Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Sanjeevan Sritharan

12 May 2020

Automated Translation of Event-B Models to SPARK Proof Annotations

Project supervisor: Dr. Thai Son Hoang
Second examiner: Dr. Julian Rathke

A project report submitted for the award of
MEng Software Engineering

Abstract

Ensuring and verifying the properties of safety- and security-critical software is paramount. Event-B is a modelling language which enables the design of systems, using mathematical proofs ensuring the conformity of the system to declared safety requirements. SPARK is a programming language making use of static analysis tools which verify written code correctly implements the properties of the system as specified in the form of written proof annotations. SPARK has been used in many industry-scale projects to implement safety-critical software. Manually writing SPARK proof annotations can be time-consuming and tedious. The aim of this project was to create a tool, in the form of a Rodin plug-in, to translate an Event-B model into a set of SPARK specifications in a SPARK specification file, in the form of proof annotations and other structures, from which SPARK code written can be verified together with, hence ensuring the correct behaviour of the software. A background literature review was conducted to find related works, to help guide the approach to this project. The common forms of a lot of predicates in Event-B, such as those involving set operations, had no direct counterpart in SPARK. Hence, an extra file in SPARK containing function definitions for common Event-B predicates was written. Using these function definitions, a set of translation rules from Event-B predicates to SPARK code was compiled, along with other translation rules for the making of other SPARK constructs which were needed in the specification file. Using these conceptual rules, the plug-in was created in Eclipse as an Eclipse plug-in to run on the Rodin target platform. Finally, this plug-in was tested with several Event-B models to ensure that the generated code was the same as the expected code.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

- **In the plug-in development project, in the 'execute' method in the 'SparkTranslate.java' file which is called when the plug-in is activated, the code used to unpack the 'ExecutionEvent' object and extract the 'IMachineRoot' object is taken from https://www.programcreek.com/java-api-examples/?class=org.eclipse.ui.handlers.HandlerUtil&method=getCurrentSelection. This is acknowledged in the code itself.**

- **Furthermore, the Java code used to extract information from the machine root, such as carrier sets, invariants etc., were written by me, adapting the code written in ch.ethz.eventb.utils.EventBSCUtils.java written by author @htson (Dr. Thai Son Hoang). This is acknowledged in the code itself.**
- **The code used to output the code into a file is adapted from a tutorial which can be found at https://www.w3schools.com/java/java_files_create.asp. This is acknowledged in the code itself.**

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment, other than the sections of work which were previously submitted in the progress report for this project, as is permitted according to the rules for academic integrity for the Part III project.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell aiofficer@ecs.soton.ac.uk

Acknowledgements

I would like to thank my project supervisor, Dr. Thai Son Hoang, for all his help during the course of this project. As this was a challenging project, it cannot be understated how helpful it was to have a supervisor who was extremely responsive and always engaging in every supervisor meeting we had. I would also like to thank him for suggesting this topic to me when I was struggling to find one.

I would also like to thank my second supervisor, Dr. Julian Rathke, for his helpful insight into some of the specifics of this project during our meeting.

Contents

Abstract **2**

Statement of Originality **3**

Acknowledgements **6**

Contents **7**

1. Introduction **10**
1.1 Event-B 10
1.2 SPARK 11
1.3 Problem 11
1.4 Project Goals 12

2. Literature Review **13**
2.1 Translating Event-B to SPARK 13
2.2 Translating Event-B to Dafny 13
2.3 Translating Event-B to JML and Java 14
2.4 Related Works 15

3. Running Example **17**
3.1 Context 17
3.1.1 Carrier Sets 17
3.1.2 Constants 17
3.1.3 Axioms 17
3.2 Machine 18
3.2.1 Variables 18
3.2.2 Invariants 18
3.2.3 Events 19

4. Specification and Analysis of Solution **21**
4.1 Scope 21
4.2 Translation Guide 22
4.2.1 Translation of Types 23
4.2.2 Translation of Sets 23
4.2.3 Translation of Relations 23
4.2.4 Translation of Predicates 24
4.3 Translation of Event-B Model 25
4.3.1 Translation of Carrier Sets 26
4.3.2 Translation of Constants 28

4.3.3 Translation of Axioms 28
4.3.4 Translation of Variables 29
4.3.5 Translation of Invariants 30
4.3.6 Translation of Events 31
4.4 Alternative Approaches 36
4.4.1 Use of Functional Sets 36
4.4.2 Use of Formal Ordered Sets 37

5. Implementation 38
5.1 Plug-In Development Environment Setup 38
5.2 Code Generation Process 38
5.2.1 Constructs Package 39
5.2.2 SparkTranslate Class 40
5.2.3 Translate Method 40
5.2.4 Translator Class 41

6. Testing and Evaluation 42
6.1 JUnit Tests for the Translate Method 42
6.1.1 Example - TestEquals 42
6.1.2 Example - TestIn 42
6.2 Testing on Other Event-B Models 44

7. Project Management 45
7.1 Risk Assessment Table 45
7.2 Gantt Chart Comparison 45
7.3 Weekly Supervisor Meetings 47

8. Conclusion and Future Work 48
8.1 Summary and Justification of Work 48
8.2 Future Work 48

References 50

Appendix 53
A.1 Full Running Example Model 53
A.1.1 Carrier Sets 53
A.1.2 Constants 53
A.1.3 Axioms 53
A.1.4 Variables 54
A.1.5 Invariants 54
A.1.6 Events 55
A.2 Plug-In Development Environment Setup 59
A.2.1 Configuring the Rodin Target Platform 59

A.2.2 Plug-in Extensions 60
A.3 Full Table of Predicate Translation Rules 60
A.4 Declarations in sr.ads 73
A.5 Examples of Translation 87
A.5.1 Carrier Set - PERSON 87
A.5.2 Invariant - location ⊆ permission 87
A.5.3 Event - AddPerson 88
A.6 Tests on Other Event-B Models 89
A.6.1 Room Booking System 89
A.6.2 Club Courses System 93
A.6.3 Island Car Access System 95
A.7 Original Project Brief 97
A.8 Contents of Design and Data Archive 100

1. Introduction

1.1 Event-B

Event-B (Abrial, 2010) is a modelling tool used to model and design software systems, of which certain properties must hold, such as safety properties. This tool is extremely useful in modelling safety-critical systems correctly, using mathematical proofs to show correctness of models in adhering to its specification. These proofs are proof obligations which are discharged to show mathematical correctness. Hoang (2013) explains how Event-B models software systems. Models consist of constructs known as machines and contexts. A context is the static part of a model, such as constants, carrier sets (which are conceptually similar to datatypes), and axioms. Axioms are properties of constants which always hold. Machines describe the dynamic part of the model, that is, how the state of the model changes. The state is represented by the current values of the variables, which may change values as the state changes. Invariants are declared in the machine, stating properties of variables which should always be true, regardless of the state. Events in the machine enable state changes. Each event has a set of actions which happen simultaneously, changing the values of the variables, and hence the state. Events have guards which are predicates on variables and parameters which must hold true for event execution. Every machine has an initialisation event which sets initial variable values. Proof obligations are generated and required to be discharged to show that no event can potentially change the state to one which breaks any invariants, a potentially unsafe state.

A core feature of Event-B, stepwise refinement, is not used within the scope of this project, which focuses on Event-B's modelling of a single abstraction level model. Further details on refinement can be found in the papers by Abrial (2010) and Hoang (2013).

1.2 SPARK

SPARK (Barnes, 2003) is a programming language used for systems with high safety standards. It includes tools performing static verification on programs written in the language. SPARK is a subset of another programming language, Ada (Booch and Bryan, 1993), which is also used for safety-critical software. SPARK removes several major constructs from Ada, allowing feasible and correct static analysis.
According to Murali and Ireland (2012), SPARK "includes a language of annotations", which are specifications for a SPARK program, clarifying what the program should do. While program annotations focus on the flow analysis part of static analysis, focusing on things such as data dependencies, proof annotations support "assertion based formal verification". This involves pre-conditions, which are required to hold true on calling a subprogram, without which the subprogram has no obligation to work correctly, and post-conditions, conditions which should be achieved by the actions of a subprogram, provided the pre-conditions held initially (Hoare, 1969). Proof annotations also involve loop invariants, which are conditions which hold true in every iteration of a loop.

Program annotations deal with aspects such as the global aspect of a subprogram, specifying which global variables are involved in this subprogram, and how they are used. Another aspect deals with the dependencies involved for global variables whose values are changed, namely, which variables or parameters affect the new value of the variable. This mix of proof and program annotations ensure that any implementation written in SPARK adheres to its specification, producing reliable, safe software.

1.3 Problem

It is important that the software in safety- and security-critical systems behaves correctly. SPARK proof annotations and program annotations help achieve this. It is possible, when implementing a system initially modelled in Event-B, to write a SPARK specification file for that system, which would contain the set of annotations modelling the behaviour of that system. However, manually generating this can be time-consuming, and so this method may be limited by time constraints (Murali and Ireland, 2012). In fact, the model checking process for any large system requires a lot of effort (Gluck and Holzmann, 2002). These two findings may be linked, as generating verification conditions is time-consuming, and may contribute to the total amount of effort needed in the

entire verification, validation and testing process. Hence, SPARK-implemented systems are no exception.

1.4 Project Goals

The overarching goal of this project is to develop and build a tool, in the form of a Rodin (Abrial et al., 2010) plug-in, which automatically generates SPARK (Barnes, 2003) pre- and post-conditions from an Event-B (Abrial, 2010) model. The aim is to have a tool which can generate these conditions from an Event-B model of a particular level of refinement. A model which is too abstract may not have a sufficient level of detail to derive these conditions from. A user can then use these pre- and post-conditions, or contracts, to write SPARK code implementing the system. Any implementation can be checked against the generated contracts using the static analysis tools in SPARK, which help verify that the implementation follows the specification represented by the contracts. As the aim is to translate models which may be relatively abstract, as well as concrete models, it is reasonable that the scope of this project focuses only on generating contracts, and not auto-generating implementing code as well, as generating code requires deriving the concrete algorithmic structure present in the Event-B model, something which might not be present in an abstract model. However, as described in the literature review report below, there do exist some tools which can generate code from abstract models. Hence, this is postulated as a possible extension to this project. Another possible extension identified is the generation of loop invariants. The following lists out the main goals of this project:

1) Extend and develop a set of translation rules from Event-B constructs to SPARK pre- and post-conditions.
2) Research the Rodin API, for the development of a Rodin plug-in which performs the automated translation using the aforementioned set of rules.
3) Test and validate this tool on several case studies.
4) If time permits, develop translation rules to generate code and loop invariants, and implement these in the Rodin plug-in as well.

2. Literature Review

2.1 Translating Event-B to SPARK

Only one paper describing an approach to generating SPARK code could be found (Murali and Ireland, 2012). This approach involves not only generating pre- and post-conditions, along with loop invariants, but also generates implementing SPARK code from Event-B models, using the merging rules described by Abrial (2010), which describe how to generate "sequential programs from Event-B models". Given that such work exists, where not only code specifications, but code itself, is generated from Event-B models, it may seem odd that the scope of this project is limited to generating pre- and post-conditions from Event-B models. However, the model used in this paper is fairly concrete. This project instead aims to derive proof annotations from models at any level of abstraction. The method of translating, implementing the merging rules, follows a process involving adding more events to an initial machine in refinement steps, until the model fully represents an algorithm to be implemented. Given this, the merging rules used in this paper may not be applicable to very abstract models, as such an algorithm may not be represented or derivable. However, it may be interesting to adapt these merging rules, so they can generate SPARK code from abstract models. Furthermore, the translation rules from Event-B to SPARK assertions shown in this paper are limited, particularly in terms of set-theoretical constructs. This is an issue to address given Event-B is a set-theory heavy tool.

2.2 Translating Event-B to Dafny

An example of this is given by Dalvandi (2018). This work explores mapping Event-B with Dafny (Leino, 2010) constructs. This paper claims that a "direct mapping between the two is not straight forward". Due to the increased richness of the Event-B notation compared to Dafny, only a subset of Event-B constructs can be translated. Like the previous paper, this paper suggests that a particular level of refinement must be achieved by the Event-B model, to reduce "the syntactic gap between Event-B and Dafny". However, the level of refinement required is needed to have a model containing only those mathematical constructs which have a counterpart in Dafny, not to obtain a model with a clear algorithmic structure present in its events. As such, this approach can still translate fairly

abstract models. This paper states the assumption that the “machine that is being translated is a data refinement of the abstract machine and none of the abstract variables are present in the refined machine”. This approach uses Hoare logic (Hoare, 1969), by transforming events into Hoare triples, and deriving the relevant pre- and post-conditions.

There exists previous work involving generating Dafny contracts from Event-B models (Dalvandi, Butler and Rezazadeh, 2015). Firstly, this paper also mentions the need for refinement of the model, again due to the syntax gap between Event-B and Dafny, to ensure every construct in the model has a Dafny counterpart. This paper also mentions the use of a method constructor statement to represent a group of similar events in the Event-B model. The paper describes the formulation of post-conditions from the guards and before-after predicates representing event actions. However, this paper only describes using the invariants of the Event-B machine to derive the pre-conditions, and does not describe using the event guards. Furthermore, there is less detail in this paper about the categorisation of guards, which in the work referenced earlier provides a valuable insight into the role of event guards in contracts. The case study example illustrated in this paper is not very general, and unsurprisingly, the previous paper has more exhaustive example case studies which are applied and tested. As such, this paper simply acts as a precursor to the main work done by Dalvandi (2018).

2.3 Translating Event-B to JML and Java

Another approach explored is the translation of Event-B to JML-annotated Java (Leavens, Baker and Ruby, 1999) programs (Rivera and Cataño, 2014), which provides a translation “through syntactic rules”. JML provides specifications which Java programs must adhere to, and so it is similar to contracts. This approach generates Java code as well as JML specifications. Unlike the previous approaches, instead of grouping similar events, every single event is translated independently. This is perhaps not as efficient, as grouping similar events and using specific case guards in the post-conditions to differentiate between the expected outcomes of the different events in a group gives an insight into how these events work and their expected behaviour, in addition to saving space in the generated code by having fewer methods. This is only foreseen to be a problem when the translated model is concrete, and has several events representing different situational implementations of a single abstract event. This paper

demonstrates translation rules of machines and events to JML-annotated programs. The approach of deriving the JML specifications can possibly be adapted for this project, and can perhaps be considered an alternative approach to the one by Dalvandi (2018). However, an interesting thing to note from this paper is that the approach given has the ability to generate code even from abstract models. The translation rules given can generate code from variables and assignments to variables in actions, in any level of abstraction or refinement. Hence, this approach of generating code can possibly be adapted for the generation of SPARK code.

2.4 Related Works

In addition to these approaches, several more academic papers and sources were researched to fully understand the domain, and any other dependencies relevant to the aims of this project.

One paper details the translation of Event-B proof obligations into Dafny (Cataño, Leino and Rivera, 2019). The Dafny code generated is then verified using the verification tools available to Dafny. The translation is done so that Dafny code is "correct if and only if the Event-B refinement-based proof obligations hold". In other words, this approach allows users to verify the correctness of their models using a powerful verification tool. Specifically, this paper focuses on refinement proof obligations, showing that the concrete model is a correct refinement of the abstract model. While this is outside of the scope of the project, it nevertheless introduces some translation rules which may be adapted for this project. For example, the paper demonstrates how invariants may be translated into Dafny and used in pre-conditions. It also shows an example of how relations in Event-B may be modelled in Dafny.

Edmunds et al. (2012) propose an approach to extending the Rodin theory plug-in, which introduces "translation rules that map Event-B formulas to code". Basically, for any theory which introduces new operators and types, this approach shows how translation rules can be added to this theory to describe how these constructs should be translated. The translation rule used to translate the newly-introduced constructs is identified by pattern matching to the correct translation rule, given the construct.

Finally, Wright (2009) discusses the translation of Event-B directly to C. Again, it has the same limitations as other approaches in that the final model being translated has to be concrete, "in forms similar to the emitted C", as the tool

requires that the model to be translated should be a subset of Event-B easy to translate. No pre- or post-conditions are created, rather, a set of translation rules are shown, and these are used to translate the Event-B constructs into C.

3. Running Example

The following running example which is used throughout the rest of this report is adapted from an example given by Butler (2017). The running example is an Event-B model modelling a building access system. To save space, only parts of the model will be elaborated on in this section, with the entire model and accompanying explanations in the appendix (Section A.1).

3.1 Context

This next section shows the context of this model, broken down into sections.

3.1.1 Carrier Sets

Carrier Sets PERSON BUILDING

Each carrier set introduces a type. For example, PERSON denotes the type of an element able to register for the system, have permissions for buildings, enter buildings and so on.

3.1.2 Constants

Constants n

This model only has a single constant, n, which is present only to show how constants are translated. It does not play any further part in the model.

3.1.3 Axioms

For each carrier set, there must exist axioms declaring its finiteness and cardinality, as such:

Axm1: finite(PERSON)
Axm3: card(PERSON) = 10

The translation tool requires these axioms for the translation of each carrier set.

The translation tool also requires axioms which give each constant a type and a value, as such:

Axm5: n = 20

3.2 Machine

The following sections describe the Event-B machine which sees the aforementioned Event-B context. Again, not all constructs in the example will be elaborated on in this section, with full details found in the appendix.

3.2.1 Variables

Variables person inside outside location permission

The variable person is the set of registered users of the system, while permission is a relation mapping which registered user has permission to access which particular building.

3.2.2 Invariants

This model has five invariants, three of which are shown here:

Inv1: person $\subseteq$ PERSON

This invariant describes the set of registered users, person, as a subset of PERSON. This indicates that elements in person have the type PERSON.

Inv2: partition(person,inside,outside)

The set person is partitioned into the sets inside and outside. The partition operator is actually a shorthand meant to describe the following formula:

partition(A,B,C) $\Leftrightarrow$ (A = B $\cup$ C) $\wedge$ (B $\cap$ C = $\varnothing$)

This invariant states that the union of inside and outside equals person, and that the intersection of inside and outside is empty. This indicates that all registered users are either inside or outside but not both, and that every element in inside and outside is a registered user.

Inv4: permission ∈ person ↔ BUILDING

Permission is a relation from person to BUILDING. Each registered user can have permission to access 0 or more buildings.

3.2.3 Events

Finally, this model contains seven events, with three shown and explained here:

INITIALISATION
THEN
    Act1: person ≔ ∅
    Act2: inside ≔ ∅
    Act3: outside ≔ ∅
    Act4: location ≔ ∅
    Act5: permission ≔ ∅
END

The initialisation event is meant to be the first event triggered. When first initialised, the system must not have any registered users. Hence, the person, inside and outside sets must be empty, along with the location and permission relations.

AddPerson
ANY
    p
WHERE
    Grd1: p ∈ PERSON ∖ person
THEN
    Act1: person ≔ person ∪ {p}
    Act2: outside ≔ outside ∪ {p}
END

This event registers a person into the access system. The guard ensures that the parameter p has the type PERSON, but is not already a registered user. The first

action makes p a registered user. In keeping with invariant 2, p must either be inside or outside. As p will not have any permissions yet, p is added to outside.

AddPermission
ANY
p
b
WHERE
Grd1: p ∈ person
Grd2: b ∈ BUILDING
Grd3: p↦b ∉ permission
THEN
Act1: permission ≔ permission ∪ {p↦b}
END

This event grants access permission for a particular registered user for a specific building. The first two guards ensure p is a registered user and b is a building, and the third guard ensures this permission doesn't already exist. The action adds the tuple to permission.

4. Specification and Analysis of Solution

4.1 Scope

The exact scope of the project was determined before any technical work was started so as to guide the project appropriately.

The first major decision was what would actually be generated by the translation tool. It was decided that the tool would only generate a SPARK package specification file, which contains the specifications of that system or model. A user could then write a SPARK package body file, which would contain the code which implements the system. The static analyser could ensure the code in the body file conforms to the requirements in the specification file. This decision was taken due to the aim of translating relatively abstract Event-B models, from which it is difficult to derive a deterministic structure from which the SPARK body file could be written. It is conceivable that a translation from an Event-B model to full code generation in SPARK can be implemented, even for relatively abstract models, so this can be something to consider for future work.

Secondly, it was decided that the only built-in primitive Event-B types that would be allowed were the set of Integers, Z, and the boolean type BOOL. This was done to simplify the translation and code generation process.

There is an inherent problem in translating mathematical sets into constructs in SPARK, whichever form they may take to represent the sets, namely that mathematical sets do not have a finite limit to its size, that is, no matter how many elements are present in a set, more elements can be added to it. This is contrasted with the structures which can be created in a safety-focused programming language like SPARK, which require a strict size limit to be set from the start, as memory is finite. As such, another constraint of this project's implementation is the requirement that every carrier set defined in the Event-B model has to be finite and have a set cardinality. The translation tool can use this information to then create a representation of this carrier set with the corresponding size.

Due to the richness of the syntax defining predicates in Event-B, it was challenging to have an implementation which translates any predicate in Event-B. This is because of the difficulty in translating certain set operators into equivalent SPARK operators, or translating the result of set operations into an equivalent form. While technically, this would have been possible by writing SPARK code which constructs operators which mirror the functions of these operators in Event-B, due to the nature of the static analyser in SPARK, it was extremely difficult to use these operators in the specification file, and then write a SPARK package body file to implement the specification in a way which satisfies the SPARK provers. Instead, this project is constrained to only translate predicates of the most common forms in Event-B. The set of translatable predicates was made to be very, but not completely, exhaustive.

Another restriction with the approach chosen is that sets and relations cannot contain any Boolean values, due to the representation of sets as SPARK arrays. This is explained further in this chapter.

Other restrictions also include the assumption that the machine does not refine a more abstract machine, and that the context seen by the machine does not extend any other contexts. Furthermore, the machine should only see one single context.

Finally, it was decided to limit relations to just having 2-tuples. Although relations are technically just sets containing n-tuples, where n can be 2 or higher, it became intractable to represent relations of n-tuples with n higher than 2, as for each value of n, a new subtype declaration for a relation with n-tuples had to be declared. Relations commonly only have 2-tuples, so the scope was restricted to this.

4.2 Translation Guide

The approach taken aims to translate an Event-B model and generate code in a package specification file. For consistency, this package specification file shall be called **test.ads**. A separate file called **sr.ads** has pre-written SPARK code with type declarations for sets and relations, as well as functions representing particular predicates. The functions written in the sr.ads file are shown in the appendix (Section A.4).

4.2.1 Translation of Types

While elements of type ℤ or BOOL in Event-B have a direct counterpart in SPARK, it is not clear how carrier set types should be represented. It was decided that every carrier set type should be represented as a subtype of Integer.

For example, to represent a carrier set type T, which has cardinality 100, the following subtype declaration is declared in SPARK:

```
subtype Ttype is Integer range 1 .. 100;
```

4.2.2 Translation of Sets

Sets are represented as SPARK arrays of Boolean values, indexed by any Integer range. The Integer range in this case is the representation of the type of each element in the set.

The following type declaration is made in sr.ads:

```
type set is array (Integer range <>) of Boolean;
```

Hence, a set called set_T containing elements of type T can be declared as:

```
set_T : set (Ttype);
```

In this case, Ttype is the subtype of Integer with a certain range, which acts as the index of the array set_T. For a value v of type T, if v is a member of set_T, then set_T (v) = True, if not, set_T (v) = False.

4.2.3 Translation of Relations

Relations are represented as two-dimensional SPARK arrays of Boolean values, indexed by any two Integer ranges. These ranges represent the types of the elements in the domain and the range of the relation.

The following type declaration is made in sr.ads:

```
type relation is array (Integer range <>, Integer range <>) of Boolean;
```

Hence, a relation called relation_TU containing elements of type T x U can be declared as:

```
relation_TU : relation (Ttype,Utype);
```

For a tuple v↦w of type T x U, if v↦w is a member of relation_TU, then relation_TU (v,w) = True, if not, relation_TU (v,w) = False.

With this approach to representing sets and relations, these Event-B constructs can thus only have carrier set or Integer type elements, and not BOOL elements, as the BOOL type is represented by the Ada Boolean type, which is not a range of Integers.

4.2.4 Translation of Predicates

The richness of the mathematical language and operators used to represent in predicates in Event-B is difficult to replicate in SPARK. An attempt was made to allow the translation tool to translate predicates which take the most common forms.

For most of these translations, it was tedious to translate these predicates into equivalent predicates. Take for example, the predicate S ⊆ T, where S and T are both sets. Translated to SPARK, this predicate would be represented as:

```
for all x in S'Range => (if (S (x)) then (T (x)))
```

As the subset operator does not exist in SPARK, the above predicate is constructed, which uses the definition of the subset operator. However, there exists an alternative solution which vastly improves the translation process. Firstly, a function called “isSubset” is defined in the sr.ads file, written the following way:

```
function isSubset (s1 : set; s2 : set) return Boolean is
    (for all x in s1'Range => (if s1 (x) then s2 (x)))
    with
     Pre => (for all x in s1'Range => (for some y in s2'Range => (x = y)));
```

This function takes two sets and returns True if for every x for which s1 (x) is True then s2 (x) is also True (s (x) being True means x is in s). Note the necessity of the pre-condition clause for this function, which requires that every value in the index of s1 is also a value of s2's index. Put into other words, it requires that the two sets are of the same type.

Then, with this function declaration in sr.ads, we can translate the original predicate, S ⊆ T, into the following:

```
isSubset (S,T)
```

This is a much more concise way to represent the predicate in SPARK. This does mean that the predicates which can be translated are limited to those which have a representative function definition in sr.ads.

The main effort in the translation process was producing the translation rules for predicates, as an exhaustive set of rules was generated to be able to translate as many predicates as possible. The full table of predicate translation rules is given in the appendix (Section A.3), as well as the full list of predicate function declarations in the sr.ads file.

4.3 Translation of Event-B Model

This next section is a guide to how the translation process works in translating a full Event-B model into SPARK specification code. The example used in this section is the running example introduced in Chapter 3.

Before proceeding, it is worth mentioning that there are a few lines which the user has to manually write themselves. This is not foreseen as a big problem, as these lines will be almost exactly the same for every specification file for any translation, the only differences arising from the name of the specification file (which in this example, is test.ads). The first two lines in the file have to be:

```
with sr;
use sr;
```

These are required so the file can use the predicate functions and the type declarations of sets and relations from the sr.ads file.

The next few lines denote the name of the package file and indicate the use of SPARK:

```
package test
with SPARK_Mode
is
```

The package name test here follows from the name of the file test.ads. Finally, the file must end with the line:

```
end test;
```

As such, the translation tool aims to generate code between the lines “is” and “end test;”, in the following way:

```
with sr;
use sr;

package test
with SPARK_Mode
is

-- code generated by translation tool

end test;
```

4.3.1 Translation of Carrier Sets

For carrier sets, there are three kinds of SPARK declarations that need to be made. Firstly, there must be a type declaration for each carrier set, introducing a new type to represent elements of that carrier set. As mentioned earlier, these types will be subtypes of Integer, which will range from 1 to the cardinality of the carrier set. The second kind of declaration will be a variable declaration declaring the carrier set as a set itself. This is required as carrier sets are frequently referenced in the machine’s invariants and event guards. The third kind of declaration is a general function declaration for all carrier sets in the machine, ensuring their “fullness”, that is, each carrier set must contain all elements of its type.

Take the carrier set PERSON from the running example. The axiom indicating its cardinality is:

Axm3: card(PERSON) = 100

Using this axiom, the type declarations for this carrier set is:

```
subtype PERSONtype is Integer range 1 .. 100;
```

Following this, the tool will also generate the declarations of the variable representing the carrier set itself:

```
PERSONcs : set (PERSONtype) := (others => True);
```

It is the PERSONcs variable which will be used to reference the carrier set as a set itself. The 'cs' suffix is added because SPARK is not case-sensitive when it comes to distinguishing variable names, and as it is relatively common practice to have carrier sets and set variables with the same name, such as PERSON and person in the running example, it was decided that the 'cs' suffix would be appended to all carrier set names.

Another note to point out is the variable definition "(others => True)". This ensures that the initial carrier set array has the value True at every indexed position, signifying that every value of that type is present in the carrier set.

Finally, a function declaration is needed to ensure that all carrier sets remain full. Using the information for the BUILDING carrier set as well from the running example, the tool will generate the following function:

```
function cs return Boolean is
    (isFullSet (PERSONcs) and then isFullSet (BUILDINGcs)) with
      Global => (PERSONcs,BUILDINGcs),
      Depends => (cs'Result => (PERSONcs,BUILDINGcs));
```

This function declaration "isFullSet" is defined in the file sr.ads, which returns True if the set passed in as the argument to this function contains the value True at every indexed position. This function is used as the pre-condition and post-condition for every SPARK procedure representing an event, ensuring they remain 'full'.

Further points to note are the use of the Global aspect, which indicates that this function only uses the variables PERSONcs and BUILDINGcs, and the Depends aspect, which indicates that the result of this function, True or False, will depend only on the variables PERSONcs and BUILDINGcs.

An attempt was made to simplify this process by declaring the carrier sets as constant variables, which would not require the function “cs”. However, this method led to further problems such as the inability of SPARK provers to prove correct code, so the method described in this section was used instead.

4.3.2 Translation of Constants

Constants require only one type of declaration, which is the declaration of a constant variable. The Event-B model is used to search for a list of constants, and the type environment of the model is used to get the type of each constant. Furthermore, as constant variable declarations in SPARK require that the variable be defined with a value the moment it is declared, an axiom defining the value of the constant is also needed.

In the running example, there exists the constant n, and the axiom:

Axm5: n = 20

The tool uses this axiom and the type environment to derive the type of constant n, which is an Integer, and its value, 20. The SPARK code generated shall be:

```
n : constant Integer := 20;
```

4.3.3 Translation of Axioms

Axioms are predicates. Hence, the predicate translation rules are used for this purpose. Note that the restrictions about the forms these predicates can take which were mentioned earlier will come into effect.

Furthermore, to reduce the number of unneeded axioms generated, axioms related to the finiteness or cardinality of carrier sets are ignored. As such, the only valid axiom in the running example is:

Axm5: n = 20

While this could be considered an unnecessary axiom, there could exist axioms containing equalities which are necessary in other models. As such, axioms such as these will not be ignored.

For every axiom, a function declaration is generated. In this case, the declaration is:

```
function Axm1 return Boolean is
    (n = 20) with
      Global => (n),
      Depends => (Axm1'Result => (n));
```

This function declaration will be used as a pre-condition and post-condition for every SPARK procedure to ensure that this axiom is maintained. As with the cs function, the Global and Depends aspects have the same roles.

If further valid axioms exist, further function declarations, such as Axm2, Axm3, Axm4 and so on will be generated. All these axioms would also then be in the pre- and post-conditions of every procedure.

4.3.4 Translation of Variables

To translate variables, all which is needed are the list of variables belonging to the machine, and the type environment of that model mapping each variable to its type. The use of the type environment means that the list of invariants do not have to be traversed to derive the required types.

From the running example, let's look at the variables person and permission. Using the model's type environment, which can be accessed from the statically checked machine root, we have access to the types of these variables (and every other variable in the model), which are as such:

- person : ℙ(PERSON)
- permission : ℙ(PERSON×BUILDING)

It can be seen that person is a set which contains elements of type PERSON, while permission is a relation which has a domain of elements of type PERSON and a range of elements of type BUILDING. For these variables, the translation tool generates the following lines of code:

```
person : set (PERSONtype);
permission : relation (PERSONtype,BUILDINGtype);
```

4.3.5 Translation of Invariants

Like axioms, invariants are also predicates, so the predicate translation rules come into play. Once again, the restrictions for predicates apply for invariants.

From the running example, three of the invariants are:

Inv1: person ⊆ PERSON
Inv2: partition(person,inside,outside)
Inv4: permission ∈ person ↔ BUILDING

Each predicate is translated into a function declaration. Other than the body of the function, which contains the actual predicate of the invariant, the Global and Depends aspects must be generated as well for each function. The global variables present in both aspects are identified by identifying all the free variables in the predicate. The following shows the translation of each invariant.

For **Inv1:** person ⊆ PERSON, the following SPARK function is generated:

```
function Inv1 return Boolean is
    (isSubset (person,PERSONcs)) with
      Global => (person,PERSONcs),
      Depends => (Inv1'Result => (person,PERSONcs));
```

The function “isSubset”, declared in the sr.ads file, is used for this invariant function. The Global clause indicates that only the global variables person and PERSONcs are involved in this function, and its result only depends on these two variables. Note that carrier sets will automatically have the ‘cs’ suffix appended to them when predicates are translated.

For **Inv2:** partition(person,inside,outside), the SPARK function for this invariant is:

```
function Inv2 return Boolean is
    (partition (person,inside,outside)) with
      Global => (person,inside,outside),
      Depends => (Inv2'Result => (person,inside,outside));
```

This time, the function uses the function “partition”, another function declared in sr.ads. Again, the Global and Depends aspects are written accordingly.

Inv4: permission ∈ person ↔ BUILDING translates to:

```
function Inv4 return Boolean is
    (relationOfSets (permission,person,BUILDINGcs)) with
      Global => (permission,person,BUILDINGcs),
      Depends => (Inv4'Result => (permission,person,BUILDINGcs));
```

All these functions will be post-conditions of all procedures, to ensure that these invariants are maintained by every procedure. These functions will also be pre-conditions of every procedure except the procedure representing the INITIALISATION event. Having these invariants as pre-conditions will help the SPARK provers prove the correctness of implementing code.

4.3.6 Translation of Events

Before showing the translation of each event, the general rules of event translation will be discussed. These rules apply to every event, with some differences for the INITIALISATION event.

Every event is translated into a SPARK procedure representing that event. As the aim of this project is to generate a SPARK specification file, the procedure generated for each event will be the specification of the procedure only, and not the body of the procedure which contains the actual implementation of the actions of the event. This specification includes the procedure’s name, the parameters of the procedure and their types, the pre-conditions and post-conditions of the procedure, as well as the Global and Depends aspects.

Firstly, the name of the procedure will be the same as the name of the event it represents. Secondly, the parameters of the procedure are the same as the parameters of the event. The types of these parameters can be found using the type environment for that specific event, which can be accessed using the statically checked machine root. These types can then be translated into SPARK the same way the types of variables were translated.

The pre-condition aspect of every event except the INITIALISATION event will contain the function cs, all the functions representing the axioms as well as all the functions representing the invariants. This is suitable as every event bar INITIALISATION should assume that the model is in a consistent state before

the event is triggered. The INITIALISATION event will only have the functions cs and the axiom functions in its Pre aspect, as it is assumed that it will be the first event triggered, meaning the variables would not have been set any values, and thus the invariants should not be involved at all in the pre-condition aspect.

Furthermore, the pre-condition aspect for every event should also contain predicates representing the guards of that event. Hence, each guard of the event needs to be translated using the predicate translation rules. Like the invariant translation, the carrier sets automatically have 'cs' appended to their names when the guards are translated. This is appropriate as guards are additional checks for events to occur, and are conceptually the same as pre-conditions.

The post-condition aspect of every event will include the function cs, all axiom functions and all invariant functions, as every event is expected to maintain all axioms and invariants, and the INITIALISATION event is expected to establish the invariants after being executed. The post-condition aspect shall also contain predicates derived from the actions of the event. This is done the following the way. Each action will be an assignment of one of three forms, where v is the variable which is assigned a new value by the action:

- v := w, where w is a constant value
- v := F(p), where p is a parameter, and F(p) is a new value for v which depends on p
- v := F(v,p), where F(v,p) depends on the initial value of v and the value p

In each case, the assignment operator := can be changed into the equality operator =, thus generating the predicate needed. However, in the case where v := F(v,p), the variable v on the right hand side needs to be changed into v'Old, as SPARK post-conditions require this syntax. Hence, the predicates are generated as such:

- v := w becomes v = w
- v := F(p) becomes v = F(p)
- v := F(v,p) becomes v = F(v'Old,p)

These predicates are then translated using the predicate translation rules, and placed in the post-condition aspect of the procedure. The same thing is done for every action in that event. This thus ensures that the procedure performs the actions of the event.

The Global aspect can contain several clauses - namely Proof_In, Input, Output and In_Out. The Proof_In clause contains global variables (constants, carrier sets and variables from the model) which are only used referenced in the

pre-condition aspect. Assuming that every global variable is involved in at least one axiom or invariant, the Proof_In clause will contain all those global variables not read or written to by the procedure. The Input clause contains global variables which only read by the procedure. The Output clause contains global variables which are only written to by the procedure. Finally, the In_Out clause contains global variables which are both read and written to.

The Depends aspect maps the dependencies of every global variable which is changed by the procedure. For every global variable which is written to by this procedure, this aspect indicates what the new value of the changed variable should depend on.

The reason the Global and Depends aspects are included is that the presence of the pre-condition and post-condition aspects is not sufficient to ensure no misbehaviour by the procedure. For example, if a procedure performed all its actions accordingly, and then also performed an extra action on another global variable which did not violate any invariants, it would successfully pass all post-condition checks despite performing this extra action. Hence, these extra aspects prevent a situation like this.

The first event in the running example is the INITIALISATION event:

INITIALISATION
THEN
    Act1: person ≔ ∅
    Act2: inside ≔ ∅
    Act3: outside ≔ ∅
    Act4: location ≔ ∅
    Act5: permission ≔ ∅
END

The procedure specification code generated for this event is:

```
procedure INITIALISATION with
       Pre => (cs and then Axm1),
       Post => (cs and then Axm1 and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then Inv5 and then
               isEmpty (person) and then
               isEmpty (inside) and then
               isEmpty (outside) and then
               isEmpty (location) and then
               isEmpty (permission)),
       Global => (Proof_In => (BUILDINGcs,PERSONcs,n),
              Output => (inside,location,outside,permission,person)),
       Depends => (person => null,outside => null,location => null,permission => null,inside
=> null);
```

The INITIALISATION event only has the function cs and the axiom functions as pre-conditions. The post-conditions are cs, the axioms and the invariants, as well as predicates representing actions, in this case setting all variables to the empty set. As the global variables BUILDINGcs, PERSONcs and n are used in the Pre aspect, they are present in the Proof_In clause. The remaining variables are all assigned values, but their initial values are not read, hence, they all belong in the Output clause. As each assigned variable's new value (the empty set) does not depend on any parameter or global variable, they have null dependencies in the Depends aspect.
The next event is the AddPerson event:

AddPerson
ANY
p
WHERE
Grd1: p ∈ PERSON ∖ person
THEN
Act1: person ≔ person ∪ {p}
Act2: outside ≔ outside ∪ {p}
END

The procedure generated is:

```
procedure AddPerson (p : in PERSONtype) with
       Pre => (cs and then Axm1 and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then Inv5 and then
               isMemberDifference (p,PERSONcs,person)),
       Post => (cs and then Axm1 and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then Inv5 and then
               equalsUnion (person,person'Old,p) and then
               equalsUnion (outside,outside'Old,p)),
       Global => (Proof_In => (BUILDINGcs,PERSONcs,n,inside,location,permission),
              In_Out => (outside,person)),
       Depends => (person =>+ (p),outside =>+ (p))
```

As the event takes a parameter p of type PERSON, this procedure takes a parameter p of type PERSONtype. The “in” keyword indicates that the value of the parameter is only read, and that the parameter is not assigned a new value. Like all events bar the INITIALISATION event, the cs, axiom and invariant functions are all pre-conditions. Furthermore, the guard p ∈ PERSON \ person is translated into the predicate isMemberDifference (p,PERSONcs,person), which is also a pre-condition. The cs, axiom and invariant functions are all post-conditions, along with the predicates for each action in the event. The only global variables used by this procedure are outside and person, so all other global variables are in the Proof_In clause as they are used in the Pre aspect. Person and outside are both read initially, and written to, so they are both in the In_Out clause. Each of these variables’ new values also depends on its initial value and the parameter p, and this is represented by the dependencies shown. The ‘+’ appended to the dependency sign “=>” indicates that the variable’s new value also depends on its old value, it is a shorthand to prevent writing the same variable name again on the right hand side.

The last event elaborated on in this section is:

AddPermission
ANY
p
b
WHERE
Grd1: p ∈ person
Grd2: b ∈ BUILDING
Grd3: p↦b ∉ permission
THEN
Act1: permission ≔ permission ∪ {p↦b}
END

The code generated for this event is:

```
procedure AddPermission (p : in PERSONtype;b : in BUILDINGtype) with
       Pre => (cs and then Axm1 and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then Inv5 and then
              person (p) and then
              BUILDINGcs (b) and then
              not (permission (p,b))),
       Post => (cs and then Axm1 and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then Inv5 and then
              equalsUnion (permission,permission'Old,p,b)),
       Global => (Proof_In => (BUILDINGcs,PERSONcs,n,inside,location,outside,person),
              In_Out => (permission)),
       Depends => (permission =>+ (p,b));
```

The three guard predicates are present in the Pre aspect, while the predicate derived from the action has also been translated into the Post aspect. Permission is the only global variable involved in this procedure, so it is in the In_Out clause, while the rest are in the Proof_In clause. The new value of permission is dependent on its initial value, as well as the parameters p and b, as specified in the Depends aspect.

The remaining events and their SPARK procedure counterparts are shown in the appendix.

4.4 Alternative Approaches

Two alternative approaches were considered. This section discusses the rudimentary stages of these approaches before the main approach using SPARK arrays was chosen.

4.4.1 Use of Functional Sets

One approach considered was the use of Ada.Containers.Functional_Sets (GNAT Reference Manual: Ada Containers Functional_Sets a-cofuse ads, n.d.). These data structures had advantages like being conceptually similar to sets and being compatible with SPARK. However, they were memory-inefficient, because every operation created a new set rather than changing the original set. Furthermore, the available set operations were limited, and they did not work well with the SPARK provers.

4.4.2 Use of Formal Ordered Sets

Another data structure considered was Ada.Containers.Formal_Ordered_Sets (GNAT Reference Manual: Ada Containers Formal_Ordered_Sets a-cforse ads, n.d.). These were also compatible with SPARK, however, there was difficulty using these to represent relations. Furthermore, operations on them were limited, and once again, these structures did not work well with the SPARK provers.

5. Implementation

After specifying the translation process and how it works on an Event-B model, a basic version of the tool was implemented as a Rodin plug-in.

The Eclipse IDE (Burnette, 2005) was used for the development of an Eclipse plug-in. The Rodin translation tool plug-in was developed as an Eclipse plug-in because the Rodin Platform is an Eclipse-based IDE (Event-B.org, n.d.), meaning that Eclipse plug-ins designed for Eclipse-based software can act as plug-ins for the Rodin Platform (Jastram, 2014).

5.1 Plug-In Development Environment Setup

The Plug-In Development Environment (Guindon, n.d.), or PDE, is a set of tools which aid the development of Eclipse plug-ins (Melhem and Glozic, 2003). PDE provides tools such as editors and launches which allows developers to easily develop and tailor their plug-ins.

This process first involved configuring Rodin as the target platform. This was done so the plug-in could be tested and debugged as a Rodin plug-in. Testing involved running an instance of the Rodin application.

The next step involved adding extensions - defining the added functionality contributed by the plug-in - thus designing how the plug-in would behave in the Rodin application.

The full details of this setup can be found in the appendix (Section A.2)

5.2 Code Generation Process

The following section describes how the translation process occurs, from the moment the user chooses an Event-B machine to translate, and the generation of the SPARK specification code in a predetermined hard-coded file.

5.2.1 Constructs Package

Before going into the translation process, it is worth touching on how the information extracted from an Event-B machine root is represented before the generation SPARK code. A package called constructs was created, with classes used to hold the information needed to generate each segment of the SPARK code.

As an example, from section 4.3.1 of this paper, it is shown that the information needed from every carrier set in the Event-B model is the name of the carrier set and its cardinality. The SPARK subtype used to represent the type of this carrier set will have the same name as the carrier set itself, but with “type” appended, while the actual SPARK set used to represent the carrier set will have the name of the carrier set with “cs” appended on. The subtype declaration also requires the cardinality of the carrier set. Hence, the class CarrierSet, defined in package constructs, will hold this information, from which SPARK code can be generated.

The same concept is then applied for the constants, variables, invariants and events. A class called Spec is used to represent an Event-B model. The Spec object holds a list of every type of construct.. For example, a Spec object can hold a list of CarrierSet objects, Constant objects, Variable objects, etc. Hence, it is this Spec object which is then used to make the relevant SPARK code.

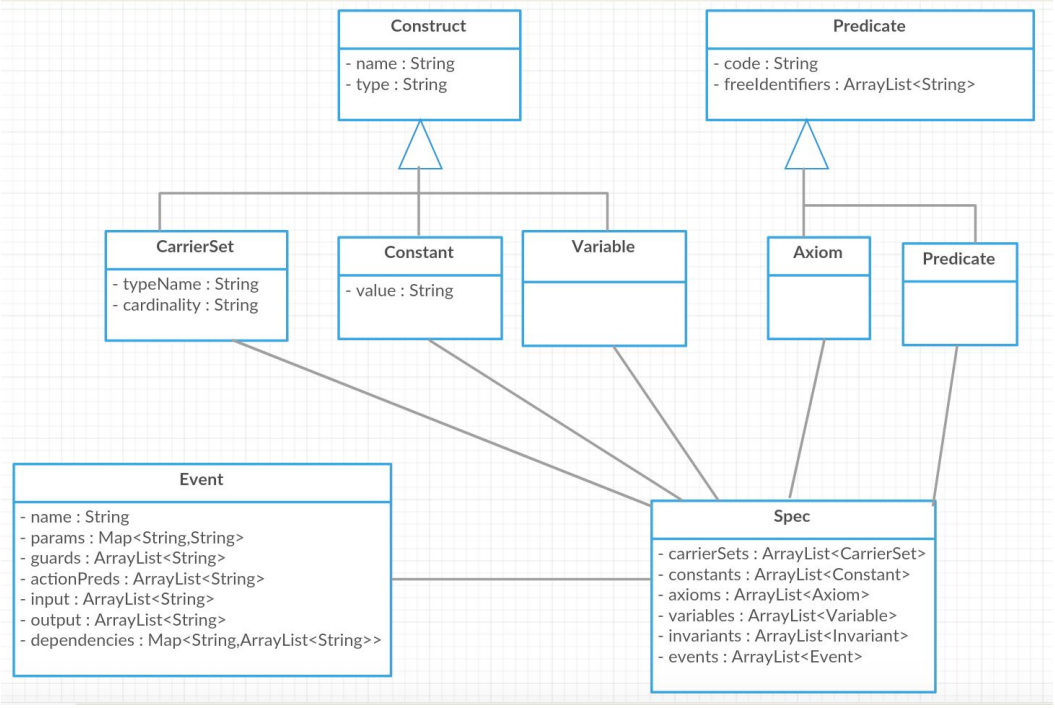

Figure 3: Class diagram of classes in constructs package, created using Creately - Online Diagram Editor - Try it Free (n.d.)

5.2.2 SparkTranslate Class

The SparkTranslate class extends the class AbstractHandler, and is the default handler for the developed plug-in. The class includes the method execute, executed when the "Translate to SPARK" command is chosen. It is this method which defines the behaviour of the plug-in.

The execute method unpacks the information needed from the selected machine root, such as constructs from the context and the machine. These constructs are represented as objects defined in the package org.eventb.core.

The SparkTranslate class contains several methods used by the execute method to extract the information from the org.eventb.core objects, and generate objects from the aforementioned constructs package.

For example, the getAxioms method takes an array of ISCAxiom objects, a class defined in org.eventb.core. The getAxioms method uses the information from these ISCAxiom objects to generate a list of Axiom objects, which hold the information needed to translate axioms. Equivalent methods for variables, invariants, events, constants and carrier sets are defined in this class.

5.2.3 Translate Method

An important method in the SparkTranslate class is the translate method, which is used to translate predicates from the Event-B constructs into predicates in SPARK. This method takes a Formula object, which is extracted from Event-B constructs such as axioms and invariants. This method also accepts an ArrayList of Strings, representing the names of all carrier sets in the model. This is only required so the translate method can append "cs" onto the end of every free identifier in the predicate which references a carrier set.

The translate method implements the list of translation rules described by checking the operator at the root of the predicate, as well as checking the operators at the root of the predicate's child or children. The list of translation rules implemented by this translate method is too long to explain how each rule is implemented by this method, so only a few examples will be explained here.

An example of a predicate to be translated can be e1 ∈ s1. This predicate signifies a single element e1 is a member of a single set s1. The translate method

detects the operator ∈ at the root of the predicate and directs the predicate to the method translateIn, which translates predicates with the ∈ operator at the root.

The translateIn method detects that both the left and right children are free identifiers, and hence returns the String “s1 (e1)”, the predicate in SPARK form. Note that s1 will have “cs” appended to it if it is a carrier set.

A more complex predicate is r1 = r2 {e1↦e2}. The translate method detects the = operator at the root of this predicate, and directs the predicate to the method translateEquals. This method detects the set difference operator ∖ on the right child of this predicate, and so generates the SPARK code “equalsDifference (r1,r2,e1,e2)”. Note here that the segment “e1,e2” is itself a translation of the expression {e1↦e2}. This means that a predicate of the form r1 = r2 ∖ r3, where r3 is a single relation, will return the String “equalsDifference(r1,r2,r3)”.

5.2.4 Translator Class

Translator is a class which contains a Spec object, which in turn contains lists of all the objects needed to make a SPARK specification file. A translator object is constructed using a Spec object, and has a method which generates the SPARK code based on the Spec object.

The code generator method uses the information in the Spec object, the only field of the Translator class, and applies the rules put forward in section 4.3 (Translation of Event-B Model) to generate the SPARK code needed.

Hence, these classes work together to generate the relevant SPARK code from the constructs in the Event-B model. Examples of how these classes work in tandem to achieve this can be found in the appendix (Section A.5).

6. Testing and Evaluation

6.1 JUnit Tests for the Translate Method

A key component of the translation implementation is the translate method, which translates Formula objects into Strings representing the predicate in SPARK code, using the function definitions in the sr.ads file. As such, JUnit (Massol and Husted, 2003) tests were written to test this method. These tests were written in the tests package.

6.1.1 Example - TestEquals

One of the test cases written tests the translation of predicates with the equals (=) operator at the root of the predicate. Within this test case, 15 separate test methods were written, testing different types of translation with the equals operator at the root. For example, one of the test methods written tests the predicate equating a set or relation to the union of two other sets or relations. By manually constructing the Formula objects, the tests assert that the translate method returns the expected Strings, as can be seen in the figure below.

6.1.2 Example - TestIn

The TestIn test case tests the translation of predicates with the membership (∈) operator at the root. 16 test methods were written for this test case. One of these methods tests the predicate with the powerset operator on the right hand side of the predicate, of the form s1 ∈ ℙ(s2), and is shown in Figure 5.

```
/**
 * Testing equality of unions, e.g. equalsUnion (s1,s2,s3)
 */
@Test
void test5() {

    ArrayList<String> carrierSets = new ArrayList<String>();

    try {

        /*
         *  f1: e1 ∈ s2 ∪ s3
            f2: e2 ∈ s2 ∪ {e1}
            f3: e1 ↦ e2 ∈ r2 ∪ r3
            f4: e3 ↦ e4 ∈ r2 ∪ {e1 ↦ e2}
         */

        Formula f1 = TestCenter.ff.makeRelationalPredicate(Formula.EQUAL, TestCenter.s1, TestCenter.s2unions3, null);
        Formula f2 = TestCenter.ff.makeRelationalPredicate(Formula.EQUAL, TestCenter.s1, TestCenter.s2unione1Set, null);
        Formula f3 = TestCenter.ff.makeRelationalPredicate(Formula.EQUAL, TestCenter.r1, TestCenter.r2unionr3, null);
        Formula f4 = TestCenter.ff.makeRelationalPredicate(Formula.EQUAL, TestCenter.r1, TestCenter.r2unione1mape2Set, null);

        System.out.println("Testing: " + f1.toString());
        System.out.println("Testing: " + f2.toString());
        System.out.println("Testing: " + f3.toString());
        System.out.println("Testing: " + f4.toString());

        System.out.println();

        assertEquals("equalsUnion (s1,s2,s3)",SparkTranslate.translate(f1,carrierSets));
        assertEquals("equalsUnion (s1,s2,e1)",SparkTranslate.translate(f2,carrierSets));
        assertEquals("equalsUnion (r1,r2,r3)",SparkTranslate.translate(f3,carrierSets));
        assertEquals("equalsUnion (r1,r2,e1,e2)",SparkTranslate.translate(f4,carrierSets));

    } catch (Exception e) {

        e.printStackTrace();

    }

}
```

Figure 4: Test method testing equalities involving the union operator in the test case TestEquals

```
/**
 * Testing membership of set in powerset of a set
 */
@Test
void test4() {

    ArrayList<String> carrierSets = new ArrayList<String>();

    try {

        /*
            f1: s1∈ℙ(s2)
            f2: r1∈ℙ(r2)
         */

        Formula f1 = TestCenter.ff.makeRelationalPredicate(Formula.IN, TestCenter.s1, TestCenter.pows2, null);
        Formula f2 = TestCenter.ff.makeRelationalPredicate(Formula.IN, TestCenter.r1, TestCenter.powr2, null);

        System.out.println("Testing: " + f1.toString());
        System.out.println("Testing: " + f2.toString());

        System.out.println();

        assertEquals("inPowerSet (s1,s2)",SparkTranslate.translate(f1,carrierSets));
        assertEquals("inPowerSet (r1,r2)",SparkTranslate.translate(f2,carrierSets));

    } catch (Exception e) {
        e.printStackTrace();
    }

}
```

Figure 5: Test method testing predicates involving membership in a powerset

6.2 Testing on Other Event-B Models

The implemented tool was tested with several models other than the running example. To save space in the main body of this report, the models and the generated SPARK code can be found in the appendix (Section A.6).

Due to time restrictions, a thorough formal test on how closely the SPARK code matches the models could not be performed and evaluated. However, testing this tool on other models shows that it can perform generally. The models tested are bound by the restrictions described throughout this report.

7. Project Management

7.1 Risk Assessment Table

Problem	Probability	Severity	Risk	Plan
Laptop damaged during Rodin plug-in development	1	4	4	Keep backup of work on memory stick or server
Difficulty learning and working with Rodin API	2	4	8	Seek guidance from supervisor who has experience with the Rodin API
Cannot adapt existing approach of using the Dafny contract generation to generating SPARK annotations as planned	2	5	10	Several alternative existing approaches have been researched and can be used, as described in the literature review
Rodin platform cannot be installed on current OS version	1	3	3	Testing and validation of tool on case studies can be done on lab PCs
Difficulty installing GPS IDE for SPARK	2	2	4	Test SPARK annotations and code on online editor

Table 1: Risk assessment table

7.2 Gantt Chart Comparison

This section compares the Gantt chart from the Progress Report, a planned schedule of remaining work to be completed, with the Gantt chart showing the actual progress which was made.

Figure 6: Gantt chart from Progress Report showing planned schedule for remaining work, made using Google Sheets (Google Sheets: Free Online Spreadsheets for Personal Use, n.d.)

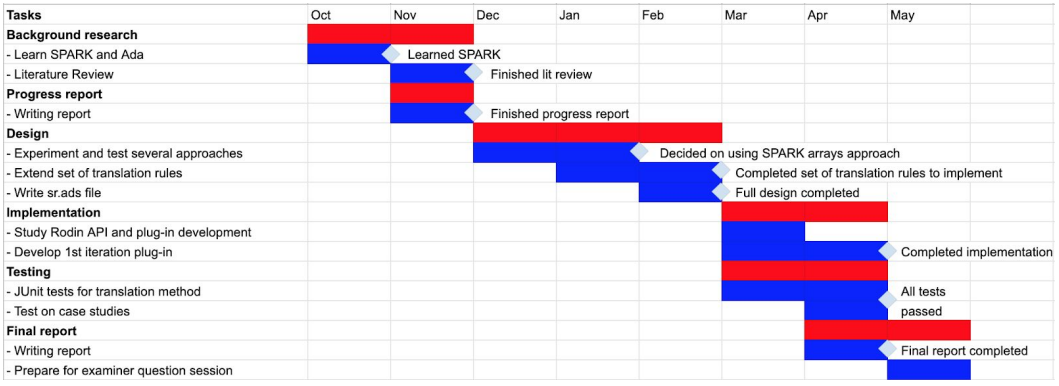

Figure 7: Gantt chart showing actual progress, made using Google Sheets (Google Sheets: Free Online Spreadsheets for Personal Use, n.d.)

There were a few deviations from the planned schedule. There was more difficulty than expected in deciding which approach to use, particularly how the model was to be represented in SPARK. The original plan was to adapt one of the existing approaches described in the literature review. However, due to the unique rules of Ada and SPARK and its type-safety, several approaches had to be experimented, such as the use of Ada Functional Sets and Ada Formal Ordered Sets. Finally, it was decided that the approach involving SPARK boolean arrays would be used. Furthermore, the extra step of designing and developing the sr.ads file was added, as it became clear that the existence of such a file would greatly improve the readability of the code to be generated. Due to the importance of the translate method, the decision was made to design JUnit tests specifically for that method.

It would be remiss to talk about any changes of plan undergone during this project without touching on the coronavirus pandemic of 2020 (World Health Organization, 2020). Being a mainly software-oriented project, this project was

not too badly affected by this pandemic. However, it did cause changes in the methods of evaluation for this project, namely the cancellation of in person viva sessions. Hence, the Gantt chart reflects this by changing the original task, “Study for viva”, into “Prepare for examiner question session”. Finally, not shown in the chart is the new additional task of filming a demonstration video showcasing the implementation and use of the translation tool.

7.3 Weekly Supervisor Meetings

In addition, weekly supervisor meetings also contributed to the management of this project. These meetings were not only used to discuss the technical details of the project, but also to gauge the overall project development process. Hence, they ensured that the project was always on-schedule.

8. Conclusion and Future Work

8.1 Summary and Justification of Work

Overall, the main goals of this project were achieved. This project puts forward an extensive list of translation rules from Event-B to SPARK. Furthermore, the work here can apply to more abstract Event-B models. While every effort was made to make the list of translation rules as exhaustive as possible within the available time frame, it is clear that there are still many forms of predicate possible in Event-B which cannot be translated by this implementation. However, given the richness of Event-B predicates, this is a reasonable approach, as most predicates in Event-B will take the forms present in the list of translation rules. Hence, the most common forms of predicates can be translated.

While seemingly inefficient, the use of SPARK boolean arrays to represent sets and relations is justified due to the difficulties in using Ada Functional Sets and Ada Formal Ordered Sets. As the goal of the translation process is to generate SPARK specification code which is then used to prove the correctness of SPARK code implementing that specification, it would seem counterintuitive to choose the approach which leads to situations where correct SPARK code cannot be proven to correctly implement the specification file due to the idiosyncrasies of these aforementioned alternative approaches. Hence, the use of SPARK arrays was the best approach for this purpose.

8.2 Future Work

The main aim of this project is the generation of SPARK code for a specification file, not code for a SPARK body file, which contains the code which implements the specification. However, there are conceivable ways in which this implementing code can also be generated in the translation process. This could possibly involve coming up with general rules for how actions in events can be interpreted and translated into code which actually performs these actions. However, the new set of translation rules will be completely separate from the set of translation rules derived for this project, as this translation will involve the

generation of SPARK code which performs the actions themselves, rather than SPARK code representing predicates.

In the process of generating implementing SPARK code, another path to explore in future work is the generation of loop invariants and variants needed to prove the correctness of any loops used to implement the specification. The process involved in this is perhaps less clear and obvious than the process above, but it is still an interesting area to explore.

Another obvious area for further work is extending the set of predicate translation rules even further. This may involve further additions to the sr.ads file, and hence make the translation tool more general.

Finally, the translation process can be adapted to work with Event-B machines which refine more abstract machines. While the exact mechanisms as to how this refinement relationship will be reflected in SPARK code is unclear, it is a potentially interesting area to explore.

Google Docs Word Count Tool: 9697 words

References

Abrial, J. (2010). *Modeling in Event-B: System and Software Engineering*. 1st ed. Cambridge University Press.

Hoang, T. (2013). An Introduction to the Event-B Modelling Method. *Industrial Deployment of System Engineering Methods*, pp.211-236.

Barnes, J. (2003). *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley.

Booch, G. and Bryan, D. (1993). *Software Engineering with Ada*. 3rd ed. Boston: Addison-Wesley Professional.

Murali, R. and Ireland, A. (2012). E-SPARK: Automated Generation of Provably Correct Code from Formally Verified Designs. *Electronic Communications of the EASST*, 53, pp.1-15.

Hoare, C. (1969). An axiomatic basis for computer programming. Communications of the ACM, 12(10), pp.576-580.

Gluck, P. and Holzmann, G., 2002. Using SPIN model checking for flight software verification. Proceedings, IEEE Aerospace Conference, p.1.

Abrial, J., Butler, M., Hallerstede, S., Hoang, T., Mehta, F. and Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6), pp.447-466.

Dalvandi, M. (2018). *Developing Verified Sequential Programs with Event-B*. Ph.D. University of Southampton.

Leino, K. (2010). Dafny: An Automatic Program Verifier for Functional Correctness. *In: Clarke E.M., Voronkov A. (eds) Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2010. Lecture Notes in Computer Science*, 6355, pp.348-370.

Dalvandi, M., Butler, M. and Rezazadeh, A. (2015). From Event-B Models to Dafny Code Contracts. *Fundamentals of Software Engineering. FSEN 2015. Lecture Notes in Computer Science*, 9392, pp.308-315. Springer, Cham.

Leavens, G., Baker, A. and Ruby, C. (1999). JML: A Notation for Detailed Design. In: *Kilov H., Rumpe B., Simmonds I. (eds) Behavioral Specifications of Businesses and Systems. The Springer International Series in Engineering and Computer Science, vol 523*. Springer, Boston, MA, pp.175-188.

Rivera, V. and Cataño, N. (2014). Translating Event-B to JML-specified Java programs. *Proceeding SAC '14 Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp.1264-1271.

Cataño, N., Leino, K. and Rivera, V. (2019). The EventB2Dafny Rodin plug-in. In: *2012 Second International Workshop on Developing Tools as Plug-Ins (TOPI)*. Zurich, Switzerland: IEEE.

Edmunds A., Butler M., Maamria I., Silva R., Lovell C. (2012) Event-B Code Generation: Type Extension with Theories. *In: Derrick J. et al. (eds) Abstract State Machines, Alloy, B, VDM, and Z. ABZ 2012. Lecture Notes in Computer Science*, 7316, pp.365-368. Springer, Berlin, Heidelberg.

Wright, S. (2009). Automatic Generation of C from Event-B. Workshop on integration of model-based formal methods and tools. Nantes, France: Springer-Verlag.

Butler, M., 2017. Reasoned modelling with Event-B. In: J. Bowen, Z. Liu and Z. Zhang, ed., Engineering Trustworthy Software Systems. Heidelberg, DE: Springer, pp.51-109.

Gcc.gnu.org. n.d. GNAT Reference Manual: Ada Containers Functional_Sets A-Cofuse Ads. [online] Available at: <https://gcc.gnu.org/onlinedocs/gcc-8.4.0/gnat_rm/Ada-Containers-Functional_005fSets-a-cofuse-ads.html> [Accessed 6 May 2020].

Gcc.gnu.org. n.d. GNAT Reference Manual: Ada Containers Formal_Ordered_Sets A-Cforse Ads. [online] Available at: <https://gcc.gnu.org/onlinedocs/gcc-7.5.0/gnat_rm/Ada-Containers-Formal_005fOrdered_005fSets-a-cforse-ads.html> [Accessed 6 May 2020].

Burnette, E., 2005. Eclipse IDE Pocket Guide: Using The Full-Featured IDE. O'Reilly Media, Inc.

Event-b.org. n.d. Event-B.Org. [online] Available at: <http://www.event-b.org/install.html> [Accessed 28 April 2020].

Jastram, M., 2014. Rodin User's Handbook. CREATESPACE.

Guindon, C., n.d. PDE | The Eclipse Foundation. [online] Eclipse.org. Available at: <https://www.eclipse.org/pde/> [Accessed 3 May 2020].

Melhem, W. and Glozic, D., 2003. PDE Does Plug-Ins. [online] Eclipse.org. Available at: <https://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html> [Accessed 3 May 2020].

Wiki.event-b.org. n.d. Using Rodin As Target Platform - Event-B. [online] Available at: <http://wiki.event-b.org/index.php/Using_Rodin_as_Target_Platform> [Accessed 3 May 2020].

Creately.com. n.d. Creately - Online Diagram Editor - Try It Free. [online] Available at: <https://creately.com/app/?tempID=h165rwt81&login_type=demo#> [Accessed 6 May 2020].

Massol, V. and Husted, T., 2003. Junit In Action. Manning Publications Co.

Google.com. n.d. Google Sheets: Free Online Spreadsheets For Personal Use. [online] Available at: <https://www.google.com/sheets/about/> [Accessed 6 May 2020].

World Health Organization, 2020. Coronavirus Disease 2019 (COVID-19): Situation Report, 72. [online] Available at: <https://apps.who.int/iris/bitstream/handle/10665/331685/nCoVsitrep01Apr2020-eng.pdf> [Accessed 4 May 2020].

Appendix

A.1 Full Running Example Model

A.1.1 Carrier Sets

Carrier Sets PERSON BUILDING

Each carrier set introduces a type. For example, PERSON denotes the type of an element able to register for the system, have permissions for buildings, enter buildings and so on.

A.1.2 Constants

Constants n

Admittedly, this constant does not serve much purpose in this model, other than to demonstrate the translation of constants. In reality, constants may be used in this scenario to model the maximum capacity of a building, or all buildings, but implementing this requirement for this system leads to difficulties for the SPARK provers to prove implemented SPARK code in the body file, despite not having any obvious mistakes or errors. This is due to the actual representations of sets and relations as SPARK arrays, which do not work well with invariants in the model relating to the cardinality of sets or relations, making the provers unable to prove perfectly correct implementations.

A.1.3 Axioms

Axioms

Axm1: finite(PERSON)
Axm2: finite(BUILDING)

These first two axioms declare the finiteness of the carrier sets, which are required to allow axioms which state the cardinality of the carrier sets. Stating the cardinality of the carrier sets is important, as the translation tool requires this information, as will be shown.

Axm3: card(PERSON) = 100
Axm4: card(BUILDING) = 5

As touched on above, these axioms state the cardinality of the carrier sets.

Axm5: n = 20

This axiom is stated so that the constant n has a type and value.

A.1.4 Variables

Variables person inside outside location permission

The variable person is the set of registered users of the system. Inside and outside are sets of registered users who are inside one of the buildings or outside. Location is a relation representing which building every 'inside' user is in at the moment, while permission is a relation representing the buildings that every registered user has access to.

A.1.5 Invariants

Inv1: person $\subseteq$ PERSON

This invariant describes the set of registered users, person, as a subset of PERSON. This indicates that elements in person have the type PERSON.

Inv2: partition(person,inside,outside)

The set person is partitioned into the sets inside and outside. The partition operator is actually a shorthand meant to describe the following formula:

partition(A,B,C) $\Leftrightarrow$ (A = B $\cup$ C) $\wedge$ (B $\cap$ C = $\varnothing$)

This invariant states that the union of inside and outside equals person, and that the intersection of inside and outside is empty. This indicates that all registered users are either inside or outside but not both, and that every element in inside and outside is a registered user.

Inv3: location ∈ inside → BUILDING

Location is a total function from the set inside to the carrier set BUILDING. This maps every user who is inside to one and only one building. Users who are outside will not have a location.

Inv4: permission ∈ person ↔ BUILDING

Permission is a relation from person to BUILDING. Each registered user can have permission to access 0 or more buildings.

Inv5: location ⊆ permission

Ensuring location is a subset of permission restricts the model so that any user being located at a building has the permission needed to be in that building.

A.1.6 Events

INITIALISATION
THEN
Act1: person ≔ ∅
Act2: inside ≔ ∅
Act3: outside ≔ ∅
Act4: location ≔ ∅
Act5: permission ≔ ∅
END

The initialisation event is meant to be the first event triggered. When first initialised, the system must not have any registered users. Hence, the person, inside and outside sets must be empty, along with the location and permission relations.

AddPerson
ANY
p
WHERE
Grd1: p ∈ PERSON ∖ person
THEN
Act1: person ≔ person ∪ {p}
Act2: outside ≔ outside ∪ {p}
END

This event registers a person into the access system. The guard ensures that the parameter p has the type PERSON, but is not already a registered user. The first action makes p a registered user. In keeping with invariant 2, p must either be inside or outside. As p will not have any permissions yet, p is added to outside.

AddPermission
ANY
p
b
WHERE
Grd1: p ∈ person
Grd2: b ∈ BUILDING
Grd3: p↦b ∉ permission
THEN
Act1: permission ≔ permission ∪ {p↦b}
END

This event grants access permission for a particular registered user for a specific building. The first two guards ensure p is a registered user and b is a building, and the third guard ensures this permission doesn't already exist. The action adds the tuple to permission.

Enter
ANY
p
b
WHERE
Grd1: p ∈ outside
Grd2: b ∈ BUILDING
Grd3: p↦b ∈ permission
THEN
Act1: outside ≔ outside ∖ {p}
Act2: inside ≔ inside ∪ {p}
Act3: location(p) ≔ b
END

This event allows a registered user to enter a building. The guards ensure that p is currently outside and b is a building, and that p has permission to enter b. The actions remove p from outside and add p to inside, signifying p's entrance. The location of p is set to b.

Leave
ANY
p
WHERE
Grd1: p ∈ inside
THEN
Act1: inside ≔ inside ∖ {p}
Act2: outside ≔ outside ∪ {p}
Act3: location ≔ {p} □ location
END

This event allows a user inside a building to exit. The guard ensures p is currently inside a building. The actions remove p from inside and add p to outside. The location tuple of p is removed from location using the domain subtraction operator as p is now outside.

RemovePermission
ANY
p
b
WHERE
Grd1: p ∈ person
Grd2: b ∈ BUILDING
Grd3: p↦b ∈ permission
Grd4: p ∈ inside ⇒ location(p) ≠ b
THEN
Act1: permission ≔ permission ∖ {p↦b}
END

This event removes a user's permission to access a particular building. The first and second guards ensure parameters p and b are a registered user and a building respectively. The third guard ensures that p has permission to access b. The fourth guard ensures that if p is currently inside, then p is not inside building b, as you cannot revoke permission for p to access b if p is currently in b. The action removes the tuple p↦b from permission.

RemoveUser
ANY
p
WHERE
Grd1: p ∈ outside
Grd2: p ∉ dom(permission)
THEN
Act1: outside ≔ outside ∖ {p}
Act2: person ≔ person ∖ {p}
END

This event unregisters a registered user. The guards ensure p is currently outside (as p should not be inside when being unregistered) and that p has no permissions for any building. The actions then remove p from outside and person.

A.2 Plug-In Development Environment Setup

The Plug-In Development Environment (Guindon, n.d.), or PDE, is a set of tools which aid the development of Eclipse plug-ins (Melhem and Glozic, 2003). PDE provides tools such as editors and launches which allows developers to easily develop and tailor their plug-ins.

A.2.1 Configuring the Rodin Target Platform

To develop the plug-in as a Rodin plug-in and test it, the target platform had to be configured in PDE to be the Rodin platform. This was done using the instructions at the wiki Using Rodin as Target Platform - Event-B (n.d.).

Setting the Rodin as the target platform also easily makes accessible the plug-ins required for the Rodin plug-in being developed. The following screenshot shows the Rodin platform set as the target platform.

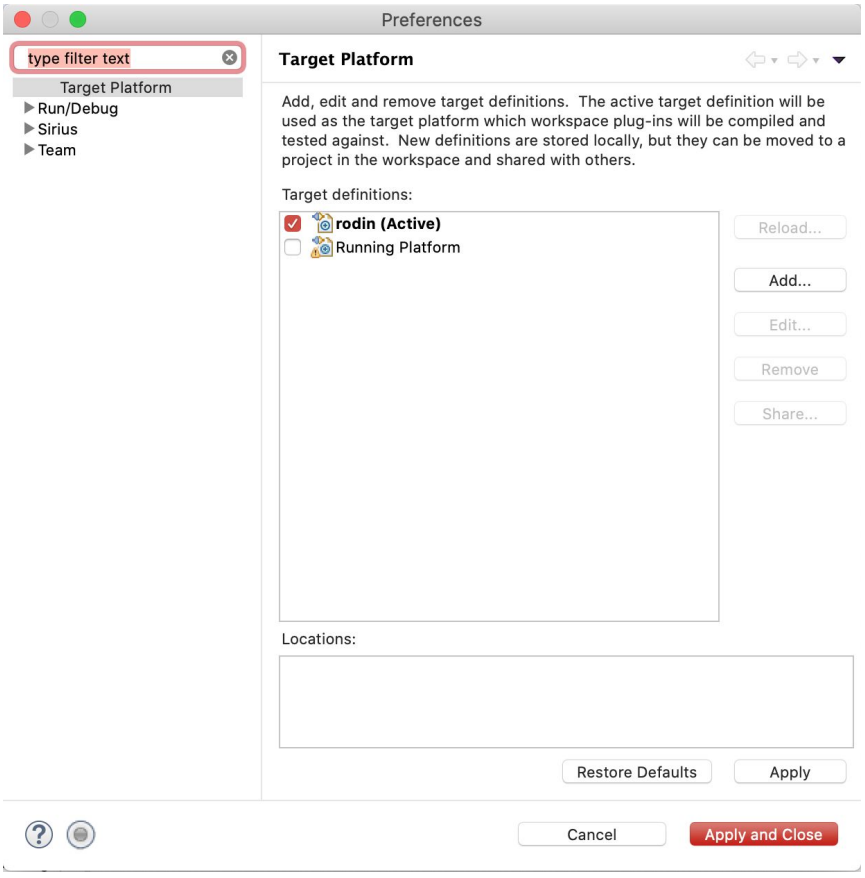

Figure 1: Setting the Rodin target platform

A.2.2 Plug-in Extensions

It was decided that the plug-in translation would be a command option on a popup menu. Furthermore, it was decided that the command option would only be enabled when the object being selected is an Event-B machine root. Plug-in extensions define the behaviour contributed to the platform by the plug-in, and so the extensions were set to perform this.

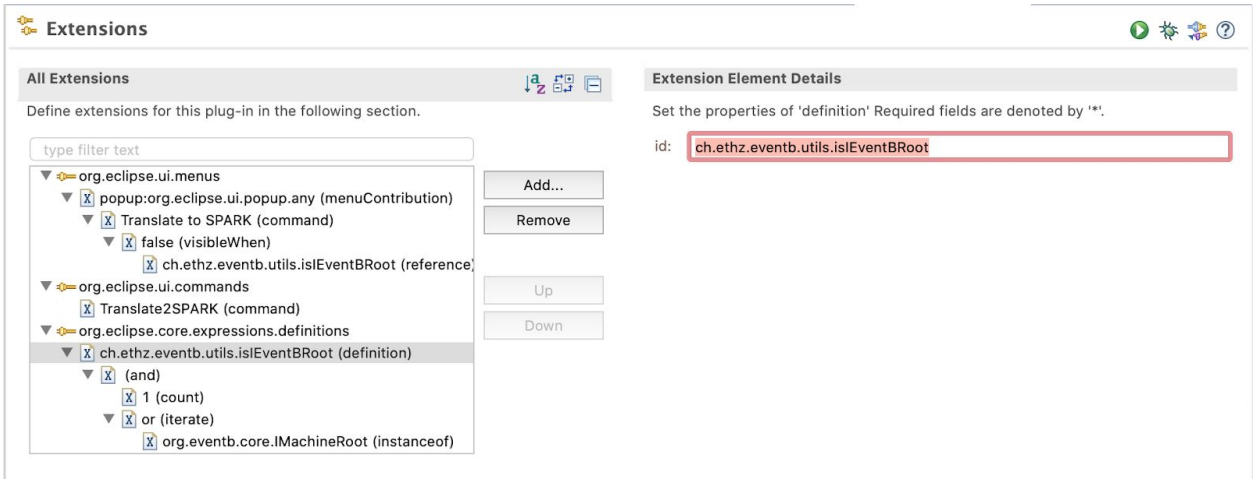

Figure 2: The extensions of this plug-in

A.3 Full Table of Predicate Translation Rules

Key:

- e1,e2,... represent single elements
- s1,s2,... represent sets
- r1,r2,... represent relations
- For each construct n in Event-B, EB2SPARK(n) is the representation of n in SPARK

Predicate in Event-B	Predicate in SPARK
s1 = s2	EB2SPARK(s1) = EB2SPARK(s2)
r1 = r2	EB2SPARK(r1) = EB2SPARK(r2)

e1 = e2	EB2SPARK(e1) = EB2SPARK(e2)
p1 = p2	EB2SPARK(p1) = EB2SPARK(p2)
e1 ∈ s1	EB2SPARK(s1) (EB2SPARK(e1))
e1 ∉ s1	not (isMember (EB2SPARK(e1),EB2SPARK(s1)))
e1↦e2 ∈ r1	EB2SPARK(r1) (EB2SPARK(e1),EB2SPARK(e2))
e1↦e2 ∉ r1	not (isMember (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1)))
p1 ⇒ p2	if (EB2SPARK(p1)) then (EB2SPARK(p2))
p1 ⇔ p2	if (EB2SPARK(p1)) then (EB2SPARK(p2)) else (not (EB2SPARK(p2)))
true	True
false	False
p1 ∧ p2	(EB2SPARK(p1)) and then (EB2SPARK(p2))
p1 ∨ p2	(EB2SPARK(p1)) or else (EB2SPARK(p2))
¬p1	not (EB2SPARK(p1))
s1 ≠ s2	EB2SPARK(s1) /= EB2SPARK(s2)
r1 ≠ r2	EB2SPARK(r1) /= EB2SPARK(r2)
e1 ≠ e2	EB2SPARK(e1) /= EB2SPARK(e2)
p1 ≠ p2	EB2SPARK(p1) /= EB2SPARK(p2)
∀z. p1 ⇒ p2	for all z in type => (if (EB2SPARK(p1)) then (EB2SPARK(p2)))*
∃z. p1 ∧ p2	for some z in type => (EB2SPARK(p1) and EB2SPARK(p2))*
e1 ∈ {e2}	EB2SPARK(e1) = EB2SPARK(e2)

e1 ∉ {e2}	EB2SPARK(e1) /= EB2SPARK(e2)
e1↦e2 ∈ {e3↦e4}	EB2SPARK(e1) = EB2SPARK(e3) and then EB2SPARK(e2) = EB2SPARK(e4)
e1↦e2 ∉ {e3↦e4}	not (EB2SPARK(e1) = EB2SPARK(e3) and then EB2SPARK(e2) = EB2SPARK(e4))

* the type derived is the type of z in SPARK form

s1 = {e1}	equalsSingleton (EB2SPARK(s1),EB2SPARK(e1))
s1 ≠ {e1}	not (equalsSingleton (EB2SPARK(s1),EB2SPARK(e1)))
r1 = {e1↦e2}	equalsSingleton (EB2SPARK(r1),EB2SPARK(e1),EB2SPARK(e2))
r1 ≠ {e1↦e2}	not (equalsSingleton (EB2SPARK(r1),EB2SPARK(e1),EB2SPARK(e2)))
s1 = ∅	isEmpty (EB2SPARK(s1))
s1 ≠ ∅	not (isEmpty (EB2SPARK(s1)))
r1 = ∅	isEmpty (EB2SPARK(r1))
r1 ≠ ∅	not (isEmpty (EB2SPARK(r1)))
e1↦e2 ∈ s1 x s2	inCartesianProduct (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(s1),EB2SPARK(s2))
e1↦e2 ∉ s1 × s2	not (inCartesianProduct (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(s1),EB2SPARK(s2)))
s1 ⊆ s2	isSubset (EB2SPARK(s1),EB2SPARK(s2))
s1 ⊈ s2	not (isSubset (EB2SPARK(s1),EB2SPARK(s2)))

r1 ⊆ r2	isSubset (EB2SPARK(r1),EB2SPARK(r2))
r1 ⊈ r2	not (isSubset (EB2SPARK(r1),EB2SPARK(r2)))
s1 ∈ ℙ(s2)	inPowerSet (EB2SPARK(s1),EB2SPARK(s2))
s1 ∉ ℙ(s2)	not (inPowerSet (EB2SPARK(s1),EB2SPARK(s2)))
r1 ∈ ℙ(r2)	inPowerSet (EB2SPARK(r1),EB2SPARK(r2))
r1 ∉ ℙ(r2)	not (inPowerSet (EB2SPARK(r1),EB2SPARK(r2)))
s1 ⊂ s2	isProperSubset (EB2SPARK(s1),EB2SPARK(s2))
s1 ⊄ s2	not (isProperSubset (EB2SPARK(s1),EB2SPARK(s2)))
r1 ⊂ r2	isProperSubset (EB2SPARK(r1),EB2SPARK(r2))
r1 ⊄ r2	not (isProperSubset (EB2SPARK(r1),EB2SPARK(r2)))
e1 ∈ s1 ∪ s2	isMemberUnion (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(s2))
e1 ∉ s1 ∪ s2	not (isMemberUnion (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(s2)))
e1 ∈ s1 ∪ {e2}	isMemberUnion (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(e2))
e1 ∉ s1 ∪ {e2}	not (isMemberUnion (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(e2)))
e1↦e2 ∈ r1 ∪ r2	isMemberUnion (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(r2))

e1↦e2 ∉ r1 ∪ r2	not (isMemberUnion (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(r2)))
e1↦e2 ∈ r1 ∪ {e3↦e4}	isMemberUnion (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3),EB2SP ARK(e4))
e1↦e2 ∉ r1 ∪ {e3↦e4}	not (isMemberUnion (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3),EB2SP ARK(e4)))
e1 ∈ s1 ∩ s2	isMemberIntersection (EB2SPARK(e1),EB2SPARK(s1),EB 2SPARK(s2))
e1 ∉ s1 ∩ s2	not (isMemberIntersection (EB2SPARK(e1),EB2SPARK(s1),EB 2SPARK(s2)))
e1 ∈ s1 ∩ {e2}	isMemberIntersection (EB2SPARK(e1),EB2SPARK(s1),EB 2SPARK(e2))
e1 ∉ s1 ∩ {e2}	not (isMemberIntersection (EB2SPARK(e1),EB2SPARK(s1),EB 2SPARK(e2)))
e1↦e2 ∈ r1 ∩ r2	isMemberIntersection (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(r2))
e1↦e2 ∉ r1 ∩ r2	not (isMemberIntersection (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(r2)))
e1↦e2 ∈ r1 ∩ {e3↦e4}	isMemberIntersection (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3),EB2SP ARK(e4))
e1↦e2 ∉ r1 ∩ {e3↦e4}	not (isMemberIntersection (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3),EB2SP ARK(e4)))

e1 ∈ s1 ∖ s2	isMemberDifference (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(s2))
e1 ∉ s1 ∖ s2	not (isMemberDifference (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(s2)))
e1 ∈ s1 ∖ {e2}	isMemberDifference (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(e2))
e1 ∉ s1 ∖ {e2}	not (isMemberDifference (EB2SPARK(e1),EB2SPARK(s1),EB2SPARK(e2)))
e1↦e2 ∈ r1 ∖ r2	isMemberDifference (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(r2))
e1↦e2 ∉ r1 ∖ r2	not (isMemberDifference (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(r2)))
e1↦e2 ∈ r1 ∖ {e3↦e4}	isMemberDifference (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(e3),EB2SPARK(e4))
e1↦e2 ∉ r1 ∖ {e3↦e4}	not (isMemberDifference (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(e3),EB2SPARK(e4)))
s1 = s2 ∖ s3	equalsDifference (EB2SPARK(s1),EB2SPARK(s2),EB2SPARK(s3))
s1 ≠ s2 ∖ s3	not (equalsDifference (EB2SPARK(s1),EB2SPARK(s2),EB2SPARK(s3)))
s1 = s2 ∖ {e1}	equalsDifference (EB2SPARK(s1),EB2SPARK(s2),EB2SPARK(e1))
s1 ≠ s2 ∖ {e1}	not (equalsDifference (EB2SPARK(s1),EB2SPARK(s2),EB2SPARK(e1)))

r1 = r2 ⧵ r3	equalsDifference (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(r3))
r1 ≠ r2 ⧵ r3	not (equalsDifference (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(r3)))
r1 = r2 ⧵ {e1⤇e2}	equalsDifference (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1),EB2SPARK(e2))
r1 ≠ r2 ⧵ {e1⤇e2}	not (equalsDifference (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1),EB2SPARK(e2)))
s1 = s2 ∪ s3	equalsUnion (EB2SPARK(s1),s2,'EB2SPARK(s3))
s1 ≠ s2 ∪ s3	not (equalsUnion (EB2SPARK(s1),s2,'EB2SPARK(s3)))
s1 = s2 ∪ {e1}	equalsUnion (EB2SPARK(s1),EB2SPARK(s2),EB 2SPARK(e1))
s1 ≠ s2 ∪ {e1}	not (equalsUnion (EB2SPARK(s1),EB2SPARK(s2),EB 2SPARK(e1)))
r1 = r2 ∪ r3	equalsUnion (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(r3))
r1 ≠ r2 ∪ r3	not (equalsUnion (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(r3)))
r1 = r2 ∪ {e1⤇e2}	equalsUnion (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1),EB2SPARK(e2))
r1 ≠ r2 ∪ {e1⤇e2}	not (equalsUnion (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1),EB2SPARK(e2)))

s1 = s2 ∩ s3	equalsIntersection (EB2SPARK(s1),EB2SPARK(s2),EB2SPARK(s3))
s1 ≠ s2 ∩ s3	not (equalsIntersection (EB2SPARK(s1),EB2SPARK(s2),EB2SPARK(s3)))
r1 = r2 ∩ r3	equalsIntersection (EB2SPARK(r1),EB2SPARK(r2),EB2SPARK(r3))
r1 ≠ r2 ∩ r3	not (equalsIntersection (EB2SPARK(r1),EB2SPARK(r2),EB2SPARK(r3)))
-	isFullSet (EB2SPARK(s1))**

** - used for carrier sets

e1 ∈ dom(r1)	inDomain (EB2SPARK(e1),EB2SPARK(r1))
e1 ∉ dom(r1)	not (inDomain (EB2SPARK(e1),EB2SPARK(r1)))
e1 ∈ ran(r1)	inRange (EB2SPARK(e1),EB2SPARK(r1))
e1 ∉ ran(r1)	not (inRange (EB2SPARK(e1),EB2SPARK(r1)))
e1↦e2 ∈ r1~	inConverse (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1))
e1↦e2 ∉ r1~	not (inConverse (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1)))
r1 = r2~	equalsConverse (EB2SPARK(r1),EB2SPARK(r2))
r1 ≠ r2~	not (equalsConverse (EB2SPARK(r1),EB2SPARK(r2)))
s1 = ran(r1)	setEqualsRange (EB2SPARK(s1),EB2SPARK(r1))

s1 ≠ ran(r1)	not (setEqualsRange (EB2SPARK(s1),EB2SPARK(r1)))
s1 = dom(r1)	setEqualsDomain (EB2SPARK(s1),EB2SPARK(r1))
s1 ≠ dom(r1)	not (setEqualsDomain (EB2SPARK(s1),EB2SPARK(r1)))
r1 ∈ s1 ↔ s2	relationOfSets (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 ↔→ s2	isPartialSurjectiveRelation (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 ←↔ s2	isTotalRelation (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 ↔↔ s2	isTotalSurjectiveRelation (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 □ s2	isPartialFunction (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 → s2	isTotalFunction (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 □ s2	isPartialInjection (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 ↣ s2	isTotalInjection (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 □ s2	isPartialSurjection (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
r1 ∈ s1 ↠ s2	isTotalSurjection (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))

r1 ∈ s1 □ s2	isBijection (EB2SPARK(r1),EB2SPARK(s1),EB2SPARK(s2))
e1↦e2 ∈ s1 ◁ r1	inDomainRestriction (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(s1))
e1↦e2 ∉ s1 ◁ r1	not (inDomainRestriction (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(s1)))
e1↦e2 ∈ {e3} ◁ r1	inDomainRestriction (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(e3))
e1↦e2 ∉ {e3} ◁ r1	not (inDomainRestriction (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(e3)))
r1 = s1 ◁ r2	equalsDomainRestriction (EB2SPARK(r1),EB2SPARK(r2),EB2SPARK(s1))
r1 ≠ s1 ◁ r2	not (equalsDomainRestriction (EB2SPARK(r1),EB2SPARK(r2),EB2SPARK(s1)))
r1 = {e1} ◁ r2	equalsDomainRestriction (EB2SPARK(r1),EB2SPARK(r2),EB2SPARK(e1))
r1 ≠ {e1} ◁ r2	not (equalsDomainRestriction (EB2SPARK(r1),EB2SPARK(r2),EB2SPARK(e1)))
e1↦e2 ∈ s1 □ r1	inDomainSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(s1))
e1↦e2 ∉ s1 □ r1	not (inDomainSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(s1)))
e1↦e2 ∈ {e3} □ r1	inDomainSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB2SPARK(r1),EB2SPARK(e3))

e1↦e2 ∉ {e3} ⩤ r1	not (inDomainSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3)))
r1 = s1 ⩤ r2	equalsDomainSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(s1))
r1 ≠ s1 ⩤ r2	not (equalsDomainSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(s1)))
r1 = {e1} ⩤ r2	equalsDomainSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1))
r1 ≠ {e1} ⩤ r2	not (equalsDomainSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1))
e1↦e2 ∈ r1 ▷ s1	inRangeRestriction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(s1))
e1↦e2 ∉ r1 ▷ s1	not (inRangeRestriction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(s1)))
e1↦e2 ∈ r1 ▷ {e3}	inRangeRestriction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3))
e1↦e2 ∉ r1 ▷ {e3}	not (inRangeRestriction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3)))
r1 = r2 ▷ s1	equalsRangeRestriction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(s1))
r1 ≠ r2 ▷ s1	not (equalsRangeRestriction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(s1)))
r1 = r2 ▷ {e1}	equalsRangeRestriction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1))

r1 ≠ r2 ▷ {e1}	not (equalsRangeRestriction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1)))
e1↦e2 ∈ r1 □ s1	inRangeSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(s1))
e1↦e2 ∉ r1 □ s1	not (inRangeSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(s1)))
e1↦e2 ∈ r1 □ {e3}	inRangeSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3))
e1↦e2 ∉ r1 □ {e3}	not (inRangeSubtraction (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(e3)))
r1 = r2 □ s1	equalsRangeSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(s1))
r1 ≠ r2 □ s1	not (equalsRangeSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(s1)))
r1 = r2 □ {e1}	equalsRangeSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1))
r1 ≠ r2 □ {e1}	not (equalsRangeSubtraction (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(e1)))
e1 ∈ r1[s1]	inRelationalImage (EB2SPARK(e1),EB2SPARK(r1),EB 2SPARK(s1))
e1 ∉ r1[s1]	not (inRelationalImage (EB2SPARK(e1),EB2SPARK(r1),EB 2SPARK(s1)))
e1 ∈ r1[{e2}]	inRelationalImage (EB2SPARK(e1),EB2SPARK(r1),EB 2SPARK(e2))

e1 ∉ r1[{e2}]	not (inRelationalImage (EB2SPARK(e1),EB2SPARK(r1),EB 2SPARK(e2)))
s1 = r1[s2]	equalsRelationalImage (EB2SPARK(s1),EB2SPARK(r1),EB 2SPARK(s2))
s1 ≠ r1[s2]	not (equalsRelationalImage (EB2SPARK(s1),EB2SPARK(r1),EB 2SPARK(s2)))
s1 = r1[{e1}]	equalsRelationalImage (EB2SPARK(s1),EB2SPARK(r1),EB 2SPARK(e1))
s1 ≠ r1[{e1}]	not (equalsRelationalImage (EB2SPARK(s1),EB2SPARK(r1),EB 2SPARK(e1)))
{e1↦e2} ∈ r1;r2	inComposition (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(r2))
e1↦e2 ∉ r1;r2	not (inComposition (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(r2)))
e1↦e2 ∈ r1 r2	inOverride (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(r2))
e1↦e2 ∉ r1 r2	not (inOverride (EB2SPARK(e1),EB2SPARK(e2),EB 2SPARK(r1),EB2SPARK(r2)))
s1 ∪ s2 = ∅	unionEmpty (EB2SPARK(s1),EB2SPARK(s2))
s1 ∪ s2 ≠ ∅	not (unionEmpty (EB2SPARK(s1),EB2SPARK(s2)))
s1 ∩ s2 = ∅	intersectionEmpty (EB2SPARK(s1),EB2SPARK(s2))
s1 ∩ s2 ≠ ∅	not (intersectionEmpty (EB2SPARK(s1),EB2SPARK(s2)))

partition(s1,s2,s3)	partition (EB2SPARK(s1),EB2SPARK(s2),EB 2SPARK(s3))
r1 ∪ r2 = ∅	unionEmpty (EB2SPARK(r1),EB2SPARK(r2))
r1 ∪ r2 ≠ ∅	not (unionEmpty (EB2SPARK(r1),EB2SPARK(r2)))
r1 ∩ r2 = ∅	intersectionEmpty (EB2SPARK(r1),EB2SPARK(r2))
r1 ∩ r2 ≠ ∅	not (intersectionEmpty (EB2SPARK(r1),EB2SPARK(r2)))
partition(r1,r2,r3)	partition (EB2SPARK(r1),EB2SPARK(r2),EB 2SPARK(r3))
r1(e1) = r2(e2)	functionApplicationEquality (EB2SPARK(r1),EB2SPARK(e1),EB 2SPARK(r2),EB2SPARK(e2))
r1(e1) ≠ r2(e2)	not (functionApplicationEquality (EB2SPARK(r1),EB2SPARK(e1),EB 2SPARK(r2),EB2SPARK(e2)))
e1 = r1(e2)	EB2SPARK(r1) (EB2SPARK(e2),EB2SPARK(e1))
e1 ≠ r1(e2)	not (EB2SPARK(r1) (EB2SPARK(e2),EB2SPARK(e1)))
r1(e1) = e2	EB2SPARK(r1) (EB2SPARK(e1),EB2SPARK(e2))
r1(e1) ≠ e2	not (EB2SPARK(r1) (EB2SPARK(e1),EB2SPARK(e2)))

A.4 Declarations in sr.ads

```
type set is array (Integer range <>) of Boolean;

type relation is array (Integer range <>, Integer range <>) of Boolean;
```

```
   function isMember (e : Integer; s : set) return Boolean is
     (s (e)) with
       Pre => (for some x in s'Range => (x = e));

   function isMember (e : Integer; f : Integer; r : relation) return Boolean
is
     (r (e,f)) with
       Pre => (for some x in r'Range (1) => (for some y in r'Range (2) => (x
= e and then y = f))),
       Post => (isMember'Result = r (e,f));

   function relationOfSets (r : relation; s : set; t : set) return Boolean is
     (for all x in r'Range (1) => (for all y in r'Range (2) => (if (r (x,y))
then ((isMember (x,s)) and then (isMember (y,t)))))) with
       Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x =
y))) and
                (for all x in r'Range (2) => (for some y in t'Range => (x =
y))));

   function equalsSingleton (s : set; e : Integer) return Boolean is
     (for all x in s'Range => (if isMember (x,s) then (x = e) else (x /=
e)));

   function equalsSingleton (r : relation; e : Integer; f : Integer) return
Boolean is
      (for all x in r'Range (1) => (for all y in r'Range (2) => (if isMember
(x,y,r) then (x = e and y = f) else (not (x = e and y = f)))));

   function isEmpty (s : set) return Boolean is
     (for all x in s'Range => (not (isMember (x,s))));

   function isEmpty (r : relation) return Boolean is
     (for all x in r'Range (1) => (for all y in r'Range (2) => (not (isMember
(x,y,r)))));

   function inCartesianProduct (u : Integer; v : Integer; s : set; t : set)
return Boolean is
     (isMember (u,s) and isMember (v,t))
       with
         Pre => ((for some x in s'Range => (x = u)) and
                 (for some x in t'Range => (x = v)));

   function isSubset (s1 : set; s2 : set) return Boolean is
     (for all x in s1'Range => (if isMember (x,s1) then isMember (x,s2)))
     with
       Pre => (for all x in s1'Range => (for some y in s2'Range => (x = y)));

   function isSubset (r1 : relation; r2 : relation) return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (isMember (x,y,r2))))) with
       Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
```

```
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));



   function inPowerSet (s : set; t : set) return Boolean is
     (isSubset (s, t))
       with
         Pre => (for all x in s'Range => (for some y in t'Range => (x = y)));

   function inPowerSet (r1 : relation; r2 : relation) return Boolean is
     (isSubset (r1, r2))
       with
         Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));


   function isProperSubset (s : set; t : set) return Boolean is
     (isSubset (s, t) and then s /= t)
       with
         Pre => (for all x in s'Range => (for some y in t'Range => (x = y)));

   function isProperSubset (r1 : relation; r2 : relation) return Boolean is
     (isSubset (r1, r2) and then r1 /= r2) with
       Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function isMemberUnion (e : Integer; s : set; t : set) return Boolean is
     (isMember (e,s) or else isMember (e,t))
       with
         Pre => ((for some x in s'Range => (x = e)) and
                 (for some x in t'Range => (x = e)));

   function isMemberUnion (e : Integer; s : set; f : Integer) return Boolean
is
     (isMember (e,s) or else e = f)
   with
     Pre => (for some x in s'Range => (x = e));

   function isMemberUnion (e : Integer; f : Integer; s : relation; t :
relation) return Boolean is
     (isMember (e,f,s) or else isMember (e,f,t))
       with
         Pre => ((for some x in s'Range (1) => (x = e)) and
                 (for some x in t'Range (1) => (x = e)) and
                 (for some x in s'Range (2) => (x = f)) and
                 (for some x in t'Range (2) => (x = f)));

   function isMemberUnion (e : Integer; f : Integer; s : relation; h :
```

```
Integer; i : Integer) return Boolean is
     (isMember (e,f,s) or else (e = h and then f = i))
       with
         Pre => ((for some x in s'Range (1) => (x = e)) and
                 (for some x in s'Range (2) => (x = f)));

   function isMemberIntersection (e : Integer; s : set; t : set) return
Boolean is
     (isMember (e,s) and then isMember (e,t))
       with
         Pre => ((for some x in s'Range => (x = e)) and
                   (for some x in t'Range => (x = e)));

   function isMemberIntersection (e : Integer; s : set; f : Integer) return
Boolean is
     (isMember (e,s) and then e = f)
   with
     Pre => (for some x in s'Range => (x = e));


   function isMemberIntersection (e : Integer; f : Integer; s : relation; t :
relation) return Boolean is
     (isMember (e,f,s) and then isMember (e,f,t))
       with
         Pre => ((for some x in s'Range (1) => (x = e)) and
                 (for some x in t'Range (1) => (x = e)) and
                 (for some x in s'Range (2) => (x = f)) and
                 (for some x in t'Range (2) => (x = f)));

   function isMemberIntersection (e : Integer; f : Integer; s : relation; h :
Integer; i : Integer) return Boolean is
     (isMember (e,f,s) and then (e = h and f = i))
       with
         Pre => ((for some x in s'Range (1) => (x = e)) and
                 (for some x in s'Range (2) => (x = f)));

   function isMemberDifference (e : Integer; s : set; t : set) return Boolean
is
     (isMember (e,s) and then not (isMember (e,t)))
   with
     Pre => ((for some x in s'Range => (x = e)) and
              (for some x in t'Range => (x = e)));

   function isMemberDifference (e : Integer; s : set; f : Integer) return
Boolean is
     (isMember (e,s) and then e /= f)
   with
     Pre => (for some x in s'Range => (x = e));

   function isMemberDifference (e : Integer; f : Integer; s : relation; t :
relation) return Boolean is
     (isMember (e,f,s) and then not (isMember (e,f,t)))
       with
```

```
        Pre => ((for some x in s'Range (1) => (x = e)) and
                (for some x in t'Range (1) => (x = e)) and
                (for some x in s'Range (2) => (x = f)) and
                (for some x in t'Range (2) => (x = f)));


   function isMemberDifference (e : Integer; f : Integer; s : relation; h :
Integer; i : Integer) return Boolean is
     (isMember (e,f,s) and then not (e = h and then f = i))
       with
         Pre => ((for some x in s'Range (1) => (x = e)) and
                 (for some x in s'Range (2) => (x = f)));

   function equalsDifference (a : set; b : set; c : set) return Boolean is
     (for all x in a'Range => (if isMember (x,a) then (isMemberDifference
(x,b,c)) else (not (isMemberDifference (x,b,c)))))
     with
       Pre => ((for all x in a'Range => (for some y in b'Range => (x = y)))
and
                 (for all x in a'Range => (for some y in c'Range => (x =
y))));

   function equalsDifference (a : set; b : set; u : Integer) return Boolean
is
     (for all x in a'Range => (if isMember (x,a) then (isMemberDifference
(x,b,u)) else (not (isMemberDifference (x,b,u)))))
     with
       Pre => ((for all x in a'Range => (for some y in b'Range => (x = y)))
and
                (for some x in a'Range => (x = u)));

   function equalsDifference (a : relation; b : relation; c : relation)
return Boolean is
     (for all x in a'Range (1) => (for all y in a'Range (2) => (if isMember
(x,y,a) then (isMemberDifference (x,y,b,c)) else (not (isMemberDifference
(x,y,b,c))))))
     with
       Pre => ((for all x in a'Range (1) => (for some y in b'Range (1) => (x
= y))) and
                (for all x in a'Range (2) => (for some y in b'Range (2) => (x
= y))) and
                (for all x in a'Range (1) => (for some y in c'Range (1) => (x
= y))) and
                (for all x in a'Range (2) => (for some y in c'Range (2) => (x
= y))));

   function equalsDifference (a : relation; b : relation; u : Integer; v :
Integer) return Boolean is
     (for all x in a'Range (1) => (for all y in a'Range (2) => (if isMember
(x,y,a) then (isMemberDifference (x,y,b,u,v)) else (not (isMemberDifference
(x,y,b,u,v))))))
     with
       Pre => ((for all x in a'Range (1) => (for some y in b'Range (1) => (x
```

```
= y))) and
               (for all x in a'Range (2) => (for some y in b'Range (2) => (x
= y))) and
               (for some x in a'Range (1) => (x = u)) and
               (for some x in a'Range (2) => (x = v)));

   function equalsUnion (a : set; b : set; c : set) return Boolean is
     (for all x in a'Range => (if isMember (x,a) then (isMemberUnion (x,b,c))
else (not (isMemberUnion (x,b,c)))))
     with
       Pre => ((for all x in a'Range => (for some y in b'Range => (x = y)))
and
               (for all x in a'Range => (for some y in c'Range => (x = y))));



   function equalsUnion (a : set; b : set; u : Integer) return Boolean is
     (for all x in a'Range => (if isMember (x,a) then (isMemberUnion (x,b,u))
else (not (isMemberUnion (x,b,u)))))
     with
       Pre => ((for all x in a'Range => (for some y in b'Range => (x = y)))
and
               (for some x in a'Range => (x = u)));

   function equalsUnion (a : relation; b : relation; c : relation) return
Boolean is
     (for all x in a'Range (1) => (for all y in a'Range (2) => (if isMember
(x,y,a) then (isMemberUnion (x,y,b,c)) else (not (isMemberUnion
(x,y,b,c))))))
     with
       Pre => ((for all x in a'Range (1) => (for some y in b'Range (1) => (x
= y))) and
               (for all x in a'Range (2) => (for some y in b'Range (2) => (x
= y))) and
               (for all x in a'Range (1) => (for some y in c'Range (1) => (x
= y))) and
               (for all x in a'Range (2) => (for some y in c'Range (2) => (x
= y))));

   function equalsUnion (a : relation; b : relation; u : Integer; v :
Integer) return Boolean is
     (for all x in a'Range (1) => (for all y in a'Range (2) => (if isMember
(x,y,a) then (isMemberUnion (x,y,b,u,v)) else (not (isMemberUnion
(x,y,b,u,v))))))
     with
       Pre => ((for all x in a'Range (1) => (for some y in b'Range (1) => (x
= y))) and
               (for all x in a'Range (2) => (for some y in b'Range (2) => (x
= y))) and
               (for some x in a'Range (1) => (x = u)) and
               (for some x in a'Range (2) => (x = v)));

   function equalsIntersection (a : set; b : set; c : set) return Boolean is
```

```
     (for all x in a'Range => (if isMember (x,a) then (isMemberIntersection
(x,b,c)) else (not (isMemberIntersection (x,b,c)))))
     with
       Pre => ((for all x in a'Range => (for some y in b'Range => (x = y)))
and
                 (for all x in a'Range => (for some y in c'Range => (x =
y))));

   function equalsIntersection (a : relation; b : relation; c : relation)
return Boolean is
     (for all x in a'Range (1) => (for all y in a'Range (2) => (if isMember
(x,y,a) then (isMemberIntersection (x,y,b,c)) else (not (isMemberIntersection
(x,y,b,c))))))
     with
       Pre => ((for all x in a'Range (1) => (for some y in b'Range (1) => (x
= y))) and
               (for all x in a'Range (2) => (for some y in b'Range (2) => (x
= y))) and
               (for all x in a'Range (1) => (for some y in c'Range (1) => (x
= y))) and
               (for all x in a'Range (2) => (for some y in c'Range (2) => (x
= y))));

   function isFullSet (s : set) return Boolean is
      (for all x in s'Range => (isMember (x,s)));

   function inDomain (u : Integer; r : relation) return Boolean is
     (for some y in r'Range (2) => (r (u,y)))
       with
         Pre => (for some x in r'Range (1) => (x = u));
   function inRange (v : Integer; r : relation) return Boolean is
     (for some x in r'Range (1) => (isMember (x,v,r)))
       with
         Pre => (for some y in r'Range (2) => (y = v));

   function inConverse (e : Integer; f : Integer; r : relation) return
Boolean is
     (isMember (f,e,r))
       with
         Pre => ((for some x in r'Range (1) => (x = f)) and
                 (for some x in r'Range (2) => (x = e)));

   function equalsConverse (r1 : relation; r2 : relation) return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inConverse (x,y,r2)) else (not (inConverse (x,y,r2)))))) with
       Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (2) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (1) =>
(x = y))));

   function setEqualsRange (s : set; r : relation) return Boolean is
     (for all x in s'Range => (if isMember (x,s) then (inRange (x, r)) else
(not (inRange (x, r)))))
```

```
     with
       Pre => (for all x in s'Range => (for some y in r'Range (2) => (x =
y)));

   function setEqualsDomain (s : set; r : relation) return Boolean is
     (for all x in s'Range => (if isMember (x,s) then (inDomain (x, r)) else
(not (inDomain (x, r)))))
     with
       Pre => (for all x in s'Range => (for some y in r'Range (1) => (x =
y)));


   function isPartialSurjectiveRelation (r : relation; s : set; t : set)
return Boolean is
     (relationOfSets (r,s,t) and then setEqualsRange (t,r))
       with
         Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x =
y))) and
                 (for all x in r'Range (2) => (for some y in t'Range => (x =
y))) and
                 (for all x in t'Range => (for some y in r'Range (2) => (x =
y))));

   function isTotalRelation (r : relation; s : set; t : set) return Boolean
is
     (relationOfSets (r,s,t) and then setEqualsDomain (s,r))
       with
         Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x =
y))) and
                 (for all x in r'Range (2) => (for some y in t'Range => (x =
y))) and
                 (for all x in s'Range => (for some y in r'Range (1) => (x =
y))));









   function isTotalSurjectiveRelation (r : relation; s : set; t : set) return
Boolean is
     (relationOfSets (r,s,t) and then setEqualsDomain (s,r) and then
setEqualsRange (t,r))
       with
         Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x =
y))) and
                 (for all x in r'Range (2) => (for some y in t'Range => (x =
y))) and
                 (for all x in s'Range => (for some y in r'Range (1) => (x =
y))) and
                 (for all x in t'Range => (for some y in r'Range (2) => (x =
y))));
```

```
   function inDomainRestriction (e : Integer; f : Integer; r : relation; s :
set) return Boolean is
     (isMember (e,s) and then r (e,f)) with
       Pre => ((for some x in s'Range => (x = e)) and
               (for some x in r'Range (1) => (x = e)) and
               (for some x in r'Range (2) => (x = f)));

   function inDomainRestriction (e : Integer; f : Integer; r : relation; g :
Integer) return Boolean is
     (e = g and then isMember (e,f,r)) with
       Pre => ((for some x in r'Range (1) => (x = e)) and
               (for some x in r'Range (2) => (x = f)));

   function equalsDomainRestriction (r1 : relation; r2 : relation; s : set)
return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inDomainRestriction (x,y,r2,s)) else (not (inDomainRestriction
(x,y,r2,s)))))) with
       Pre => ((for all x in r1'Range (1) => (for some y in s'Range => (x =
y))) and
               (for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function equalsDomainRestriction (r1 : relation; r2 : relation; e :
Integer) return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inDomainRestriction (x,y,r2,e)) else (not (inDomainRestriction
(x,y,r2,e)))))) with
       Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function inDomainSubtraction (e : Integer; f : Integer; r : relation; s :
set) return Boolean is
     ((not isMember (e,s)) and then isMember (e,f,r)) with
       Pre => ((for some x in s'Range => (x = e)) and
               (for some x in r'Range (1) => (x = e)) and
                   (for some x in r'Range (2) => (x = f)));

   function inDomainSubtraction (e : Integer; f : Integer; r : relation; g :
Integer) return Boolean is
     ((e /= g) and then isMember (e,f,r)) with
       Pre => ((for some x in r'Range (1) => (x = e)) and
               (for some x in r'Range (2) => (x = f)));

   function equalsDomainSubtraction (r1 : relation; r2 : relation; s : set)
return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inDomainSubtraction (x,y,r2,s)) else (not (inDomainSubtraction
(x,y,r2,s)))))) with
```

```
      Pre => ((for all x in r1'Range (1) => (for some y in s'Range => (x =
y))) and
              (for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
              (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function equalsDomainSubtraction (r1 : relation; r2 : relation; e :
Integer) return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inDomainSubtraction (x,y,r2,e)) else (not (inDomainSubtraction
(x,y,r2,e)))))) with
      Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
              (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function inRangeRestriction (e : Integer; f : Integer; r : relation; t :
set) return Boolean is
     (isMember (f,t) and then isMember (e,f,r)) with
       Pre => ((for some x in t'Range => (x = f)) and
              (for some x in r'Range (1) => (x = e)) and
              (for some x in r'Range (2) => (x = f)));

   function inRangeRestriction (e : Integer; f : Integer; r : relation; g :
Integer) return Boolean is
     (f = g and then isMember (e,f,r)) with
       Pre => ((for some x in r'Range (1) => (x = e)) and
              (for some x in r'Range (2) => (x = f)));

   function equalsRangeRestriction (r1 : relation; r2 : relation; s : set)
return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inRangeRestriction (x,y,r2,s)) else (not (inRangeRestriction
(x,y,r2,s)))))) with
      Pre => ((for all x in r1'Range (2) => (for some y in s'Range => (x =
y))) and
              (for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
              (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function equalsRangeRestriction (r1 : relation; r2 : relation; e :
Integer) return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inRangeRestriction (x,y,r2,e)) else (not (inRangeRestriction
(x,y,r2,e)))))) with
      Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
              (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function inRangeSubtraction (e : Integer; f : Integer; r : relation; t :
```

```
set) return Boolean is
     ((not isMember (f,t)) and then isMember (e,f,r)) with
       Pre => ((for some x in t'Range => (x = f)) and
               (for some x in r'Range (1) => (x = e)) and
               (for some x in r'Range (2) => (x = f)));

   function inRangeSubtraction (e : Integer; f : Integer; r : relation; g :
Integer) return Boolean is
     ((f /= g) and then isMember (e,f,r)) with
       Pre => ((for some x in r'Range (1) => (x = e)) and
               (for some x in r'Range (2) => (x = f)));

   function equalsRangeSubtraction (r1 : relation; r2 : relation; s : set)
return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inRangeSubtraction (x,y,r2,s)) else (not (inRangeSubtraction
(x,y,r2,s)))))) with
       Pre => ((for all x in r1'Range (2) => (for some y in s'Range => (x =
y))) and
               (for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function equalsRangeSubtraction (r1 : relation; r2 : relation; g :
Integer) return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (if isMember
(x,y,r1) then (inRangeSubtraction (x,y,r2,g)) else (not (inRangeSubtraction
(x,y,r2,g)))))) with
       Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function inRelationalImage (e : Integer; r : relation; s : set) return
Boolean is
     (for some x in s'Range => (isMember (x,s) and then isMember (x,e,r)))
with
     Pre => ((for all x in s'Range => (for some y in r'Range (1) => (x = y)))
and
              (for some x in r'Range (2) => (x = e)));

   function inRelationalImage (e : Integer; r : relation; f : Integer) return
Boolean is
     (isMember (f,e,r)) with
       Pre => ((for some x in r'Range (1) => (x = f)) and
               (for some x in r'Range (2) => (x = e)));

   function equalsRelationalImage (s : set; r : relation; t : set) return
Boolean is
     (for all x in s'Range => (if isMember (x,s) then (inRelationalImage
(x,r,t)) else (not (inRelationalImage (x,r,t))))) with
     Pre => ((for all x in s'Range => (for some y in r'Range (2) => (x = y)))
```

```
and
             (for all x in t'Range => (for some y in r'Range (1) => (x =
y))));

   function equalsRelationalImage (s : set; r : relation; e : Integer) return
Boolean is
     (for all x in s'Range => (if isMember (x,s) then (inRelationalImage
(x,r,e)) else (not (inRelationalImage (x,r,e))))) with
     Pre => ((for all x in s'Range => (for some y in r'Range (2) => (x = y)))
and
             (for some x in r'Range (1) => (x = e)));

   function inComposition (e : Integer; f : Integer; r1 : relation; r2 :
relation) return Boolean is
     (for some x in r1'Range (2) => (isMember (e,x,r1) and then isMember
(x,f,r2))) with
       Pre => ((for all x in r1'Range (2) => (for some y in r2'Range (1) =>
(x = y))) and
               (for some x in r1'Range (1) => (x = e)) and
               (for some x in r2'Range (2) => (x = f)));

   function inOverride (e : Integer; f : Integer; p : relation; q : relation)
return Boolean is
     (isMember (e,f,q) or else ((not (inDomain (e,q))) and then isMember
(e,f,p))) with
       Pre => ((for some x in q'Range (1) => (x = e)) and
               (for some x in q'Range (2) => (x = f)) and
               (for some x in p'Range (1) => (x = e)) and
               (for some x in p'Range (2) => (x = f)));

   function isPartialFunction (r : relation; s : set; t : set) return Boolean
is
     ((relationOfSets (r,s,t)) and then (for all x in r'Range (1) => (for all
y in r'Range (2) => (for all z in r'Range (2) => (if (r (x,y)) and then r
(x,z) then y = z))))) with
     Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x = y)))
and
               (for all x in r'Range (2) => (for some y in t'Range => (x =
y))));

   function isTotalFunction (r : relation; s : set; t : set) return Boolean
is
     ((isPartialFunction (r,s,t)) and then setEqualsDomain (s,r)) with
       Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x =
y))) and
               (for all x in r'Range (2) => (for some y in t'Range => (x =
y))) and
               (for all x in s'Range => (for some y in r'Range (1) => (x =
y))));

   function isPartialInjection (r : relation; s : set; t : set) return
Boolean is
     ((isPartialFunction (r,s,t)) and then (for all x in r'Range (1) => (for
```

```
all y in r'Range (1) => (for all z in r'Range (2) => (if isMember (x,z,r) and
then isMember (y,z,r) then x = y))))) with
     Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x = y)))
and
             (for all x in r'Range (2) => (for some y in t'Range => (x =
y))));

   function isTotalInjection (r : relation; s : set; t : set) return Boolean
is
     ((isPartialInjection (r,s,t)) and then (setEqualsDomain (s, r))) with
     Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x = y)))
and
             (for all x in r'Range (2) => (for some y in t'Range => (x = y)))
and
             (for all x in s'Range => (for some y in r'Range (1) => (x =
y))));

   function isPartialSurjection (r : relation; s : set; t : set) return
Boolean is
     ((isPartialFunction (r,s,t)) and then (setEqualsRange (t,r))) with
     Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x = y)))
and
             (for all x in r'Range (2) => (for some y in t'Range => (x = y)))
and
             (for all x in t'Range => (for some y in r'Range (2) => (x =
y))));

   function isTotalSurjection (r : relation; s : set; t : set) return Boolean
is
     ((isTotalFunction (r,s,t)) and then (setEqualsRange (t,r))) with
     Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x = y)))
and
             (for all x in r'Range (2) => (for some y in t'Range => (x = y)))
and
             (for all x in s'Range => (for some y in r'Range (1) => (x = y)))
and
             (for all x in t'Range => (for some y in r'Range (2) => (x =
y))));




   function isBijection (r : relation; s : set; t : set) return Boolean is
     ((isTotalInjection (r,s,t)) and then (setEqualsRange (t,r))) with
     Pre => ((for all x in r'Range (1) => (for some y in s'Range => (x = y)))
and
             (for all x in r'Range (2) => (for some y in t'Range => (x = y)))
and
             (for all x in s'Range => (for some y in r'Range (1) => (x = y)))
and
             (for all x in t'Range => (for some y in r'Range (2) => (x =
y))));
```

```
   function unionEmpty (s : set; t : set) return Boolean is
     (for all x in s'Range => (not (isMember (x,s) or else isMember (x,t))))
with
       Pre => (for all x in s'Range => (for some y in t'Range => (x = y)));

   function intersectionEmpty (s : set; t : set) return Boolean is
     (for all x in s'Range => (not (isMember (x,s) and then isMember (x,t))))
with
       Pre => (for all x in s'Range => (for some y in t'Range => (x = y)));

   function partition (s1 : set; s2 : set; s3 : set) return Boolean is
     ((equalsUnion (s1,s2,s3)) and then (intersectionEmpty (s2,s3))) with
     Pre => ((for all x in s1'Range => (for some y in s2'Range => (x = y)))
and
             (for all x in s1'Range => (for some y in s3'Range => (x = y)))
and
             (for all x in s2'Range => (for some y in s3'Range => (x = y))));

   function unionEmpty (r1 : relation; r2 : relation) return Boolean is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (not
(isMember (x,y,r1) or else isMember (x,y,r2))))) with
       Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function intersectionEmpty (r1 : relation; r2 : relation) return Boolean
is
     (for all x in r1'Range (1) => (for all y in r1'Range (2) => (not
(isMember (x,y,r1) and then isMember (x,y,r2))))) with
        Pre => ((for all x in r1'Range (1) => (for some y in r2'Range (1) =>
(x = y))) and
               (for all x in r1'Range (2) => (for some y in r2'Range (2) =>
(x = y))));

   function partition (r1 : relation; r2 : relation; r3 : relation) return
Boolean is
     ((equalsUnion (r1,r2,r3)) and then (intersectionEmpty (r2,r3))) with
     Pre => ((for all x in r2'Range (1) => (for some y in r3'Range (1) => (x
= y))) and
             (for all x in r2'Range (2) => (for some y in r3'Range (2) => (x
= y))) and
             (for all x in r1'Range (1) => (for some y in r2'Range (1) => (x
= y))) and
             (for all x in r1'Range (2) => (for some y in r2'Range (2) => (x
= y))) and
             (for all x in r1'Range (1) => (for some y in r3'Range (1) => (x
= y))) and
             (for all x in r1'Range (2) => (for some y in r3'Range (2) => (x
= y))));

   function functionApplicationEquality (r : relation; a : Integer; s :
relation; b : Integer) return Boolean is
```

```
     (for some x in r'Range (2) => (isMember (a,x,r) and then isMember
(b,x,s))) with
       Pre => ((for some x in r'Range (1) => (x = a)) and
               (for some x in s'Range (1) => (x = b)) and
               (for all x in r'Range (2) => (for some y in s'Range (2) => (x
= y))));
```

A.5 Examples of Translation

The following section shows the translation process for the different types of Event-B constructs into SPARK code. To save space, not all construct translations will be shown, and for each construct translation shown, only one example from the running example is used.

Initially, the SparkTranslate class's execute method obtains the machine root object representing the model, and uses the methods found in org.eventb.core to get the statically checked elements needed. The execute method then passes these elements to the relevant methods to extract the information needed to generate SPARK code.

A.5.1 Carrier Set - PERSON

The SparkTranslate class passes the list of carrier sets to the getCarrierSets method. This method reads the ISCCarrierSet object representing PERSON, and creates a CarrierSet object which holds the name "PERSONcs", a typename "PERSONtype" and a cardinality value "100", all as Strings. The cardinality value is derived from reading the list of ISCAxiom objects from the machine root and finding the relevant axiom.

This CarrierSet object is then added to a Spec object, which a Translator object then uses. The Translator object uses the information in the CarrierSet object to make the relevant subtype and variable declarations as per the translation process in section 4.3.1.

A.5.2 Invariant - location ⊆ permission

The execute method passes a list of ISCInvariant objects to the method getInvariants, one of which represents the invariant location ⊆ permission. This

method translates the predicate using the translate method, obtaining a String representing the predicate in SPARK code, and also gets a list of free identifiers found in the predicate as an ArrayList of Strings. This method creates an Invariant object using this String and ArrayList of Strings. In this case, the predicate will be “isSubset (location, permission)” and the two free identifiers will be “location” and “permission”.

The Translator class uses this Invariant object to make the SPARK function representing this invariant, as per the rules in section 4.3.4.

A.5.3 Event - AddPerson

The most complex Event-B constructs to translate are the events themselves.

The getEvents method uses the information in the ISCEvent object to make the Event object. The Event object is instantiated using the following information: the name of the Event as a String, the list of parameters of the event as mappings of the parameter name to its type as a Map<String,String>, the list of predicates representing the guards of the event in SPARK form as an ArrayList of Strings, another ArrayList of Strings representing the before-after predicates of the actions in SPARK form, a list of variables read by the event as an ArrayList of Strings, a list of variables assigned to by the event as an ArrayList of Strings, and finally, a list of dependencies of every variable assigned to by this event as a Map<String,ArrayList<String>>, a map which maps every variable which is changed by this event, as a String, to the list of parameters and variables which determine the variable’s new value, as an ArrayList of Strings. The list of guards and before-after predicates of the actions are obtained by getting a list of guards and action predicates respectively, and passing them to the translate method, similar to how invariants and axioms are translated.

For this example, the information used in making the Event object shall be:

- “AddPerson” : String
- [“p” => “PERSONtype”] : Map<String,String>
- [“isMemberDifference (p,PERSONcs,person)”] : ArrayList<String>
- [“equalsUnion (person,person’Old,p)”, “equalsUnion (outside,outside’Old,p)”] : ArrayList<String>
- [“person”, “outside”] : ArrayList<String>
- [“person”, “outside”] : ArrayList<String>

- [“person” => [“person”, “p”], “outside” => [“outside”, “p”]] : Map<String,ArrayList<String>>

The Translator class uses this information from the Event object to construct the correct SPARK procedure specification, as per the rules in section 4.3.6. Note that the Translator class must obtain a list of all carrier sets, constants and variables in the model, so as to place all of these constructs which are not in the input or output ArrayLists of this Event object in the Proof_In aspect of the SPARK procedure. This carries the assumption that all carrier sets, constants and variables will be used by either an axiom or an invariant, and so will appear in the Proof_In aspect if it is not read or written to directly by the event.

A.6 Tests on Other Event-B Models

A.6.1 Room Booking System

Event-B Model:

Carrier Sets BOOKING
Constants
Axioms
Axm1: finite(BOOKING)
Axm2: card(BOOKING) = 500
Variables booking closed used inroom
Invariants
Inv1: booking ⊆ BOOKING
Inv2: closed ⊆ booking
Inv3: used ⊆ booking
Inv4: inroom ∈ BOOL
Inv5: ∀b1,b2 · b1 ∈ booking∖closed ∧ b2 ∈ booking∖closed ⇒ b1 = b2
Events

INITIALISATION
THEN
Act1: booking ≔ ∅
Act2: closed ≔ ∅
Act3: used ≔ ∅
Act4: inroom ≔ FALSE

END

check_in
ANY
B
WHERE
Grd1: b ∉ booking
Grd2: closed = booking
THEN
Act1: booking ≔ booking ∪ {b}
END
check_out
ANY
b
WHERE
Grd1: b ∈ booking∖closed
THEN
Act1: closed ≔ closed ∪ {b}
END

use
ANY
b
WHERE
Grd1: b ∈ booking∖used
Grd2: inroom = FALSE
THEN
Act1: used ≔ used ∪ {b}
Act2: inroom ≔ TRUE
END

re_enter
WHERE
Grd1: inroom = FALSE
THEN
Act1: inroom ≔ TRUE

END

leave
WHERE
Grd1: inroom = TRUE
THEN
Act1: inroom := FALSE
END

SPARK code generated:

```
subtype BOOKINGtype is Integer range 1 .. 500;

BOOKINGcs : set (BOOKINGtype) := (others => True);
function cs return Boolean is
       (isFullSet (BOOKINGcs)) with
              Global => (BOOKINGcs),
              Depends => (cs'Result => (BOOKINGcs));

booking : set (BOOKINGtype);
closed : set (BOOKINGtype);
inroom : Boolean;
used : set (BOOKINGtype);

function Inv1 return Boolean is
       (isSubset (booking,BOOKINGcs)) with
              Global => (booking,BOOKINGcs),
              Depends => (Inv1'Result => (booking,BOOKINGcs));

function Inv2 return Boolean is
       (isSubset (closed,booking)) with
              Global => (closed,booking),
              Depends => (Inv2'Result => (closed,booking));

function Inv3 return Boolean is
       (isSubset (used,booking)) with
              Global => (used,booking),
              Depends => (Inv3'Result => (used,booking));

function Inv4 return Boolean is
       (for all b1 in BOOKINGtype => (for all b2 in BOOKINGtype => ((if
(isMemberDifference (b1,booking,closed) and then isMemberDifference
(b2,booking,closed)) then (b1 = b2))))) with
              Global => (booking,closed),
```

```
              Depends => (Inv4'Result => (booking,closed));

procedure INITIALISATION with
       Pre => (cs),
       Post => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4
and then
              isEmpty (booking) and then
              isEmpty (closed) and then
              isEmpty (used) and then
              inroom = False),
       Global => (Proof_In => (BOOKINGcs),
              Output => (booking,closed,inroom,used)),
       Depends => (inroom => null,booking => null,closed => null,used =>
null);

procedure check_in (b : in BOOKINGtype) with
       Pre => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then
              not (booking (b)) and then
              closed = booking),
       Post => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4
and then
              equalsUnion (booking,booking'Old,b)),
       Global => (Proof_In => (BOOKINGcs,closed,inroom,used),
              In_Out => (booking)),
       Depends => (booking =>+ (b));

procedure check_out (b : in BOOKINGtype) with
       Pre => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then
              isMemberDifference (b,booking,closed)),
       Post => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4
and then
              equalsUnion (closed,closed'Old,b)),
       Global => (Proof_In => (BOOKINGcs,booking,inroom,used),
              In_Out => (closed)),
       Depends => (closed =>+ (b));

procedure use (b : in BOOKINGtype) with
       Pre => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then
              isMemberDifference (b,booking,used) and then
              inroom = False),
       Post => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4
and then
              equalsUnion (used,used'Old,b) and then
              inroom = True),
       Global => (Proof_In => (BOOKINGcs,booking,closed),
              Output => (inroom),
              In_Out => (used)),
       Depends => (inroom => null,used =>+ (b));

procedure re_enter with
```

```
        Pre => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then
                inroom = False),
        Post => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4
and then
                inroom = True),
        Global => (Proof_In => (BOOKINGcs,booking,closed,used),
                Output => (inroom)),
        Depends => (inroom => null);

procedure leave with
        Pre => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4 and
then
                inroom = True),
        Post => (cs and then Inv1 and then Inv2 and then Inv3 and then Inv4
and then
                inroom = False),
        Global => (Proof_In => (BOOKINGcs,booking,closed,used),
                Output => (inroom)),
        Depends => (inroom => null);
```

A.6.2 Club Courses System

Event-B Model:

Carrier Sets CRS
Constants m
Axioms
    Axm1: finite(CRS)
    Axm2: card(CRS) = 30
    Axm3: m = FALSE
Variables crs
Invariants
    Inv1: crs ∈ ℙ(CRS)
Events

INITIALISATION
THEN
    Act1: crs ≔ ∅
END

OpenCourses
ANY
    cs

WHERE

Grd1: cs ⊆ CRS

Grd2: crs ∩ cs = ∅

Grd3: cs ≠ ∅

THEN

Act1: crs ≔ crs ∪ cs

END

CloseCourses

ANY

cs

WHERE

Grd1: cs ⊆ crs

Grd2: cs ≠ ∅

THEN

Act1: crs ≔ crs ∖ cs

END

SPARK code generated:

```
subtype CRStype is Integer range 1 .. 30;

CRScs : set (CRStype) := (others => True);

function cs return Boolean is
       (isFullSet (CRScs)) with
              Global => (CRScs),
              Depends => (cs'Result => (CRScs));

m : constant Boolean := False;

function Axm1 return Boolean is
       (m = False) with
              Global => (m),
              Depends => (Axm1'Result => (m));

crs : set (CRStype);

function Inv1 return Boolean is
       (inPowerSet (crs,CRScs)) with
              Global => (crs,CRScs),
              Depends => (Inv1'Result => (crs,CRScs));

procedure INITIALISATION with
       Pre => (cs and then Axm1),
       Post => (cs and then Axm1 and then Inv1 and then
              isEmpty (crs)),
       Global => (Proof_In => (CRScs,m),
              Output => (crs)),
```

```
        Depends => (crs => null);




procedure OpenCourses (cs : in set (CRStype)) with
        Pre => (cs and then Axm1 and then Inv1 and then
                isSubset (cs,CRScs) and then
                intersectionEmpty (crs,cs) and then
                not (isEmpty (cs))),
        Post => (cs and then Axm1 and then Inv1 and then
                equalsUnion (crs,crs'Old,cs)),
        Global => (Proof_In => (CRScs,m),
                In_Out => (crs)),
        Depends => (crs =>+ (cs));

procedure CloseCourses (cs : in set (CRStype)) with
        Pre => (cs and then Axm1 and then Inv1 and then
                isSubset (cs,crs) and then
                not (isEmpty (cs))),
        Post => (cs and then Axm1 and then Inv1 and then
                equalsDifference (crs,crs'Old,cs)),
        Global => (Proof_In => (CRScs,m),
                In_Out => (crs)),
        Depends => (crs =>+ (cs));
```

A.6.3 Island Car Access System

Event-B Model:

Carrier Sets CAR
Constants
Axioms
Axm1: finite(CAR)
Axm2: card(CAR) = 500
Variables IL_CARS B
Invariants
Inv1: IL_CARS ⊆ CAR
Inv2: (∃ car·car ∈ IL_CARS) ∨ (∃ car·car ∉ IL_CARS)
Inv3: B = 500

Events

INITIALISATION
THEN

Act1: IL_CARS ≔ ∅
Act2: B ≔ 500
END

IL_IN
ANY
car
WHERE
Grd1: car ∉ IL_CARS
THEN
Act1: IL_CARS ≔ IL_CARS ∪ {car}
END

IL_OUT
ANY
car
WHERE
Grd1: car ∈ IL_CARS
THEN
Act1: IL_CARS ≔ IL_CARS ∖ {car}
END

SPARK code generated:

```
subtype CARtype is Integer range 1 .. 500;

CARcs : set (CARtype) := (others => True);

function cs return Boolean is
       (isFullSet (CARcs)) with
              Global => (CARcs),
              Depends => (cs'Result => (CARcs));

B : Integer;
IL_CARS : set (CARtype);



function Inv1 return Boolean is
       (isSubset (IL_CARS,CARcs)) with
              Global => (IL_CARS,CARcs),
              Depends => (Inv1'Result => (IL_CARS,CARcs));
```

```
function Inv2 return Boolean is
	((for some car in CARtype => (IL_CARS (car))) or else (for some car in
CARtype => (not (IL_CARS (car))))) with
		Global => (IL_CARS),
		Depends => (Inv2'Result => (IL_CARS));


function Inv3 return Boolean is
	(B = 500) with
		Global => (B),
		Depends => (Inv3'Result => (B));

procedure INITIALISATION with
	Pre => (cs),
	Post => (cs and then Inv1 and then Inv2 and then Inv3 and then
		isEmpty (IL_CARS) and then
		B = 500),
	Global => (Proof_In => (CARcs),
		Output => (B,IL_CARS)),
	Depends => (B => null,IL_CARS => null);

procedure IL_IN (car : in CARtype) with
	Pre => (cs and then Inv1 and then Inv2 and then Inv3 and then
		not (IL_CARS (car))),
	Post => (cs and then Inv1 and then Inv2 and then Inv3 and then
		equalsUnion (IL_CARS,IL_CARS'Old,car)),
	Global => (Proof_In => (CARcs,B),
		In_Out => (IL_CARS)),
	Depends => (IL_CARS =>+ (car));

procedure IL_OUT (car : in CARtype) with
	Pre => (cs and then Inv1 and then Inv2 and then Inv3 and then
		IL_CARS (car)),
	Post => (cs and then Inv1 and then Inv2 and then Inv3 and then
		equalsDifference (IL_CARS,IL_CARS'Old,car)),
	Global => (Proof_In => (CARcs,B),
		In_Out => (IL_CARS)),
	Depends => (IL_CARS =>+ (car));
```

A.7 Original Project Brief

The problem:

It is important to ensure that the software in safety- and security-critical systems behaves correctly, as any errors may have serious consequences. The Event-B language is a modelling language which is used to design software systems. It uses mathematical proofs to ensure that designs behave as intended. SPARK is a

programming language with a number of static analysis tools which is used to verify that programs written in it behave correctly. SPARK is a subset of Ada, also a language used for safety-critical systems (Dross and Moy, 2019). SPARK uses proof annotations as assertions to formally verify the correct behaviour of its sub-programs. These proof annotations may take the form of pre- and post-conditions, as well as loop invariants. However, manually generating these can be time-consuming, and so this method of ensuring the safety of software may be limited by time constraints (Murali and Ireland, 2012).

Goals:
The overall goal is to build a tool which can automatically convert pre-defined and verifiably-correct Event-B designs of software systems into corresponding proof annotations, namely a set of pre- and post-conditions, in the SPARK language, which any program implementing the system written in the SPARK language can use as the contracts for their sub-programs. This automation would save the time and effort needed to derive these proof annotations manually from the design of the system.

The plan to achieve this goal is broken down into several objectives:

1) Research the SPARK and Ada languages, and understand how the static analysis tools use given proof annotations to verify the correctness of the written programs.
2) Extend the set of known translation rules from Event-B to SPARK.
3) Research and understand the use of the Eclipse Modelling Framework (EMF) for Event-B and how this can be used to simplify the process of translating from Event-B to SPARK.
4) Implement the tool as a Rodin plugin for convenience of translating Event-B models.
5) Test and evaluate the tool by applying it to different case studies, as well as some own examples.

The scope:
This project is limited to only translating Event-B models into SPARK proof annotations. The project does not involve automatic code generation in SPARK, as realistically, this would involve the Event-B models having to be extremely concrete in its design. This project has a greater focus on translating more abstract Event-B designs into SPARK proof annotations, and while it may be possible to generate implementing programs from such designs, it is infeasible for this project due to time constraints. Building a tool which translates more abstract designs is in a sense more powerful, as it can be used more generally. Another aspect of this project is building on existing sets of translation rules from Event-B to SPARK proof annotations. Currently-existing translation rules are

limited in their scope due to the concrete nature of the Event-B designs included. This project will aim to shed light on greater insight into the relationship between structures of both languages. Finally, the proof annotations generated will only be pre- and post-conditions for the SPARK programs. If time permits, there will be an attempt to try and include loop invariants in the set of auto-generated proof annotations as well, if possible.

References:

- Dross, C. and Moy, Y. (2019). SPARK Overview. [online] Learn.Adacore. Available at: https://learn.adacore.com/courses/intro-to-spark/chapters/01_Overview.html [Accessed 6 Oct. 2019].

- Murali, R. and Ireland, A. (2012). E-SPARK: Automated Generation of Provably Correct Code from Formally Verified Designs. Electronic Communications of the EASST, [online] Volume 53: Automated Verification of Critical Systems 2012, p. 1. Available at: https://pdfs.semanticscholar.org/481c/d4d2409115429f4b824f370eb08fa338d67a.pdf [Accessed 6 Oct. 2019].

A.8 Contents of Design and Data Archive

The design archive contains all the files containing the code developed for the translation plug-in, as well as the test files for the translate method. Furthermore, for the running building access example, the archive contains the SPARK specification and body files. Also included is the sr.ads file.

- Java files
    - constructs
        - Axiom.java
        - CarrierSet.java
        - Constant.java
        - Construct.java
        - Event.java
        - Invariant.java
        - Predicate.java
        - Spec.java
        - Variable.java
    - spark
        - SparkTranslate.java
        - Translator.java
    - tests
        - SubsetTests.java
        - TestAndOr.java
        - TestBoolValues.java
        - TestCarrierSets.java
        - TestCenter.java
        - TestEquals.java
        - TestEquiv.java
        - TestImplies.java
        - TestIn.java
        - TestPartition.java
        - TestQuantifiers.java
- SPARK files
    - test9.ads
    - test9.adb
    - sr.ads