables
        n
        a
        b
        c
    Invariants
        inv1
        inv2
        thm1
        inv3
        inv4
        inv5
        thm2
    Events
        INITIALISATION
        ML_out
        ML_in
        IL_in
        IL_out
    Proof Obligations
        thm1/THM
        INITIALISATION/inv1/INV
        INITIALISATION/inv2/INV
        ML_out/inv1/INV
        ML_out/inv2/INV
        ML_in/inv1/INV
        ML_in/inv2/INV
```

(b)

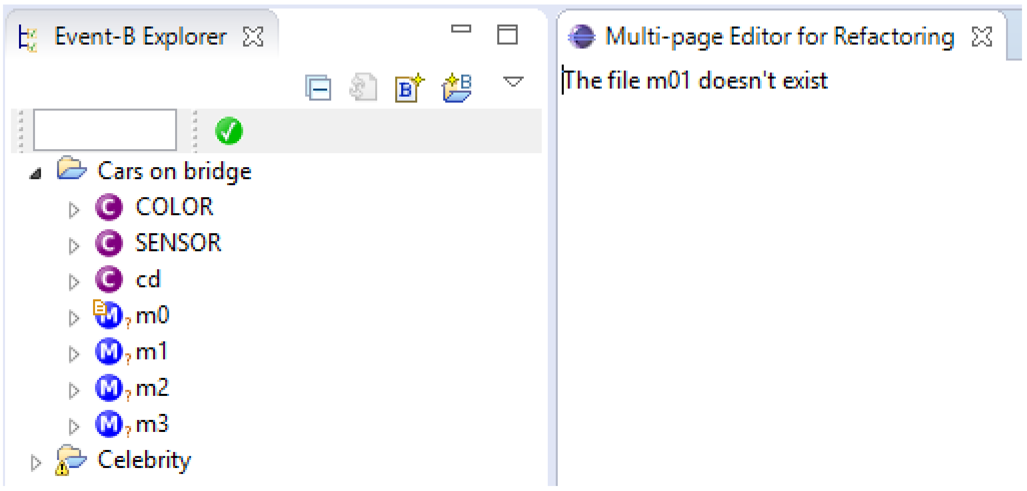
Fig. 4.5 Full elements tree of the machine m0 before the composition (a) and after (b) the composition

The third and fourth requirements **R3** and **R4** are related to each other and are responsible for checking the user's input and giving feedback. The requirement **R3** is quite obvious, but it was decided to distinguish it, because some programs may save changes or perform operations even if the user's input is incorrect. As Event-B is a formal language and requires a high quality and accuracy of storing objects, performing operations with syntax or other errors is unacceptable. Requirement **R4** was fulfilled by creating the second tab in multi-page editor and printing on the screen the result of the execution of commands, as entered by the user. This feature allows the user to discover errors early and to pay attention to problems in a piece of code. To support this requirement, Figures 4.6 and 4.7 are attached.



```
copy m01 m3
```

Fig. 4.6 Incorrect command, which should be not executed



```
The file m01 doesn't exist
```

Fig. 4.7 The feedback from the plugin on incorrect input

The requirement **R5** was the most important requirement in the whole work and its main idea was to make Event-B model changes inside components with the help of certain key words. Key words were chosen according to the existing well-structured formal language CASL and were integrated into the editor. All key words, implemented in the plugin, were presented on screenshots above (Figures 4.1, 4.3 and 4.6). As can be seen from this review, all requirements were fulfilled and the created plugin does all operations which it supposed to do.

4.2 Comparison to previous work

There were two existing plugins introduced in Sections 2.3.1 and 2.3.2: the Refactoring framework and Feature composition plugin. These two plugins support almost the same features as ours. Table 4.1 represents the comparison of the functionality of existing plugins and features and the improvements of our solution. The main disadvantage of the existing plugins is that they both perform only one particular objective, while the newly created plugin combines features of both of them. They interact with the user by modal windows. These modal windows contain buttons, input fields, checkboxes, etc. This allows the user to visualize the interaction between the user and the system and prevents the user from possible mistakes. Our solution differs from this solution. Our plugin doesn't have visual representation, it is more about writing commands in the editor and keeping the user focused on the correctness of commands.

The functionality of the Refactoring platform allows the user to change the name of the only one Event-B element at once. The functionality of the second plugin [14] is more powerful, but more complicated at the same time. It allows the user to create not only compositions of the machines, but also compositions of events. The result of the newly created composition is stored in a separate file and doesn't look like the simple Event-B machine or context. The composition created by our plugin stores in the first of composed machines. It appears in the Event-B explorer tree and has the same properties as a simple machine. Our plugin has one more feature and allows the user to copy the existing machine with the help of one command.

Table 4.1 A comparison with existing plugins

Features	Our plugin	Refactoring plugin	Feature composition plugin
Allows renaming components	✓	✓	
Performs only one operation per one execution	✓	✓	
Allows creating composition of machines	✓		✓
Allows creating composition of the event	✓		✓
Newly created composition stored as usual machine	✓		
Allows copying existing machine	✓		
Gives the user feedback about incorrect input	✓	✓	✓
Doesn't make changes if the input is incorrect	✓	✓	✓
Uses graphic user interface to interact with the user		✓	✓
Uses specification-based operators to call methods	✓		
Could be extended with other functionality	✓		✓

As can be seen from the comparison table, the functionality of our plugin coincides with the functionality of the two existing plugins. The main difference which gives the advantage of using our plugin is the extra feature of copying machines. The existing refactoring plugin doesn't offer the functionality of composing either machines or events, while the Feature composition plugin [13] cannot do the renaming of Event-B model components. Our plugin is designed in a way such that powerful functionality of the three main operators is hidden behind the simple handling. It is obvious, that all three of plugins interact with the user with the help of warnings or error messages. Nevertheless, the two existing plugins have been developed completely as a GUI with modal windows and visual elements on them, while our plugin is represented using a simple multi-page editor with two tabs and absence of other visual elements.

The multi-page editor can be used for writing commands, which is one more advantage of our plugin. These commands are SBOs and the most important feature of the plugin. They are not only commands, but also a means of transformation of Event-B into a well-structured formal language. As only three operators were implemented, there is a possibility of extending the functionality of our plugin. Some new features like extension (key word **then**), hiding (key word **hide**), etc. could be added. This possibility will be described more precisely in Chapter 6, which contains information about future work. The feature composition plugin could be extended as well [15], because it doesn't work for composing Variants and Theorems. One more problem about extending this plugin is that it is compatible only with Rodin 2.0, but not with later versions and this problem should be fixed first.

Our plugin makes it possible to make the Event-B language well-structured and easy to use for all developers working with formal languages. It combines features of two existing plugins and adds its own functionality. SBOs which were used as our scriptable language operators, are standard and will allow the user to manipulate Event-B elements in a very unusual way, which was not implemented by anybody else before [26]. This approach allows us to add originality in the Rodin Platform and to fill the gaps in existing functionality of the Rodin Platform plugins. By extending our plugin developers will reach the goal of Event-B 'standardization', which will become a huge contribution into the formal language development and popularization.

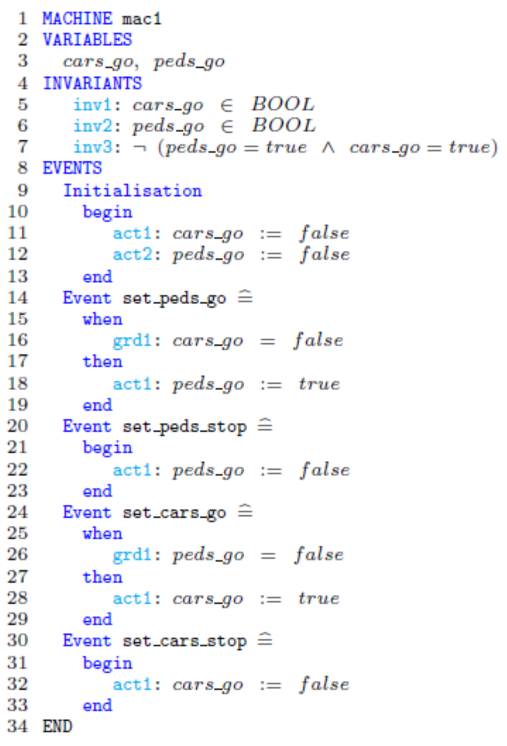
Chapter 5

Case study: traffic-light simulation

To perform the case study, a paper by Marie Farrell, PhD student of the Maynooth University, was used [8]. Marie is a final year PhD student and her research paper is dedicated to modularization of Event-B specifications. The modularization is based on specification-based operators, so the main idea of 'standardization' of Event-B is similar between our and Marie's work. The difference between our works is in the approaches. Marie is creating her own language, so called Modular Event-B, to be able to create her own models from scratch, while our idea was to make an editor which will be able to manipulate Event-B elements in existing models, but not create new ones. A case study of the most popular example in Event-B will be used to compare two solutions. This example is described in Abrial's book [3] and represents a traffic-light simulation. With this case-study we want to: 1. Demonstrate two different approaches of solving the same problem; 2. Determine advantages and disadvantages of the found solution; 3. Understand how the plugin can be improved.

5.1 Case study configuration

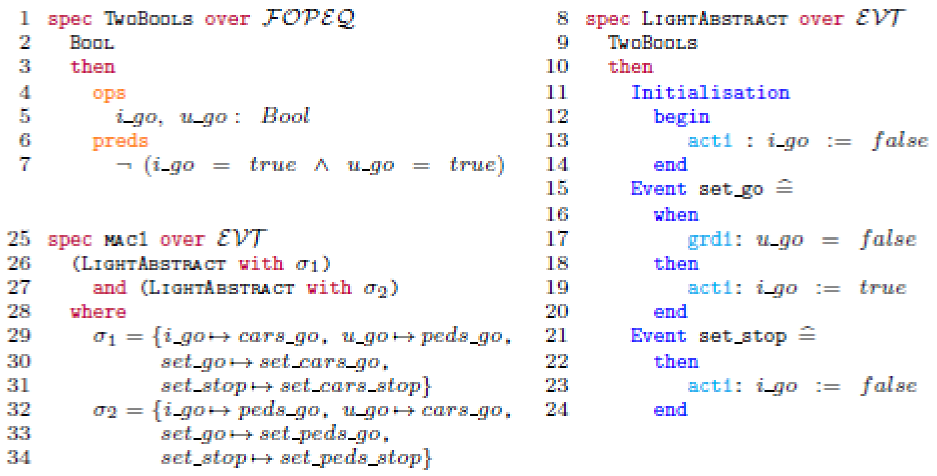
As was said above, to describe all operations, the simulator of traffic-lights will be used. For convenience, we will put the model source code below (Figure 5.1). This example's source code was taken from Marie's paper [8], because the whole modularization of Event-B in her paper has been done on the base of this example.



```
MACHINE mac1
VARIABLES
  cars_go, peds_go
INVARIANTS
    inv1: cars_go ∈ BOOL
    inv2: peds_go ∈ BOOL
    inv3: ¬ (peds_go = true ∧ cars_go = true)
EVENTS
  Initialisation
    begin
       act1: cars_go := false
       act2: peds_go := false
    end
  Event set_peds_go ≙
    when
       grd1: cars_go = false
    then
       act1: peds_go := true
    end
  Event set_peds_stop ≙
    begin
       act1: peds_go := false
    end
  Event set_cars_go ≙
    when
       grd1: peds_go = false
    then
       act1: cars_go := true
    end
  Event set_cars_stop ≙
    begin
       act1: cars_go := false
    end
END
```

Fig. 5.1 Event-B machine specification for a traffic system, with cars and pedestrians controlled by Boolean flags.

We used SBOs (Section 3.3.2) to build the set of operations which could be used to make some changes in Event-B models. The set of operations, available in the plugin, contains three items. These operations will be compared to Marie paper's analogues. The first key word used for renaming Event-B components is **with**. In our plugin there are two variations of using this key word. The first variation is simple one, used for renaming machines and contexts. The syntax and restrictions of using this operator were described in Section 3.3.4. For simplicity, we will repeat the syntax in this Chapter. For the simple version of the renaming operator it looks like: "[machine_1/context_1] **with** [machine_2/context_2]". Let's have a look at Marie's modularized machine mac1 (Figure 5.2).



```
1  spec TwoBools over FOPEQ
2    Bool
3    then
4      ops
5        i_go, u_go : Bool
6      preds
7        ¬ (i_go = true ∧ u_go = true)

25 spec mac1 over EVT
26   (LightAbstract with σ1)
27     and (LightAbstract with σ2)
28   where
29     σ1 = {i_go ↦ cars_go, u_go ↦ peds_go,
30            set_go ↦ set_cars_go,
31            set_stop ↦ set_cars_stop}
32     σ2 = {i_go ↦ peds_go, u_go ↦ cars_go,
33            set_go ↦ set_peds_go,
34            set_stop ↦ set_peds_stop}
```

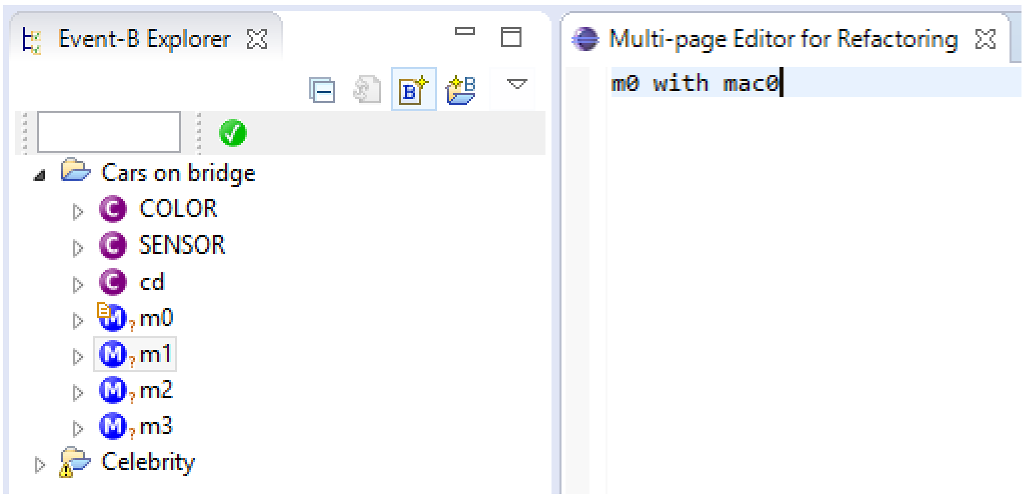
```
8  spec LightAbstract over EVT
9    TwoBools
10   then
11     Initialisation
12       begin
13         act1 : i_go := false
14       end
15     Event set_go ≙
16       when
17         grd1: u_go = false
18       then
19         act1: i_go := true
20       end
21     Event set_stop ≙
22       then
23         act1: i_go := false
24       end
```

Fig. 5.2 A modular presentation of the machine mac1 described in Marie Farrell's thesis

This piece of code represents a modularized version of the Event-B machine mac1. As can be seen from the screenshot, one machine mac1 was split into three parts. The first part (lines 1-7) represents the specification TwoBools, declaring two boolean variables constrained to have different values [8]. It is clear that this specification describes the INVARIANTS part of the original machine. The second part (lines 8-24) represents the specification LightAbstract, extending TwoBools and having just three events, while the original machine has five. This feature will be explained in the next part. The only constraint in this part states that a light can only be set to "go" if its opposite light is not [8]. The third part of the modular representation (lines 25-34) explains the reduced number of events in LightAbstract specification. As can be seen from lines 26 and 27, the specification LightAbstract is used twice in the final specifications. Lines 29-34 show all replacements of variables which were used in LightAbstract specification.

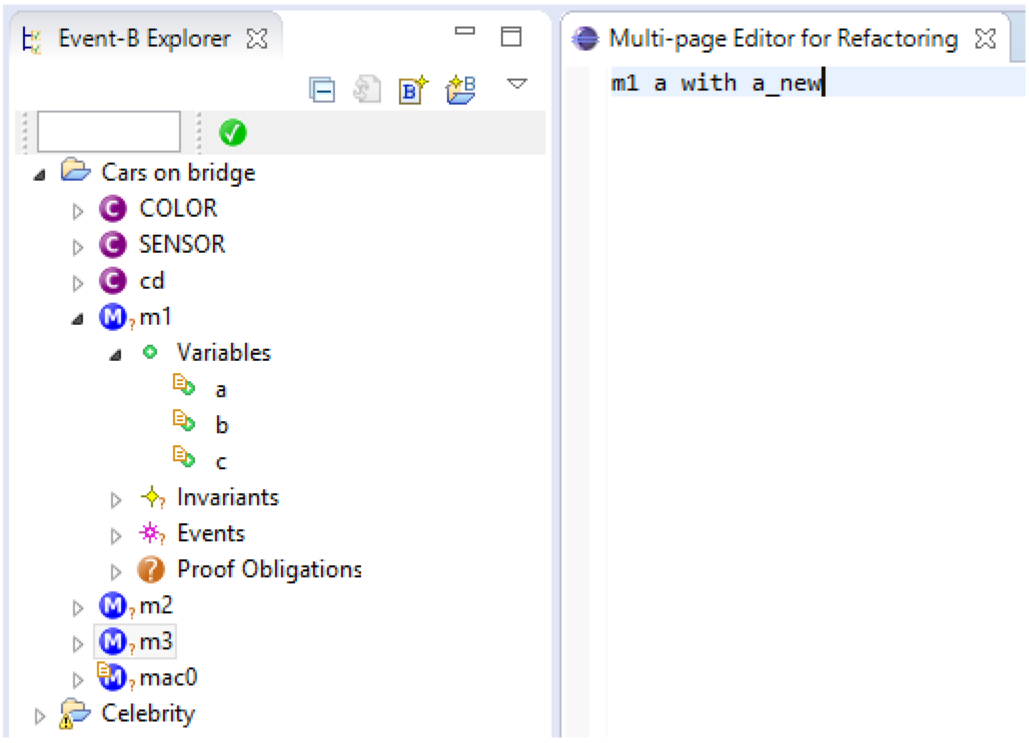
As can be seen from the screenshot (Figure 5.2) some key words were used by Marie as well. They are highlighted with red. Let's have a closer look at the key words **with**, **where** and **and**. As was explained before, the key word **with** was used to replace variables which were used in the specification LightAbstract with other variables. This replacement means simple renaming in this case. The key word where is an auxiliary command, which is used to define the correspondance between old and new variables. So these two key words together allow the user to rename several variables at once. It is clear from this example that the key words **with** and **where** are able to rename not only variables inside events, but also events themselves.

To compare these key words with ours, let's have a look at Figures 5.3 and 5.4. As can be seen from screenshots, only one operation can be executed at once, so only one element can be renamed. However, Marie's code can rename several elements including both events and variables inside events, which seems to be a better option for creating new models. The possibility of renaming several objects can be considered as a future improvement for our plugin.



```
m0 with mac0
```

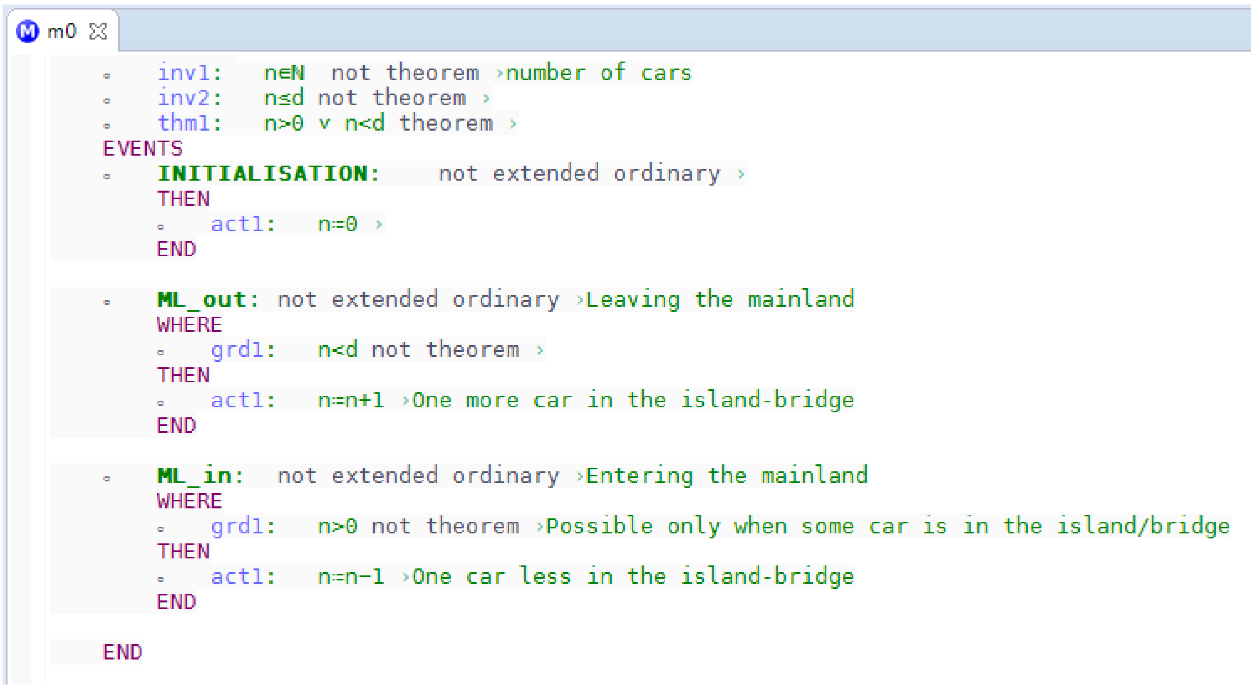
Fig. 5.3 The renaming of the machine m0



```
m1 a with a_new
```

Fig. 5.4 The renaming of the variable a in machine m0

The next operator, which should be examined, is operator **and**. In both our solutions this operator is responsible for merging events (in our case machines as well). Marie used this operator on the line 27 to merge two different versions of the specification LightAbstract. Our plugin allows the user to merge two machines and if there are two events in two different machines with the same name, the plugin will merge them too. Merged events contain all actions and guards from original events. If there are two actions or guards with the same name, they will not be duplicated. The execution of this operator by our plugin is shown in Figures 5.5-5.9. Figures 5.5 and 5.6 represent machines m0 and m1 before the merging. As can be seen from two these Figures, both machines contain INITIALISATION event, but the number and names of actions inside this event are different.



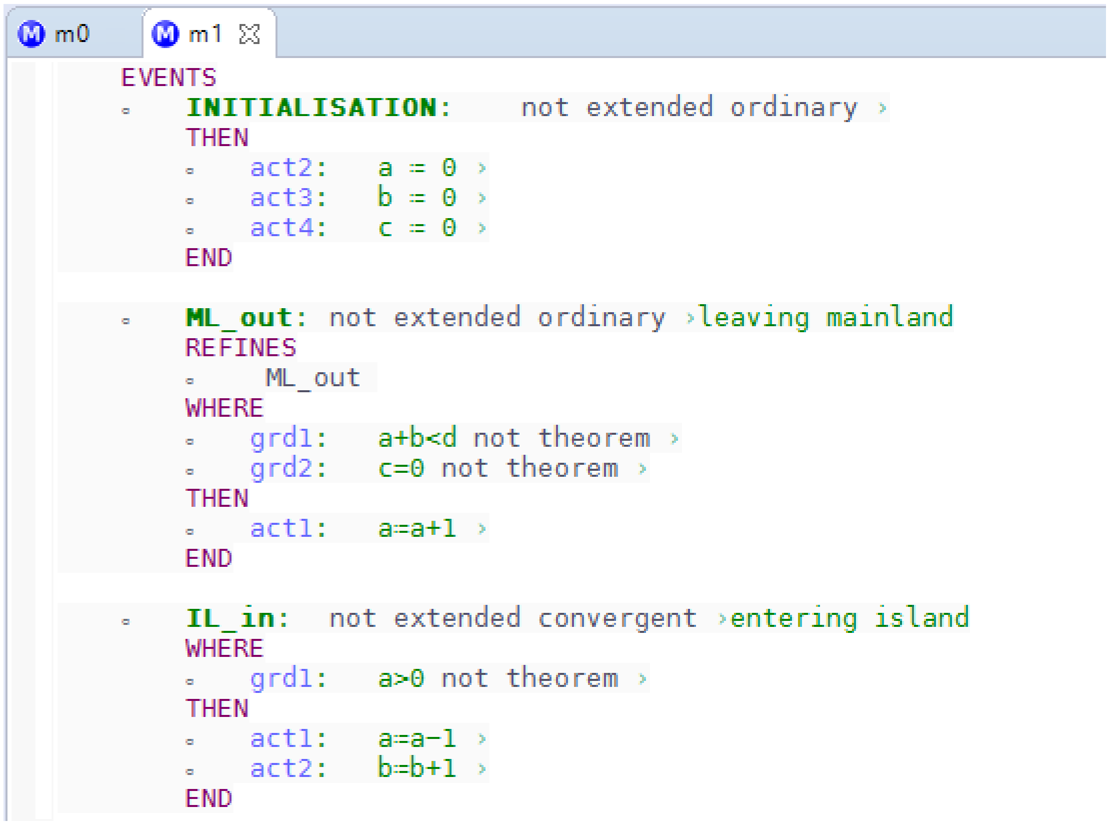
```
m0
    ◦ inv1:   n∈ℕ  not theorem ›number of cars
    ◦ inv2:   n≤d not theorem ›
    ◦ thm1:   n>0 ∨ n<d theorem ›
EVENTS
    ◦ INITIALISATION:    not extended ordinary ›
      THEN
      ◦ act1:   n≔0 ›
      END

    ◦ ML_out: not extended ordinary ›Leaving the mainland
      WHERE
      ◦ grd1:   n<d not theorem ›
      THEN
      ◦ act1:   n≔n+1 ›One more car in the island-bridge
      END

    ◦ ML_in:  not extended ordinary ›Entering the mainland
      WHERE
      ◦ grd1:   n>0 not theorem ›Possible only when some car is in the island/bridge
      THEN
      ◦ act1:   n≔n−1 ›One car less in the island-bridge
      END

END
```

Fig. 5.5 Machine m0 before the merging



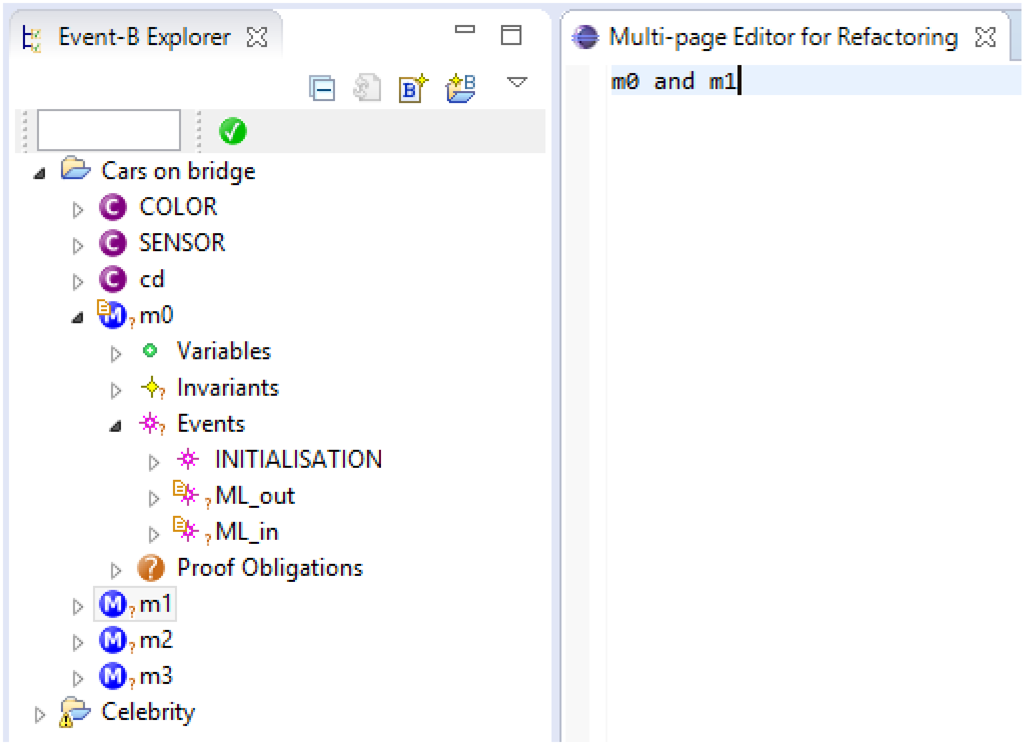
```
m0   m1
EVENTS
  INITIALISATION:     not extended ordinary ›
  THEN
    act2:    a ≔ 0 ›
    act3:    b ≔ 0 ›
    act4:    c ≔ 0 ›
  END

  ML_out: not extended ordinary ›leaving mainland
  REFINES
       ML_out
  WHERE
    grd1:    a+b<d not theorem ›
    grd2:    c=0 not theorem ›
  THEN
    act1:    a≔a+1 ›
  END

  IL_in:  not extended convergent ›entering island
  WHERE
    grd1:    a>0 not theorem ›
  THEN
    act1:    a≔a−1 ›
    act2:    b≔b+1 ›
  END
```

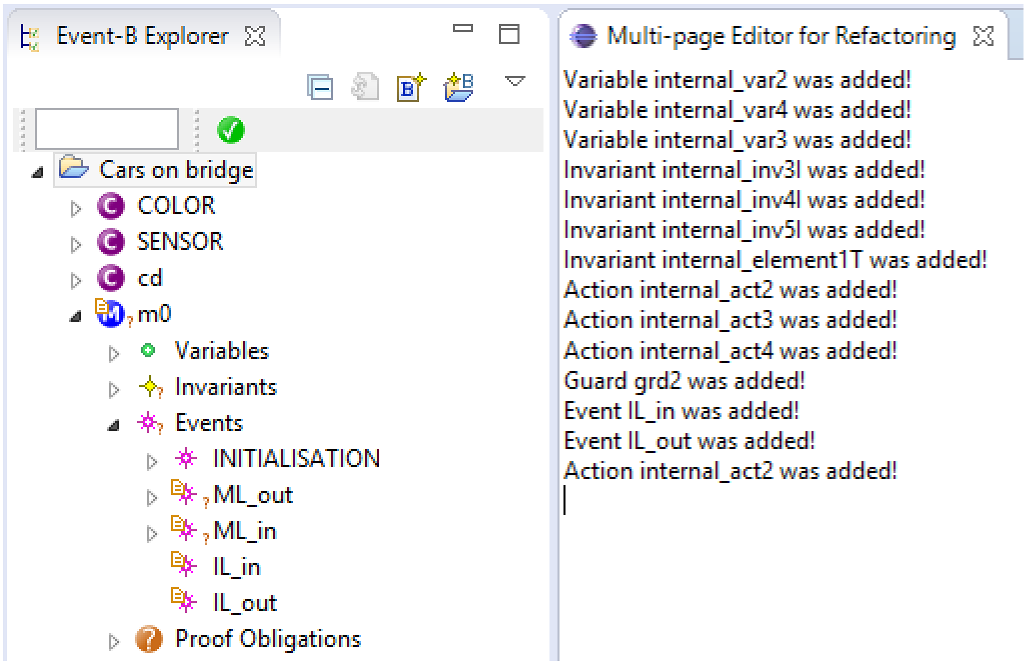
Fig. 5.6 Machine m1 before the merging

After the execution of the merging operation, the number of events in machine m0 will change, as well as the number of actions and guards (Figures 5.7-5.9).



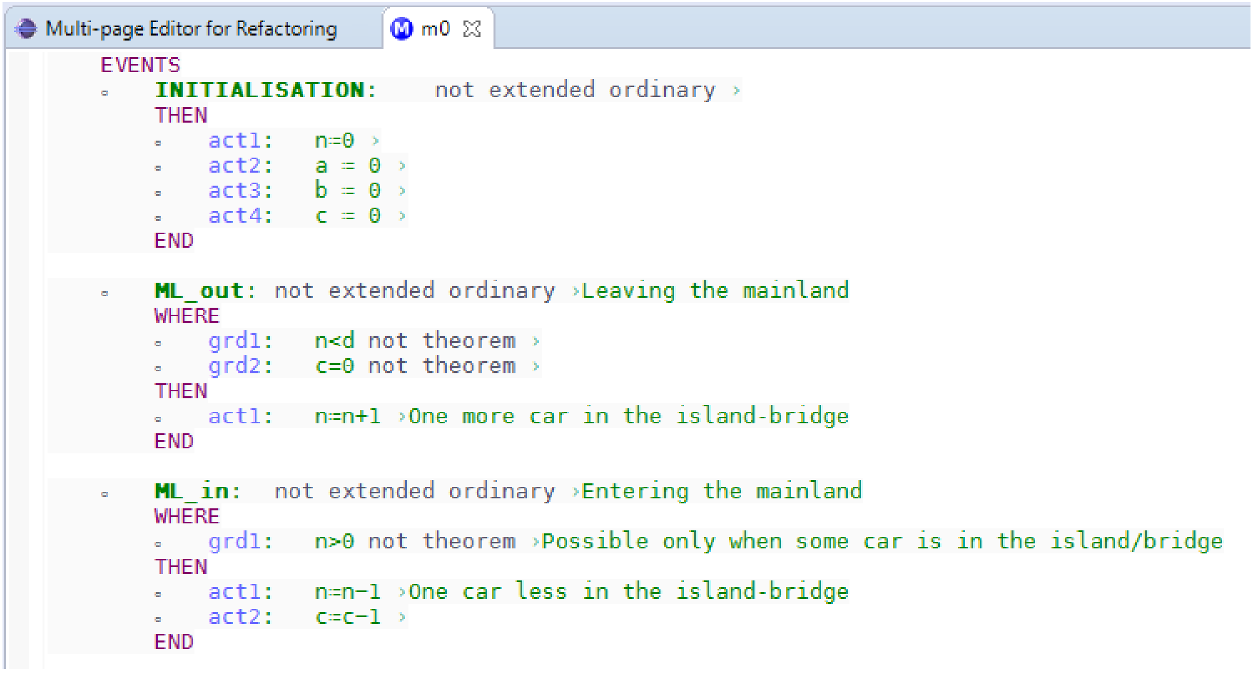
```
Event-B Explorer                    Multi-page Editor for Refactoring
                                    m0 and m1
Cars on bridge
  COLOR
  SENSOR
  cd
  m0
    Variables
    Invariants
    Events
      INITIALISATION
      ML_out
      ML_in
    Proof Obligations
  m1
  m2
  m3
Celebrity
```

Fig. 5.7 Machine m0 before the merging



```
Variable internal_var2 was added!
Variable internal_var4 was added!
Variable internal_var3 was added!
Invariant internal_inv3l was added!
Invariant internal_inv4l was added!
Invariant internal_inv5l was added!
Invariant internal_element1T was added!
Action internal_act2 was added!
Action internal_act3 was added!
Action internal_act4 was added!
Guard grd2 was added!
Event IL_in was added!
Event IL_out was added!
Action internal_act2 was added!
```

Fig. 5.8 Machine m0 after merging



```
EVENTS
    INITIALISATION:     not extended ordinary ›
    THEN
        act1:    n≔0 ›
        act2:    a ≔ 0 ›
        act3:    b ≔ 0 ›
        act4:    c ≔ 0 ›
    END

    ML_out:  not extended ordinary ›Leaving the mainland
    WHERE
        grd1:    n<d not theorem ›
        grd2:    c=0 not theorem ›
    THEN
        act1:    n≔n+1 ›One more car in the island-bridge
    END

    ML_in:   not extended ordinary ›Entering the mainland
    WHERE
        grd1:    n>0 not theorem ›Possible only when some car is in the island/bridge
    THEN
        act1:    n≔n−1 ›One car less in the island-bridge
        act2:    c≔c−1 ›
    END
```

Fig. 5.9 Events in machine m0 after merging

There are two key words that make difference between two of our solutions. Marie has key word **then**, which allows the user to extend the existing specification. Our plugin doesn't have this functionality at the moment, it is intended future work. There is a key word **copy**, which is used in our solution to create a copy of the existing machine. Marie's paper doesn't have an example of copying a machine, so it is unclear if her solution allows her to do so or not. As Marie's modularized Event-B allows the user to create models from scratch, it is obvious that not only abstract machines can be created, but also more complicated ones, which refine existing machines. The rest of Marie's paper is dedicated to the description of creating a refinement. Our solution doesn't allow us to create machines from scratch, so there is no possibility of creating refinements as well.

5.2 Conclusions

To conclude and answer the questions asked at the beginning of this Chapter, let's create a comparison table of the two solutions (Table 5.1). This table will compare main characteristics of the two approaches to solving the problem of poor structure of the formal language Event-B.

Table 5.1 A comparison with Modular Event-B

№	Characteristic	Our solution	Modular Event-B
1	Allows the user to create a specification from scratch		✓
2	Supports the refinement of the existing machine		✓
3	Allows the user to rename an event/variable inside the event	✓	✓
4	Allows the user to merge two or more events	✓	✓
5	Allows the user to extend existing event		✓
6	Allows the user to copy an existing event	✓	
7	Supports specification-based operators	✓	✓

According to this table, our solution for the problem of 'standardization' of Event-B was quite successful. Despite the fact that functions supported by our plugin are limited, it is obvious, that our plugin allows the user to manipulate Event-B elements with the help of simple, but 'standard' commands. The lack of functionality can be eliminated by adding new commands and implementing them. The first two characteristics, specified in Table 5.1 and implemented in Modular Event-B, are closely related to each other and the implementation of the first one will make it possible to implement the second one. The reason why this functionality wasn't added to the plugin is that the main purpose of this project was to make it possible to use 'standard' operations to be able to manipulate Event-B elements. This purpose was achieved by implementing the third, fourth and sixth characteristics. The plugin uses three 'standard' operators, so called specification-based operators. As all these operators are standard and keep their properties, regardless of the formal language's syntax, it was quite easy to compare our and Marie's solutions. These two approaches are quite different but are trying to solve the same problem and the results obtained confirm that the current problem is not only important, but also solvable. Our solution has both advantages and disadvantages, and these weak points can be used in future to improve the current version of the plugin. Future possible improvements will be discussed in detail in the next Chapter.

Chapter 6

Conclusions and future work

In this thesis one of the main problems of existing formal languages was considered. This problem can be described as poor 'standardization' of the syntax. This means that every single language uses its own key words and syntax. This leads to huge problems when the developer of one formal language wants to work on the model written in another formal language. Sometimes the difference between these languages is so huge, that the development in a new language takes a lot of time in studying the syntax and features of the certain IDE. The same problem can be met when starting to work with Event-B. It is a very powerful formal language, which allows the user not only to create models, but also to write proof obligations and prove some logic assumptions. The principal tool which supports development in Event-B is the Rodin Platform. The main purpose of this project was to create a plugin for the Rodin Platform and implement some basic 'standard' operators to manipulate Event-B elements without requiring a deep knowledge of Event-B. This Chapter presents results of the research, analysis and development, described in this thesis.

6.1 Conclusions

The development of the plugin started after research and deep analysis of the subject area. Chapter 2 presented main features of Event-B, basic information about theory of institutions and specification-based operators. The comparison between our solution and existing plugins was provided as well. Chapter 3 contained detailed information about the key requirements, the implementation phase and a comparison with the well-structured formal language CASL. In addition, attention was paid to dependability attributes as the most important evaluation criteria. Chapter 4 described the evaluation of the developed plugin based on requirements fulfillment and a comparison with existing plugins. Chapter 5 contained a case study, which was based on the simple Event-B example of a traffic-light simulator. The case study represented the comparison between two approaches of 'standardization' – our plugin and PhD student Marie Farrell's Modular Event-B.

The research question **RQ1** can be finally answered based on the work done. Software verification and the use of different formal languages can be simplified by making these languages more 'standard', which can be done by standardization of the syntax of these languages. Answering the question **RQ1.1**, we can say that developers, who use other formal languages, will be able to use the Rodin Platform and our plugin without the necessity of having deep knowledge of Event-B. This was reached by the implementation of SBOs inside the Rodin Platform. The question **RQ1.2** was the most difficult question to answer, but even the problem of keeping the proof obligations consistent during the renaming and merging operations was solved successfully. Creating the proof tree allowed us to do so. Simplicity, the speed of commands execution and usability are some of main features of our plugin. It reduces the efforts of studying and using Event-B and combines features of two existing plugins. This was described in detail in Section 4.2.

The case study showed two different approaches in solving the same problem of 'standardization' of Event-B. The comparison was conducted with the Modular Event-B language invented by Maynooth University PhD student – Marie Farrell. All the implemented key words were compared to Modular Event-B analogues and both advantages and disadvantages of our plugin were noticed.

Our plugin for the Rodin Platform supports a scriptable language, which is based on SBOs. It will be interesting for formal languages developers who intend to work with models written in Event-B, but don't have deep knowledge of this language. Our plugin will help these developers not only to understand Event-B deeper, but also to manipulate its elements using 'standard' operators. The functionality of the plugin will increase and Event-B will gradually become 'standard' formal language, easy to use and more popular.

6.2 Future work

The number of specification-based operators is limited, however, not all of them were implemented in the plugin. This work can be further extended by other developers who are particularly interested in formal reasoning and want to improve the current situation of using different operators for the same operations in different formal languages. For instance, the current version of plugin allows the user to rename only one element per execution. This could be extended by multiple renaming.

During the case study some problems of missing functionality in the implementation had appeared. One of these problems that could be potentially solved in future versions is a problem of impossibility of creating new machines from scratch. Sometimes it is necessary to add one more machine into the model, but the current version of our plugin doesn't have this functionality. Copying the existing machine does not always help to solve this particular problem. The lack of a possibility to creating machines also prevents creating their refinements as well.

The next step which could be also done to improve the plugin is implementing key word **then**. This key word will allow the user to extend the event, adding new actions or guards. The part of this functionality currently works with key word **and**. This key word allows the user to merge two events and actions/guards inside, but it works as part of merging machines and doesn't work separately.

As our plugin represents a multi-page editor, which supports some kind of scriptable language, the syntax of this language could be highlighted, at least for key words. This would make the source code more understandable and easy to read and write. During the work on the current version of the plugin we were more focused on the functionality, than on representation, so some changes in user interface will definitely have a benefit. One more improvement in the user interface, which could be done, is auto code completion. This feature will allow the user not to worry about the spelling of elements names and concentrate on writing code. It will reduce the number of errors as well.

One more thing that could be changed in the current version of the plugin is the implementation of key word **and**, which is used for merging machines. At the moment the created composition is saved in the first specified machine, whereas it should be saved in separate machine. The reason for that implementation was the speed of creating a copy of existing machine. This creating takes too much time and copying of missing events from the second machine fails, because the destination machine for composition still doesn't exist.

The list of possible features is not definitive. The plugin has lots of features and lots of features could be added. Every developer decides how his application should look like and what functionality should it have. The further development of this plugin depends on the next developer.

References

[1] J.-R. Abrial. The Event-B Modelling Notation, October 2007, Available at http://sourceforge.net/projects/rodin-b-sharp/

[2] Event-B homepage. http://www.event-b.org

[3] J.-R. Abrial. Modeling in Event-B. System and Software Engineering, Cambridge University Press, 2010

[4] J.-R. Abrial. A system development process with Event-B and the Rodin platform. Springer, Heidelberg, 2007

[5] RODIN project homepage. http://rodin.cs.ncl.ac.uk/.

[6] M. Jastram, M. Butler. Rodin User's Handbook: Covers Rodin V.2.8. CreateSpace Independent Publishing Platform, 2014.

[7] E. W. Dijkstra. Notes On Structured Programming, Second edition. T.H.-Report 70-WSK-03, April 1970

[8] M. Farrell, R. Monahan, J. F. Power. An Institutional Approach to Modularisation in Event-B, Draft report available from http://www.cs.nuim.ie/~mfarrell/

[9] J.-R. Abrial. Formal Methods in Industry: Achievements, Problems, Future. Swiss Federal Institute of Technology Zurich, 2008

[10] A. Gondal, M. Poppleton, C. Snook. Feature Composition – Towards product lines of Event-B models. Dependable Systems and Software Engineering Group University of Southampton, Southampton SO17 1BJ, UK, October 2010

[11] A. Gondal, M. Poppleton, M. Butler. Composing Event-B Specifications - Case-Study Experience. Springer, 2011

[12] M. Antkiewicz, K. Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. ACM Press, 2004

[13] J. Sorge, M. Poppleton, M. Butler: A basis for feature-oriented modelling in Event-B. Springer, 2010

[14] A. Gondal, M. Poppleton, M. Butler, C. Snook. Feature-Oriented Modelling Using Event-B. School of Electronics and Computer Science University of Southampton, Southampton, SO17 1BJ, UK, 2010

[15] M. Poppleton, B. Fischer, C. Franklin, A. Gondal, C. Snook, J. Sorge. Towards Reuse with "Feature-Oriented Event-B". Dependable Systems and Software Engineering University of Southampton Southampton, SO17 1BJ, UK. 2008.

[16] J.-R. Abrial. The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA, 1996.

[17] S.Holl. Refactoring of B models. Bachelor thesis, Computer Science Department, Heinrich Heine University Düsseldorf, 2007, pp.1-52

[18] Event-B plugins developer tutorial. http://wiki.event-b.org/index.php/Rodin_Plug-ins

[19] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. Journal of the A.C.M., 39(1):95 146, January 1992

[20] Peter D Mosses, editor. CASL Reference Manual, volume 2960 of Lecture Notes in Computer Science. Springer, 2004.

[21] Michel Bidoit and Peter D Mosses. CASL User Manual, volume 2900 of Lecture Notes in Computer Science. Springer, 2004.

[22] T. Mossakowski, A. Haxthausen, D. Sannella, A. Tarlecki. Casl — the common algebraic specification language: semantics and proof theory. Computing and Informatics, 2003-Sep-4.

[23] F. Lichtenberger. Introduction to CASL, the Common Algebraic Specification Language presentation. Research Institute for Symbolic Computation (RISC) Johannes Kepler University, Linz, Austria, 2011

[24] R. S. Boyer, J S. Moore. A Computational Logic. Academic Press, 1979.

[25] D. Kalish, R. Montague, G. Mar. Logic: Techniques of Formal Reasoning (Second Edition). Harcourt Brace Jovanovich, 1980.

[26] R. Silva, M. Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. School of Electronics and Computer Science University of Southampton, UK, 2009

[27] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. EMF: Eclipse Modelling Framework. Addison-Wesley, 2009.

[28] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, Ph. Vanderheyden. Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. IBM, February 2004.

[29] IEC*, Electropedia del 192 Dependability*, http://www.electropedia.org, select 192 Dependability, see 192-01-22 Dependability. 2015-02

[30] N. Madhav, S. Sankar. Application of Formal Specification to Software Maintenance. Stanford University, 1990